Rule Condition Testing and Action Execution in Ariel[†]

Eric N. Hanson
Database Systems Research and Development Center
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611

Abstract

This paper describes testing of rule conditions and execution of rule actions in the Ariel active DBMS. The Ariel rule system is tightly coupled with query and update processing. Ariel rules can have conditions based on a mix of patterns, events, and transitions. For testing rule conditions, Ariel makes use of a discrimination network composed of a special data structure for testing single-relation selection conditions efficiently, and a modified version of the TREAT algorithm, called A-TREAT, for testing join conditions. The key modification to TREAT (which could also be used in the Rete algorithm) is the use of *virtual* α-memory nodes which save storage since they contain only the predicate associated with the memory node instead of copies of data matching the predicate. The rule-action executor in Ariel binds the data matching a rule's condition to the action of the rule at rule fire time, and executes the rule action using the query processor.

1 Introduction

Designers of database management systems have long wanted to transform databases from passive repositories for data into *active* systems that can respond immediately to a change in the state of the data, an event, or a transition between states [5]. However, to create a successful active database system, many problems must be solved, including:

- design of a suitable language for expressing active rules,
- design of a condition-testing mechanism for rules that is efficient enough to still allow fast transaction processing, and
- integration of rule condition testing and execution with the transaction processing system.

[†]This work was supported in part by the Air Force Office of Scientific Research under grant number AFOSR-89-0286.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0049...$1.50

The Ariel system is an implementation of a relational DBMS with a built-in rule system which has been designed to address the above issues. The Ariel rule system is based on the production system model [7]. Our approach has been to adopt as much as possible from previous work on main-memory production systems such as OPS5 [6], but make changes where necessary to improve the functionality and performance of a production system in a database environment. The features of Ariel that distinguish it from other commercial and research active database rule systems are the following:

- Ariel is a complete implementation of a relational DBMS with a rule system that is *tightly coupled* with the query processor,
- the design of Ariel places strong emphasis on efficient testing of rule conditions in a database environment, and a high-performance discrimination network for testing rule conditions in that environment has been designed and implemented.

This paper emphasizes the testing of join conditions of rules and execution of rule actions in Ariel. Section 2 describes the query and rule languages used in Ariel. Section 3 gives an overview of the Ariel system architecture. Section 4 presents the structure of tokens that are created by database operations, and the discrimination network used in Ariel for efficiently testing both selection and join conditions of rules against those tokens. Section 5 describes optimization and execution of rule actions. Section 6 describes the status of the implementation and gives some performance results. Finally, section 7 discusses related research, and section 8 summarizes and presents conclusions.

2 The Ariel Query and Rule Languages

To simplify and narrow the scope of the Ariel project, we chose to support the relational data model and provide a subset of the POSTQUEL query language of POSTGRES for specifying data definition commands, queries and updates [21]. POSTQUEL query and update commands **retrieve**, **append**, **delete**, and **replace**, are supported, along with commands for creating and destroying relations, and performing other utility functions. We extended POSTQUEL with a production-rule language called the Ariel Rule Language (ARL) to be discussed next. ARL is not related to the POSTGRES rule language [19, 20].

2.1 Rule Language

ARL is a production-rule language with enhancements for defining rules with conditions based not only on patterns, but also on events and transitions. The ARL syntax is based on the syntax of the query language. The general form of an ARL rule is the following:

define rule *rule-name* [**in** *ruleset-name*]
[**priority** *priority-val*]
[**on** *event*]
[**if** *condition*]
then *action*

A unique *rule-name* is required for each rule so the rule can be referred to later by the user. The user can optionally specify a *ruleset name* to place the rule in a ruleset. Rulesets are simply a means of grouping rules together for programmer convenience. If no ruleset name is specified, the rule is placed in the system-defined ruleset *default_rules*. The **priority** clause allows specification of a priority to control the order of rule execution. The **on** clause allows specification of an event that will trigger the rule. The following types of events can be specified after an **on** clause:

- **append** [**to**] *relation-name*
- **delete** [**from**] *relation-name*
- **replace** [**to**] *relation-name* [(*attribute-list*)]

The *condition* after the **if** clause has the following form:

qualification [**from** *from-list*]

The syntax of the pattern in a rule condition is identical to that for the **where** clause of a query, with minor exceptions. The **from** clause is for specifying bindings of tuple variables to relations. Relation names can be used as default tuple variables in both rules and queries. The **then** part of the rule contains the action to be performed when the rule fires. The action can be a single data manipulation command, or a *compound command* which is a **do** ... **end** block surrounding a list of commands.

There will be cases where a rule must be awakened when any new tuple value is created in a relation (due to an **append** or a **replace**). For this case, the following conditional expression to reference a relation is provided:

new (*tuple-variable*)

New can be thought of as a selection condition which is always "true."

2.2 Rule Semantics

The Ariel rule system uses a production system model, where the "working memory" is stored in the database relations and rules are stored separately in the *rule catalog*. Execution of rules is governed by a *recognize-act cycle* similar to that used in OPS5 [7]. Ariel rules get an opportunity to wake up after every database *transition*. Below, we describe in detail Ariel's treatment of transitions, events and the rule execution cycle.

2.2.1 Transitions

A transition in Ariel is defined to be the changes in the database induced by either a single command, or a **do ... end** block containing a list of simple commands. Blocks may not be nested. The programmer designing a database transaction thus has control over where transitions occur. If desired, the programmer can put a **do ... end** block around all the commands in the transaction so the entire transaction is a single transition. Each command in a transaction will be considered a transition by itself unless it is enclosed in a block. Blocks are provided to allow programmers to safely update the database with multiple commands when data integrity or consistency might be temporarily violated during the update. Programmers are encouraged to only put a block around groups of commands which might violate integrity or consistency, since use of blocks does incur some performance overhead.

2.2.2 Logical vs. Physical Events

In Ariel, triggering of event-based rules is based on *logical* events rather than physical events. Logical events are defined as follows. The life of an individual tuple t updated by a single transition always falls in one of the following four categories, where i, m and d represent insertion, modification, and deletion respectively. Superscripts $*$ and $+$ indicate a sequence of zero or more and one or more individual updates, respectively.

update type	description	net effect
im^*	insertion of t followed by zero or more modifications	insert
im^*d	insertion of t followed by zero or more modifications and then deletion.	nothing
m^+	t existed at the beginning of the transition and was modified one or more times.	modify
m^*d	t existed at the beginning of the transition, was modified zero or more times, and then deleted.	delete

The table above shows how the net effect of a sequence of updates to one tuple can be summarized as a single insert, delete or modify operation, or no operation.

We made the decision to use logical rather than physical events for the following reasons:

1. When multiple event-based rules triggered by the same event are active, execution of one rule may invalidate (e.g., delete) the data bound to another. If all binding of data to event-based rules occurs at the time the event occurs, there is no way to avoid execution of rules bound to data that is no longer valid. If events are treated as logical events as defined in the table above, rules are always bound to valid data when they execute.
2. Treating events as logical operations provides additional data integrity compared with treating them as physical operations. Consider the example relations below that will be

used in the rest of the paper, and the example rule that follows:

```
emp(name, age, salary, dno, jno)
dept(dno, name, building)
job(jno, title, paygrade, description)

define rule NoBobs
on append emp
if emp.name = "Bob"
then delete emp
```

The effect of this rule is to never let anyone named "Bob" be appended to the emp relation. Consider the following block of update commands:

```
do
  append emp(name="", age=27,
    sal=55000, dno = 12)
  replace emp (name="Bob")
    where emp.name = ""
end
```

If events are interpreted as physical operations, then this sequence of commands will not trigger rule NoBobs. However, NoBobs will be triggered if the block is treated as the following single logical event:

```
append emp(name="Bob", age=27,
      sal=55000, dno = 12)
```

In general, interpretation of events as logical rather than physical is expected to be more intuitive and easy to use for rule programmers, since they will only have to be concerned with *effects* of database operations, not the *expression* of them. Since many different sequences of commands can have the same effect, considering only the logical effects of updates will simplify design of event-based rules.

The above example also shows that it can be difficult to specify event-based rules to achieve a desired goal (e.g., ensuring that there is no one named "Bob" in the emp relation). Hence, we recommend use of purely *pattern-based* rules whenever possible, since they will be triggered whenever any data matches a specific pattern, regardless of the event that created or modified the data. An alternative to the NoBobs rule that is purely pattern-based is the following:

```
define rule NoBobs2
if emp.name = "Bob"
then delete emp
```

This rule deletes all emp records with name "Bob" whether they are created by an **append** or a **replace** command.

2.2.3 The Rule Execution Cycle

Ariel controls rule execution using the *recognize-act cycle*, shown in Figure 1, which is commonly used in production systems [6]. The *match* step finds the set of rules that are eligible to run. The *conflict resolution* step selects a single rule for execution from the set of eligible rules. Finally, the *act* step executes the statements in the rule action. The cycle repeats until no rules are eligible to run, or the system executes an explicit halt. The discrimination network used in the match phase is discussed in section 4. In Ariel, data

```
match
while (rules left to run and not halt executed) do
      conflict resolution
      act
      match
end
```

Figure 1: The recognize-act cycle.

matching the rule condition is stored in a temporary relation called the *P-node*. In the *act* phase, the statement(s) in the **then** part of the rule are bound to the P-node for the rule by a process of query modification [18]. The modified syntax tree for the command is then passed to the query optimizer which generates an optimal query execution plan. The plan is then interpreted to carry out the command. Details of the query modification procedure will be discussed in section 5.

2.2.4 Event and Transition Conditions

One feature of Ariel that distinguishes it from most other active database rule systems is support for event and transition conditions that is fully integrated with pure pattern-based rule condition testing. ARL provides a special keyword **previous** for referring to the previous value of an attribute. The value that a tuple attribute had at the beginning of a transition can be accessed using the following notation:

previous *tuple-variable.attribute*

An example of a rule with a transition condition in it is:

```
define rule raiseLimit
if emp.sal > 1.1 * previous emp.sal
then append to salaryError(emp.name,
      previous emp.sal, emp.sal)
```

The affect of this rule is to place the name and new/old salary pair of every employee that received a raise of greater than ten percent in a relation salaryError. Other rules could be defined to trigger on appends to salaryError to take an appropriate action, such as reversing the update, or notifying a person to verify the correctness of the update.

As an example of how pattern-based conditions and transition conditions can be combined, suppose we wished to make the raiseLimit rule specific to just the Toy department. This can be done using a normal pattern-based condition to select the Toy department, and joining the resulting tuples to the emp tuple variable in the normal fashion. A rule that does this is the following:

```
define rule toyRaiseLimit
if emp.sal > 1.1 * previous emp.sal
and emp.dno = dept.dno
and dept.name = "Toy"
then append to toySalaryError(emp.name,
      previous emp.sal, emp.sal)
```

Moreover, event, pattern and transition conditions can all be combined. Consider this example of a rule that uses all three types of conditions to log "demotion" of an employee in the demotions relation:

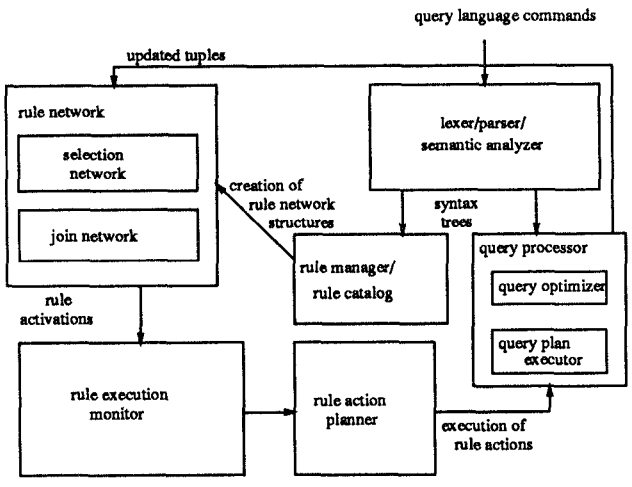

Figure 2: Diagram of the Ariel system architecture.

```
define rule findDemotions
on replace emp(jno)
if newjob.jno = emp.jno
and oldjob.jno = previous emp.jno
and newjob.paygrade < oldjob.paygrade
from oldjob in job, newjob in job
then append to demotions
        (name=emp.name, dno=emp.dno,
        oldjno=oldjob.jno, newjno=newjob.jno)
```

Similar to previous examples, other rules could be made to trigger when new tuples are appended to the demotions relation to take appropriate action.

In summary, ARL is a comprehensive active rule language for a relational DBMS which supports a set-oriented execution style, complex rule conditions combining patterns, events, and transitions, and execution of rule actions containing multiple database commands.

3 Architectural Overview

The architecture of Ariel, shown in Figure 2, is similar to that of System R [1] with additional components attached for rule processing. When commands enter Ariel they are processed by the lexer, parser, and semantic analyzer. If they are queries or data manipulation commands, they are passed to the query optimizer. Execution plans produced by the optimizer are carried out by the query plan executor. The executor is built on top of the storage system provided by the EXODUS database toolkit [2, 14]. In addition to the standard components, Ariel has a *rule manager* to handle creation and activation of rules, a *rule catalog* for maintaining the definitions of rules, a *discrimination network* for testing rule conditions, a *rule execution monitor* for carrying out rule execution, and a *rule action planner* for binding the data matching a rule condition with the rule action and producing an execution plan for that action. The discrimination network and the rule action planner are discussed in detail below.

4 The Discrimination Network

An efficient strategy for incrementally testing rule conditions as small changes in the database occur is critical for fast rule processing. Ariel contains a rule condition testing network called A-TREAT (short for Ariel TREAT) which is designed to both speed up rule processing in a database environment and reduce storage requirements compared with TREAT. An important performance optimization in A-TREAT is the use of a special top-level discrimination network for testing single-relation selection conditions of rules [11]. In addition, we introduce a technique for reducing the amount of state information stored in the network, whereby α-memory nodes are replaced in some cases by *virtual* α-memory nodes which contain only the predicate associated with the node, not the tuples matching the predicate. In addition to these performance enhancement techniques, we have developed some enhancements to the standard TREAT network in order to effectively test both transition and event-based conditions with a minimum of restrictions on how such conditions can be used.

4.1 Testing Selection Conditions

Ariel uses a special index optimized for testing selection conditions as the top layer in its discrimination network. This index makes use of an *interval index* called the *interval binary search tree* to efficiently test conditions that specify closed intervals (e.g., constant1 $<$ relation.attribute $\leq$ constant2), open intervals (e.g., constant $<$ relation.attribute) , or points (e.g., constant = relation.attribute). Readers are referred to [11] for a detailed discussion of the selection condition testing network in Ariel. A data structure called the *interval skip list* [9] can be used as an interval index in place of the interval binary search tree discussed in [11, 10]. The interval skip list is much easier to implement than the IBS tree and performs as well.

4.2 Saving Storage Using Virtual α-memories

Here we describe a variation of the Rete and TREAT algorithms for minimizing storage use in database rule systems. In the standard Rete and TREAT algorithms, there is an α-memory node for every selection condition on every tuple-variable present in a rule condition. If the selection conditions are highly selective, this is not a problem since the α-memories will be small. However, if selection conditions have low selectivity, then a large fraction of the tuples in the database will qualify, and α-memories will contain a large amount of data that is redundant since it is already stored in base tables. Storing large amounts of duplicate data is not acceptable in a database environment since the data tables themselves can be huge.

In order to avoid this problem, for memory nodes that would contain a large amount of data, a *virtual* memory node can be used which contains a predicate describing the contents of the node rather than the qualifying data itself. In a sense, this virtual node is a database view. When the virtual node is accessed, the (possibly modified) predicate stored in the node is processed to derive the value of the

node. The predicate can be modified by substituting constants from a token in place of variables in the predicate to make the predicate more selective and thus reduce processing time.

The algorithm for processing a single insertion token t in a TREAT network containing a mixture of stored and virtual α-memory nodes is as follows. A stored α-memory node contains a collection C of the tuples matching the associated selection predicate. A virtual α-memory node contains a selection predicate P and the identifier of the relation R on which P is defined. In addition, each transaction T maintains a data structure ProcessedMemories containing a set of the identifiers of the virtual α-memory nodes in which token t has been inserted. ProcessedMemories is emptied before processing of each token.

Suppose a single tuple X is to be inserted in R. *Before* putting X in R, create a token t from X and propagate t through the selection network. When t filters through the network to an α-memory node A, the identifier of A is placed in ProcessedMemories and then t is joined to neighboring α-memories. When joining t to a memory node A', if A' is a normal α-memory, everything proceeds as in the standard TREAT algorithm. If A' is virtual, then join t through to the base relation R' identified in A' using predicate P' of A' as a filter. In addition, if ProcessedMemories contains A', then t belongs to to A'. Hence, we must try to join the copy of t just placed in A to the copy of t in A'. If t joins to itself, a compound token is created and the process continues. At the end of processing t, empty ProcessedMemories, and then insert tuple X in R. An analogous procedure is used for processing a deletion (−) token.

The algorithm just described has the same effect as the normal TREAT strategy because at every step, a virtual α-memory node implicitly contains *exactly* the same set of tokens as a stored α-memory node. This ensures that if a token joins to itself, it does so exactly the right number of times. A TREAT-based join condition testing algorithm enhanced with virtual α-memories has been implemented in the Ariel system.

The following rule will be used to illustrate a standard TREAT network, and an A-TREAT network that accomplishes the same task:

define rule SalesClerkRule
if emp.sal > 30000
and emp.dno = dept.dno
and dept.name = "Sales"
and emp.jno = job.jno
and job.title ="Clerk"
then *action*

The TREAT network for the rule SalesClerkRule is shown in Figure 3. An A-TREAT network for the rule is shown in Figure 4. The A-TREAT network is identical to the TREAT network, except that the middle α-memory node (alpha2) is virtual, as indicated by the dashed box around it. If the predicate sal>30000 is not very selective, then making alpha2 be virtual may be a reasonable choice for SalesClerkRule since it can save a significant amount of storage.

The ability to use virtual memory nodes opens up several possible avenues of investigation. It allows trading space for time in a Rete or TREAT network. When to use a virtual

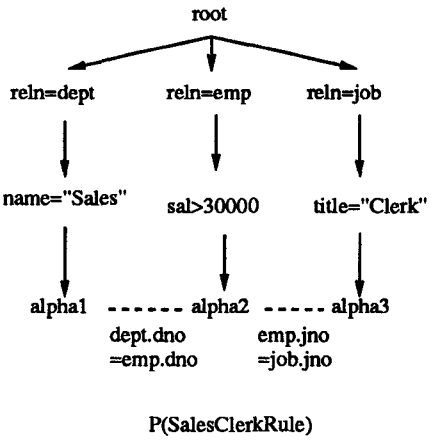

Figure 3: TREAT network for rule SalesClerkRule.

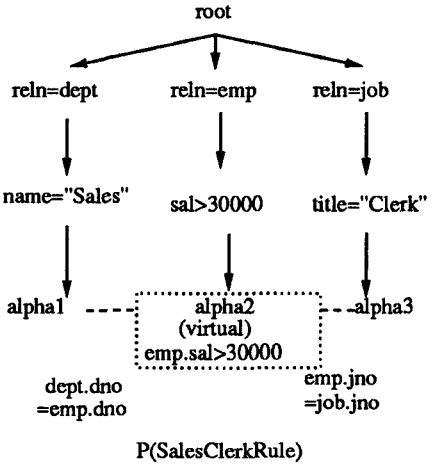

Figure 4: Example A-TREAT network.

memory node and when not to use one is an interesting optimization problem. Also, the base relation scan done when joining a token to a virtual α-memory can be done with any scan algorithm – index scan or sequential scan. Some optimization strategy is needed to decide whether or not to use an index if one is available, depending on the type of index (primary or secondary, hash or B-tree etc.) and the size of the base relation.

4.3 Testing Transition, Event, and Normal Conditions Together

Quite unlike standard production systems, Ariel allows rules with transition and event-based conditions in addition to normal conditions. To integrate all these types of conditions into a coherent framework, we generalized the notions of both tokens and α-memory nodes.

4.3.1 Identifying Transitions

To accommodate transitions, in addition to standard + and − tokens, Ariel uses $\Delta+$ and $\Delta-$ tokens which contain a (new,old) pair for a tuple with the value it had before

and after being updated. A Δ+-token inserts a new transition event into the rule network, and a Δ−-token removes a transition event from the rule network. In addition, all tokens have an event-specifier of one of the following forms to indicate the type of event which created the token:

- **append**
- **delete**
- **replace**(*target-list*)

The target-list included with the **replace** event specifier indicates which fields of the tuple contained in the token were updated. **On**-conditions in the top-level discrimination network are the only conditions that ever examine the event-specifier on a token. Tokens with their event-specifier are also called *eventTokens*.

In order to send the correct type of token through the network at the correct time, Ariel builds a data structure containing a pair of Δ-*sets* [I,M] for each relation updated during a transition. Set I contains an entry for each tuple which was inserted during the current transition. Set M contains an entry for each tuple that existed in the relation at the beginning of the transition and was modified during the transition. It is not necessary to maintain a third set for deletions since once a tuple is deleted it cannot be accessed again.

A Δ-set (I or M) contains a set of entries with the following contents:

eventSpecifier: one of **append** or **replace**(*target-list*), describing the type of event that created the entry,

isDelta: true or **false**,

tupleValue: a single tuple if isDelta is false, or a pair of old and new tuple values concatenated together if isDelta is true,

descriptor: a pointer to a format descriptor describing the locations of fields in tupleValue.

The possible sequences of operations that may occur to a single tuple during a transition are shown below [13]:

- *Case 1:* An insertion of a tuple t followed by one or more modifications of t (im^*). The net effect of this transition is an insertion. The first insert generates an $insert^+$ token, and each modify generates an $insert^-$ followed by an $insert^+$ containing the new tuple value.
  Example:

transition	eventTokens
insert t	($insert^+$)
modify t	($insert^-$, then $insert^+$)
modify t	($insert^-$, then $insert^+$)

- *Case 2*: A tuple t is inserted, modified one or more times, and then deleted (im^*d). The net effect is nothing. Tokens are generated as in Case 1, except that the final delete operation generates an $insert^-$ token.
  Example:

transition	eventTokens
insert t	($insert^+$)
modify t	($insert^-$, then $insert^+$)
delete t	($insert^-$)

- *Case 3*: Tuple t exists prior to a transition in which it is modified one or more times (m^+). The net effect is a modification. The first modify operation generates a simple − token (with no event specifier) and then a $modify\Delta^+$. Each subsequent modify operation generates a $modify\Delta^-$, followed by a $modify\Delta^+$.
  Example:

  {t}: assertion that t exists

transition	eventTokens
modify t	(−, then $modify\Delta^+$)
modify t	($modify\Delta^-$, then $modify\Delta^+$)
modify t	($modify\Delta^-$, then $modify\Delta^+$)

- *Case 4*: Tuple t is modified zero or more times and then deleted (m^*d). The net effect is a deletion. Tokens are generated as in Case 3, except that the final delete operations generates a $modify\Delta^-$, which removes the previous $modify\Delta^+$ token, followed by a $delete^-$, which will match any applicable **on delete** rule conditions.
  Example:

  {t}: assertion that t exists

transition	eventTokens
modify t	(−, then $modify\Delta^+$)
modify t	($modify\Delta^-$, then $modify\Delta^+$)
delete t	($modify\Delta^-$, then $delete^-$).

These four cases completely specify how tokens are to be created during any possible sequence of updates to a single tuple, in order to insure that α-memories associated with both event- and pattern-based conditions are updated correctly. The sequence of updates is identified at run time by using the Δ-sets [I,M], providing the information necessary to determine what type of token to create for each operation on a tuple.

4.3.2 Identifying Event and Transition Conditions

If a tuple variable appears in the **on** clause of an Ariel rule condition, then the selection condition defined on that variable is considered to be an *event-based* condition. Similarly, if any tuple variable in the condition has a **previous** keyword in front of it, then the selection condition associated with that variable is a *transition* condition. Both transition and event-based conditions have the property that the data matching them is relevant only during the transition in which the matching occurred. Afterwards, the binding between the matching data and the condition should be broken. This is accomplished in Ariel using α-memory nodes that are *dynamic*, i.e., they only retain their contents during the current transition.

4.3.3 Summary of Token and α-memory Types

In general, for the Ariel rule condition testing system we have identified four kinds of tokens and seven kinds of α-memory nodes. The token types are:

+ token for insertion of a new tuple,

− token for deletion of a tuple,

Δ+ token for insertion of a new transition token (new/old pair),

$\Delta-$ token for deletion of an old transition token.

The α-memory node types include:

stored-α standard memory node holding a collection of tuples matching the associated selection predicate,

virtual-α virtual memory node holding the predicate but not a collection of matching tuples,

dynamic-ON-α a dynamic memory node for an ON-condition which has a temporary tuple collection that is flushed after each database transition,

dynamic-TRANS-α a dynamic memory node for a transition-condition which is also flushed after each transition.

simple-α an alpha memory for a simple selection predicate for a rule with only one tuple variable in its condition. *Simple* memories are only used when the rule has just one tuple variable in its condition. Simple memories never contain a persistent collection of the data matching the conditions associated with them since matching data is passed directly to the P-nodes.

simple-TRANS-α A simple memory node for a transition condition.

simple-ON-α A simple memory node for an event-based (ON) condition.

A different action needs to be taken when each type of token arrives at each type of memory node. The actions for each of the possible combinations are shown in the table in Figure 5. In the table, "$\pi_{new}t$" represents projection of just the new part of the new/old pair contained in t. A "don't care" entry indicates that the combination can never occur, since normal + and − tokens can never match a transition condition.

The information in this chart allows the standard TREAT or Rete algorithm to be generalized to handle normal pattern-based conditions as well as event-based and transition conditions, changing only the behavior of individual components, not the overall structure or information flow. This strategy is one of the keys to successful use of TREAT to support condition testing for the Ariel rule language.

This concludes the discussion of how rule conditions are tested in Ariel. We now turn to the problem of how to execute a rule action once it has been determined that the rule should fire.

5 Optimization and Execution of Rule Actions

At the time an Ariel rule is scheduled for execution, the data matching the rule condition is stored in the P-node for the rule. Binding between the condition and action of an Ariel rule is indicated by using the same tuple variable in both. These tuple variables are called *shared*. To run the action of the rule, a query execution plan for each command in the action is generated by the query optimizer. Shared tuple variables implicitly range over the P-node. When a command in the rule action is executed, actual tuples are bound to the shared tuple variables by including a scan of the P-node in the execution plan for the command. Optimization and execution of Ariel rule actions is discussed in detail below, and illustrated using an example.

```
define rule SalesClerkRule2
if emp.sal > 30000
and emp.jno = job.jno
and job.title ="Clerk"
then do
        append to salaryWatch(emp.all)
        replace emp (sal = 30000)
        where emp.dno = dept.dno
        and dept.name = "Sales"
        replace emp (sal = 25000)
        where emp.dno = dept.dno
        and dept.name ! = "Sales"
end
```

Figure 6: Example rule to illustrate query modification.

5.1 Query modification

When an Ariel rule is first defined, its definition, represented as a syntax tree, is placed in the rule catalog. At the time the rule is *activated*, the discrimination network for the rule is constructed, and the binding between the condition and the action of the rule is made explicit through a process of *query modification* [18], after which the modified definition of the rule is stored in the rule catalog. During query modification, references to tuple variables shared between the rule condition and the rule action are transformed into explicit references to the P-node. Specificly, for a tuple variable V found in both the condition and action, every occurrence of an expression of the form *V.attribute* is replaced by *P.V.attribute*. In addition, if V is the target relation of a **replace** or **delete** command, then it is replaced by $P.V$, and the command is modified to be **replace'** or **delete'** as appropriate. The commands **replace'** and **delete'** behave similarly to the standard **replace** and **delete** commands, except that the tuples to be modified or deleted are located by using tuple identifiers that are part of tuples in the P-node, rather than by performing a scan of the relation to be updated.

For example, consider the rule shown in Figure 6. After query modification is performed on this rule, the commands in its action look as shown in Figure 7, where P is a tuple variable that ranges over the P-node. The tuple variable emp which appears both in the condition and action of the rule has been replaced throughout the action by P.emp in Figure 7. Also, the **replace** and **delete** commands have been transformed into **replace'** and **delete'**, respectively. The tuple variable dept which does not appear in the condition is unchanged in the action.

5.2 Rule action query plan construction

To execute a command in the rule action, an execution plan for that command must be generated, and this plan must include an operator to scan the P-node if any tuple

	type of token t			
α-memory type	+	−	$\Delta+$	$\Delta-$
stored-α	insert t	delete t	insert $\pi_{new}t$	delete $\pi_{new}t$
virtual-α	insert t	delete t	insert $\pi_{new}t$	delete $\pi_{new}t$
dynamic-ON-α	insert t	delete t	insert $\pi_{new}t$	delete $\pi_{new}t$
dynamic-TRANS-α	don't care	don't care	insert t	delete t
simple-α	insert t in P-node	delete t from P-node	insert $\pi_{new}t$ in P-node	delete $\pi_{new}t$ from P-node
simple-TRANS-α	don't care	don't care	insert t in P-node	delete t from P-node
simple-ON-α	insert t in P-node	delete t from P-node	insert $\pi_{new}t$ in P-node	delete $\pi_{new}t$ from P-node

Figure 5: Table showing actions taken by each α-memory type for each token type

```
then do
        append to salaryWatch(P.emp.all)
        replace' P.emp (sal = 30000)
        where P.emp.dno = dept.dno
        and dept.name = "Sales"
        replace' P.emp (sal = 25000)
        where P.emp.dno = dept.dno
        and dept.name ! = "Sales"
end
```

Figure 7: Rule action after query modification.

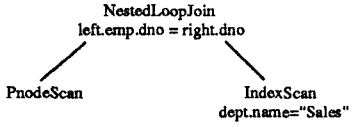

Figure 8: Example execution plan for a command in a rule action.

variables in the command also appear in the rule condition. The Ariel query processor provides an operator called **PnodeScan** which can scan a P-node and optionally apply a selection predicate to it. When the query optimizer sees the special tuple variable P, it always generates a **PnodeScan** to find tuples to be bound to P. The rest of the query plan is constructed as usual by the query optimizer. For example, consider construction of the plan for the following command from the action of the rule SalesClerkRule2:

```
replace' P.emp (sal = 30000)
where P.emp.dno = dept.dno
and dept.name = "Sales"
```

The data to be updated by this command are identified by running a query plan which scans P and dept, and joins tuples from these scans. The tuple identifier of the emp subtuples bound to the variable P is extracted and used to locate the emp tuples to update. One possible query plan that uses a nested loop join, a **PnodeScan** on P, and an index scan on dept, is shown in Figure 8. The query optimizer is free to choose the best operators for other operations in the plan besides the **PnodeScan**, e.g., it could have chosen **SortMergeJoin** instead of **NestedLoopJoin** in Figure 8.

5.3 Time of Rule Plan Construction

The time a rule action plan is constructed can have a substantial impact on performance. Our implementation uses a strategy called **always reoptimize** that produces all plans for execution of rule actions at rule fire time. Other strategies can be developed which attempt to pre-optimize plans for rule actions, store them, and retrieve them at rule fire time to avoid the cost of run-time optimization [8]. All strategies that store plans must maintain the dependencies between those plans and database objects the plans touch such as tables and indexes, which makes those strategies more complicated from the outset. Moreover, pre-planning strategies are all subject to errors where they run non-optimal plans, whereas **always recompute** always runs the optimal plan. A thorough investigation of pre-planning strategies vs. **always recompute** is a potential topic for future investigation.

6 Implementation and Performance

Ariel is implemented using the EXODUS toolkit [2, 14] and in particular the E programming language [15], an extension of C++ with persistent objects. The current version of Ariel consists of about 28000 lines of C++/E code. Persistent objects simplified implementation of our catalogs and the rule index. The object-oriented programming features of C++ simplified and streamlined our design [12].

Below we give some performance figures for installing and activating rules in Ariel, as well as for testing tokens using the discrimination network. Performance was measured on a Sun SPARCstation 1 computer, running at approximately 12 MIPS. Three types of rules were defined such that type 1, 2 and 3 rules have 1, 2 and 3 tuple variables, respectively. We considered rules with different numbers of tuple variables to assess the cost of testing join conditions of rules. Each rule type has a single-relation predicate on the table emp of the form $C_1 <$ emp.sal $\leq C_2$. For each rule type, a set of unique rules was created by starting with one rule as the base rule (rule 0) and generating rule i by adding i times 1000 to C_1 and C_2, for $i = 1$ to the total number of rules. The emp, dept and proj relations contain only a small number of tuples (25, 7 and 5 respectively) in our tests. We would have preferred to use larger relations, but Ariel currently does not support indexes on relations, and we felt that without them, the cost of processing rules would be

no. of rules	installation	activation	token test
25	3.29	11.84	0.0021
50	7.18	24.50	0.0024
100	16.29	79.89	0.0025
200	32.50	97.26	0.0026

Figure 9: Times for one-tuple variable rules in seconds.

no. of rules	installation	activation	token test
25	4.08	21.80	0.0021
50	8.78	44.45	0.0025
100	18.71	90.47	0.0026
200	41.44	196.82	0.0028

Figure 10: Times for two-tuple variable rules in seconds.

dominated by sequential scans of relations and α-memory nodes. Hence, the results would not reflect the potential performance of our discrimination network in the presence of indexes. Fortunately, the performance depends primarily on the structure of the discrimination network, not just the size of the database so our results with small relations will still be useful. Moreover, with large tables and appropriate indexes defined on those tables, performance results similar to the ones reported below are expected. The Ariel architecture is designed to make use of indexes for installing and activating rules and testing tokens through the discrimination network. B-trees for Ariel will be developed using a new B-tree facility being included with the next version of EXODUS.

Figures 9 and 10 show the total time required to install and activate 25 to 200 type 1 and 2 rules, as well as the time to test a token generated by a single insert into emp. Figure 11 shows the same information for 25 to 200 type 3 rules. Rule installation involves storing a persistent copy of the rule syntax tree in the rule catalog, and rule activation involves running one one-variable query for each tuple variable in the rule condition to "prime" the α-memory nodes, plus running a query equivalent to the entire rule condition to load the P-node. These figures show quite reasonable performance for rule installation, which takes a fraction of a second, and rule activation, which takes just under a second. Token testing time takes 2 to 3 milliseconds in our tests, which closely matches earlier predictions [11]. This speed should scale to much larger numbers of rules (given rules of similar structure) because of Ariel's top-level discrimination network for testing selection conditions of rules [11]. Not shown in the figures is that it takes approximately 0.06 seconds to run the action of a type 1, 2 or 3 rule in all cases. We are working to resolve some problems with Ariel and related system software so that a much larger number of rules can be tested. We are encouraged by these results and feel they show the power of a good discrimination network for testing rule conditions. Rule condition testing techniques that do not use some form of discrimination network simply cannot compete when the number of rules becomes large.

no. of rules	installation	activation	token test
25	4.48	27.80	0.0025
50	10.49	58.38	0.0026
100	20.03	112.59	0.0027
200	44.56	228.03	0.0028

Figure 11: Times for three-tuple variable rules in seconds.

7 Relationship to Other Work

There has been a significant amount of research on active databases recently. The main thing that differentiates Ariel from other active database systems is its use of a discrimination network specially designed for testing rule conditions efficiently. Other database rule system projects either:

- do not address the need for efficient data structures for finding which rules match a particular tuple (RPL [4], Starburst rule system [22]),
- do not provide a data structure for testing selection conditions, or
- provide a data structure for testing selection conditions which cannot efficiently handle conditions placed on an arbitrary attribute (e.g., one without an index) (POSTGRES rule system [19, 20, 21], HiPAC [3], DIPS [17], Alert [16]).

Other distinguishing features of Ariel are its close adherence to the production system model, its unified treatment of rules with normal conditions as well as event-based and transition conditions, its ability to run rule action commands without creating any additional joins to the P-node, and its use of a rule-action planner that produces optimal plans for executing rule actions.

8 Conclusions

The Ariel project has shown that a database system can be built with an active rule system that is: (1) based on the production system model, (2) set-oriented, (3) tightly integrated with the DBMS and (4) implemented in efficiently using (a) a specially designed discrimination network, and (b) a rule-action planner that takes advantage of the existing query optimizer. Ariel is unique in its use of a selection-predicate index that can efficiently test point, interval and range predicates of rules on *any* attribute of a relation, regardless of whether indexes to support searching (e.g., B+-trees) exist on the attribute. In addition, the concept of *virtual* α- (and β-) memory nodes introduced in Ariel can save a tremendous amount of storage, yet still allow efficient testing of rules with joins in their conditions. The ability to use virtual memory nodes in a database rule system discrimination network opens up tremendous possibilities for optimization, in which the most worthy memory nodes would be materialized for the best possible performance given the available storage. Prior to the development of the virtual memory node concept, it was mandatory to materialize the α-memory nodes, limiting potential optimizations.

For the future, there are a number of potential research avenues for enhancing active database systems, including:

- support for streamlined development of applications that can receive data from database triggers asynchronously (e.g., safety and integrity alert monitors, stock tickers),
- optimization of the use of storage available throughout the memory hierarchy (memory, disk, tertiary store) for storing memory nodes in a combined Rete/TREAT network augmented with virtual memory nodes,
- support for more efficient rule condition testing and execution in a DBMS using parallelism.

Transformation of databases from passive to active is a landmark in the evolution of DBMS technology. We hope the development of fast, robust active database systems that may come from this research will lead to innovative new applications that make more productive use of the information in the DBMS of the future.

References

[1] M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2), June 1976.

[2] M. Carey, D. DeWitt, D. Frank, G. Graefe, J. Richardson, E. Shekita, and M. Muralikrishna. The architecture of the EXODUS extensible DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, September 1986.

[3] S. Chakravarthy et al. HiPAC: A research project in active, time-constained database management, Final Technical Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[4] L. M. L. Delcambre and J. N. Etheredge. The relational production language: A production language for relational databases. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 153–162, April 1988.

[5] K. P. Eswaran. Specifications, implementations and interactions of a trigger subsystem in an integrated database system. Technical report, IBM Research Laboratory, San Jose, CA, 1976.

[6] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA 15213, July 1981.

[7] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[8] E. N. Hanson. The design and implementation of the Ariel active database rule system. Technical Report WSU-CS-91-06, Wright State University, September 1991.

[9] E. N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proceedings of the 1991 Workshop on Algorithms and Data Structures*. Springer Verlag, August 1991.

[10] E. N. Hanson and M. Chaabouni. The IBS-tree: A data structure for finding all intervals that overlap a point. Technical Report WSU-CS-90-11, Dept. of Computer Science and Eng., Wright State Univ., April 1990.

[11] E. N. Hanson, M. Chaabouni, C. Kim, and Y. Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, May 1990.

[12] E. N. Hanson, T. Harvey, and M. Roth. Experiences in DBMS implementation using an object-oriented persistent programming language and a database toolkit. In *Proceedings of the 1991 ACM Conference on Object-oriented Programming Systems, Languages and Applications*, October 1991.

[13] A. Rastogi. Transition and event condition testing and rule execution in Ariel. Master's thesis, Dept. of Computer Science and Eng., Wright State Univ., June 1991.

[14] J. E. Richardson and M. J. Carey. Programming constructs for database system implementation in EXODUS. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, May 1987.

[15] J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the E programming language. Technical report, University of Wisconsin, 1989.

[16] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. 17th International Conference on Very Large Data Bases*, Barcelona, September 1991.

[17] T. Sellis, C.-C. Lin, and L. Raschid. Data intensive production systems: The DIPS approach. *SIGMOD Record*, September 1989.

[18] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data*, June 1975.

[19] M. Stonebraker, E. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[20] M. Stonebraker, M. Hearst, and S. Potaminos. A commentary on the POSTGRES rules system. *SIGMOD Record*, 18(3), September 1989.

[21] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(7):125–142, March 1990.

[22] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, 1991.