Parallel Rule Processing in a Distributed Object Environment

Technical Report TR99-014

Minsoo Lee, Stanley Y. W. Su and Herman Lam
Database Systems R&D Center
University of Florida, Gainesville, FL 32611 U.S.A.

Abstract *The use of rules is becoming a major trend for adding active capability to passive systems. Although adding the rule processing capability into a system is highly desirable, the effect on performance can be detrimental. To solve the performance problem that is introduced by the overhead of rule processing, parallel processing techniques would be most beneficial. In order to effectively apply these techniques, an easy yet powerful trigger specification method for various rule execution structures is needed, along with the algorithm to schedule their execution. We have developed a trigger specification language and its accompanying GUI tool for specifying events and various parallel rule execution structures associated with distributed objects. Different from the traditional event-condition-action rules used in the existing active databases, the rule model used in this work separates event specifications from rule specifications. Events are dynamically linked to various rule structures by triggers. We have also implemented a rule processor for scheduling and processing the parallel rule execution structures. Parallel processing of triggered rule structures can improve the performance of an active integrated system of distributed objects.*

Keywords: rules, parallel processing, distributed objects

1 Introduction

Recently, rule processing has been used effectively in the areas of active databases, expert systems, workflow systems, and automated control systems as a paradigm to embed active capabilities into passive systems [2][3][7]. Rules enhance a system by using a high-level declarative specification of knowledge which otherwise will have to be written in program code. These rules can automatically be executed when a specific event occurs or some data condition is satisfied. Rules can be used to automatically perform security and integrity constraint checking, alert systems or users of important situations, enforce business policies and regulations, etc.

Most existing active database systems adopt the Event-Condition-Action (ECA) paradigm. Active database research and prototype systems such as HiPAC [2], ODE [3], Sentinel [1] follow this paradigm. An ECA rule consists of three parts: event, condition and action. The semantics of an ECA rule is: when an event occurs, the condition is checked. If the condition evaluates to true, the action is performed. Otherwise, the action is not performed. Condition-Action (CA) rules, which do not have the event specification part, are also widely used in expert systems [5].

Although rules can easily add active capabilities to a system, the additional time it takes to process rules can lead to a serious performance problem. For example, checking if rules can be fired and then actually carry out the rule execution can complicate and lengthen the processing time of a

simple transaction. As a solution to this performance problem, parallel processing of rules is widely used in expert systems [5]. But the parallelism provided by these systems is quite limited. Expert systems mainly focus on parallelizing the condition checking, and users can not easily specify the intended parallel execution of rules using priority-based specifications used by some systems, such as Sentinel [1].

In this paper, we present our work on parallel rule processing in a distributed object environment. In particular, we present the rule specification part of an extended object model, which provides an easy yet powerful way of specifying rule execution structures. An algorithm for scheduling the parallel processing of these rule structures has been developed, based on which a parallel rule processor was implemented for use in a CORBA environment. Rule processing in a distributed object environment can provide many benefits such as supporting inter-operability among distributed objects [6]. Parallel rule processing in a distributed environment can be especially effective since distributed objects are highly independent and promote concurrent processing.

The organization of the remainder of this paper is as follows. Section 2 provides an overview of our rule model. Section 3 explains how various rule execution structures can be specified. In Section 4, the scheduling algorithm and the implementation of the parallel rule processor in the distributed object environment are presented. Finally, Section 5 gives our conclusion.

2 Overview of the Rule Model

Our rule model is designed specifically for supporting parallel rule processing and the interoperation of distributed objects. The key features that differentiate our rule model from other conventional ECA rule models are as follows. First, the event definition and rule definition are separated. Events are defined as objects, and although various ways of raising events are possible, the subscribers of events see a uniform event object which is independent of how it is raised. Rules also do not directly reference event objects. Rather, they are tied to events through intermediate structures called *triggers*. This separation of event and rule definitions provides the flexibility of linking events to rules and fits well with object processing in a distributed environment where distributed objects can independently operate and have an open interface to one another. Second, events can carry any number and any types of parameters and can be delivered in synchronous or asynchronous modes. This makes it possible for a distributed object to define a new type of event without any restriction, whereas many existing systems that support events have a default set of parameters, which is tied to an event when it is defined, and only support the asynchronous mode of delivery. Third, a set of rules that are triggered by an event can be executed following a rule structure specification, which gives the order of rule execution. In conventional ECA rules, a rule is specified by an event, a condition and an action. Some systems allow the specification of a priority value for each rule. If multiple rules contain the same event, the occurrence of the event will fire all these rules in the order of their priorities. This priority-based approach does not support more complex execution rule structures. Furthermore, if rules are recursively fired (i.e., when a set of rules fired by an event recursively fire the same set of rules), using priority values to schedule the processing of rules can create serious semantic problems. Our rule model allows complex rule structures to be explicitly specified by rule designers to control the sequential, parallel, and/or synchronized orders of rule execution. Parallel processing of these rule structures enhances the efficiency of their execution. Fourth, in our model, rules can make reference to distributed objects and call the methods of these objects. This allows rules to be deployed in a more general environment than a single centralized system, and to be used to model the relationships and/or constraints among multiple distributed objects.

Our rule model assumes an underlying object model, which supports the common object-oriented concepts such as classes, objects, attributes/properties, methods and inheritance found in all the existing object models. The rule model can be considered as an add-on to the underlying object model. It consists of three components: events, rules and triggers. An *event* is an occurrence of something of interest to others. It could be things such as an execution of a method of a `Retailer` object, the signaling of the failure of a disk, a reminder of a deadline, etc. An example of a method-associated event `update_product_event` is shown below. The `update_product_event` is associated with the `UpdateProduct` method of the `Retailer` class and has the coupling mode of `before`. This means that this event will be raised *before* the execution of the method. The event will carry the `product_id`, `price`, and `date` parameters given to the `UpdateProduct` method.

```
IN            Retailer
EVENT         update_product_event(String product_id,int price,Date date)
TYPE          method
COUPLING_MODE before
OPERATION     UpdateProduct(String product_id,int price,Date date)
```

A *rule* is a high-level specification of a granule of control and logic, which is traditionally implemented by a piece of executable code. A rule in our system composes of condition, action, and alternative action specifications. When a rule is triggered for processing, the condition of the rule is first evaluated. If the condition is true, the statements in the action specification are executed. Otherwise, the statements in the alternative action part are executed. A rule has an interface, which specifies what parameters are used in the rule. The actual values of these parameters are provided by an event that triggers the rule at run time. An example of a simple rule specification is shown below.

```
RULE        check_price_change(int price,String product_id)
DESCRIPTION if the price is less than $100,
            notify the sales department
RULEVAR     existing SalesDept sales_dept("SALESDEPT1223");
CONDITION   price < 100
ACTION      sales_dept.MakeAdvertisement(product_id,price);
```

The rule checks the price change that occurs, and if the price is less than $100, the sales department is notified to create a sales advertisement. The RULEVAR clause declares the variables that are used in the rule body. The `existing` keyword in RULEVAR indicates that the variable `sales_dept` is to be bound to an *existing* `SalesDept` object in the distributed object environment. The `SalesDept` object should have the ID of SALESDEPT1223.

Now that events and rules have been specified separately, triggers can be specified to connect events with rules. A *trigger* basically specifies which events can trigger which rules or rule structures. It also allows the specification of composite events and maps the parameters between event parameters and rule parameters. An important functionality of the trigger is its capability to specify the structure of a parallel rule execution. As will be shown in later sections, a rule structure carries more semantics than a set of unrelated or prioritized rules. Here, we shall first give a simple example of a trigger that makes use of the example event and rule given above.

```
TRIGGER      update_product_trigger(String p0,int i0)
TRIGGEREVENT update_product_event(p0,i0,d0)
RULESTRUC    check_price_change(i0, p0)
```

The TRIGGER clause specifies the name of the trigger and the trigger parameters. The trigger parameters are used to map the event parameters to the rule parameters. The TRIGGEREVENT clause specifies the events that can trigger the set of rules specified in the RULESTRUC clause. Several events can be OR-ed (i.e., connected by a disjunctive operator), which means that the occurrence of any one of the events can trigger the processing of the rule structure. The RULESTRUC

clause specifies the set of rules to be executed in a specified structure. This simple example contains only one rule that is to be triggered. Complex rule structures will be given later. The parameters of the event are renamed as `p0,i0,d0` and are mapped to the trigger parameters. These trigger parameters `p0,i0` are again mapped to the rule parameters `i0,p0`.

3 Specification of Rule Execution Structures

A general rule structure can be specified by three main constructs: sequential execution, parallel execution, and synchronization.

• *Sequential Execution*: Rules that need to be executed one after another in a certain order can be specified using the sequential execution construct. A sequential execution construct is specified by using a '>' operator. The rule on the left-hand side of the operator is executed first and followed by the rule on the right-hand side. The following example shows a sequence of four rules to be executed in the order of R1, R2, R3, and R4:

```
R1 > R2 > R3 > R4
```

• *Parallel Execution*: Some rules can be executed in parallel to maximize the throughput of a rule processing system and minimize the response time of a transaction execution that triggers rules. To specify the parallel execution of rules, the ',' operator is used. The following example shows that rules R1, R2, R3 and R4 are to be executed in parallel.

```
(R1, R2, R3, R4)
```

• *Synchronization*: Although a majority of rule executions are expected to be either sequential or tree structured, more complex structures that involve synchronization points may also be needed to express the inter-relationships among rules. There are two types of synchronization points: the *AND synchronization point* and the *OR synchronization point*. A rule may not be allowed to start its execution until the completion of several other rules, which are called the predecessors of the rule. A synchronization point where a rule needs to wait for the completion of all of its predecessors is called an AND synchronization point. For example, an AND synchronization point, at which rule R4 will start its execution only after the completion of R1, R2 and R3 is given below:

```
AND (R1, R2, R3) > R4
```

The OR synchronization point needs only the completion of a subset of the predecessors. The size of the subset can be explicitly specified within the brackets beside the OR keyword. Assume that rule R8 has three predecessors, R5, R6, and R7, and rule R8 must wait for two out of the three predecessors to complete before it can start its execution. The specification would be as follows:

```
OR[2] (R5, R6, R7) > R8
```

These AND and OR synchronization points can be nested to specify more complicated execution structures.

• *Complex rule execution structures and rule alias*: In order to express more complex rule execution structures, an entire structure can be broken into pieces. These pieces are separated by ';' and each piece is specified using the concepts of *fan-in* and *fan-out*. Fan-in occurs at a synchronization point (at R8 and R9 shown in Figure 1) and is considered critical for the correctness of a rule execution structure. Fan-out occurs when a rule has multiple descendants, such as rule R3 has descendants R5 and R6. A fan-out may be specified using a single structure (e.g., R3 > (R5, R6)), or by the individual relationships between the predecessor and descendants in separate pieces (i.e., R3 > R5; R3 > R6). The complex rule structure in Figure 1 can be specified as :

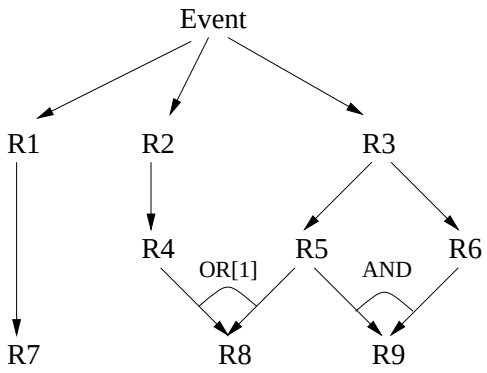

Figure 1: An example complex rule execution structure

```
R1 > R7 ;
R2 > R4 ;
R3 > (R5, R6) ;
OR[1]( R4, R5 ) > R8 ;
AND ( R5, R6 ) > R9
```

The sequential sequences such as R1 > R7 and R2 > R4 are specified as before. A fan-out is used to specify that, after the execution of R3, rules R5 and R6 can start their execution independently. Two fan-in sequences are also specified. After R4 *or* R5 finishes its execution, R8 can then start; and after R5 *and* R6 finish their execution, R9 can then start its execution. Any kind of complex rule structure can be decomposed using this fan-in and fan-out mechanism. We note here that a rule may participate in many rule structures, which are triggered by many different events. The execution of a rule may post an event(s), which in turn triggers other rule structures. Changing trigger specifications at run-time can change the behavior of a distributed object system.

A rule may appear more than once in a single rule structure. For example, an integrity rule can be executed before a set of rules and then again after the set of rules to ensure that rule triggering does not compromise the data integrity. In this case, a rule alias mechanism is needed to differentiate between these different occurrences of the same rule.

3.1 Event Parameter Mapping to Multiple Rules

An event which triggers a rule structure should be able to pass its parameters to its rules. This requires a mapping between the event parameters and the rule parameters. This mapping is given in the trigger specification. A trigger has a set of parameters that are type compatible with both the event parameters and the rule parameters. The triggering events can pass their event parameters through the trigger parameters to the rule parameters. An example mapping is shown below.

```
TRIGGER      sample_trigger(int v1,int v2,classX v3)
TRIGGEREVENT E1(v1,v2,t1,v3) or E2(v2,v3,v1)
RULESTRUC    R1(v1,v2)>R2(v3,v1)>R3(v1,v2)
```

The example shows a trigger with three parameters v1, v2, and v3. The types of the parameters are integer, integer and classX, respectively. E1 and E2 are the two triggering events. The event parameters are mapped to the trigger parameters by sharing the same name for the parameters. For instance, the parameter v1 of E1 is mapped to the parameter v1 of the trigger. Note that parameter t1 of event E1 is not mapped to any trigger parameter. The parameter mapping from

the trigger to the rules is done once again in the same way by sharing the names between the trigger parameters and the rule parameters. R1 uses the parameters v1 and v2. R2 uses the parameters v3 and v1, etc. In our present implementation, the parameters are passed from events to rules by value.

4 Implementation of the Parallel Rule Processor

Although implementing the rule processor in C or C++ may have a better performance, we consider platform independence as one of our major goals to provide rule processing in the distributed object environment and selected Java as the implementation language. Because using an interpreted language such as Java increases the overhead of rule processing, supporting parallel rule processing becomes even more important.

• *Parallel Processing*: Multi-threading with Java threads is used to implement the parallel execution of rules. When a trigger is fired, a thread is spawned for each rule in the trigger and the rules are executed in parallel via the concurrent execution of threads. When executing rules on a single system, the performance gain is limited by the number of physical processors available. Thus, depending on the workload, several configurations of the rule processor are possible. A single rule processor can be running on a single multiprocessor system or several rule processors, each of which can be running on a machine where each rule processor is dedicated to the processing of a subset of the triggers, or additional slave processors (i.e., rule executors) can be connected to each of the rule processors to perform only rule executions that are requested by the rule processor.

The synchronization among threads is done by a scheduling algorithm, which uses a *queue* having two Java synchronized methods: `put` and `get`. The `put` method puts the rule index (which is used to identify a rule within a trigger) and the results of a rule that finished its execution into the queue. Then it calls the `notifyAll` method to wake up all the threads that are waiting for an item in the queue. The `get` method retrieves an object from the queue that contains information about a completed rule. If the queue is empty at this time, the `get` method calls the `waitfor` method and sleeps until it is waked up.

• *Data Structure*: The scheduling algorithm uses data structures generated from trigger specifications, which are input through a trigger editor. For each rule, the data structures store the number of predecessors that each rule needs to wait for before it can start its execution. Also, for each rule, a list of the succeeding rules (i.e., the rules that are to be executed directly after the rule in the sequence) are stored. Other data structures that are generated by trigger specifications involve data structures that hold the parameter mapping information.

Once these data structures are generated at the trigger definition time, the scheduling algorithm interprets the data structures at run-time and performs the necessary operations to schedule the rule execution.

• *Scheduling Algorithm* : When a triggering event occurs, it is forwarded to a rule processor, which can trigger a rule structure. The high-level description of the scheduling algorithm is shown in Figure 2 and is described below. The scheduling algorithm used in our system is based on the idea that, when the predecessors of a rule S complete their execution, rule S can start its execution in parallel with other rules. Therefore, whenever a rule is finished, its successor rules (Ss) are checked one by one to see if its (other) predecessors have finished their execution. If so, the successor rule S is executed, otherwise it must wait for the completion of its predecessors.

• *Run-time Architecture* : The run-time architecture for parallel rule processing in a distributed object environment is shown in Figure 3. The Rule Processor is a CORBA server implemented in an Orbix environment [8]. It is registered as a consumer of the CORBA Event Service, subscribing

```
Start all rules with no predecessors;
While (all rules in the rule structure are not finished)
{
   Wait for any rule to finish and put its rule index
        into the queue;
   Denote finished rule as ruleA;
   For each successor (denoted as ruleS) of ruleA do,
   {
      Increase terminated_predecessor_count of ruleS;
      Check the predecessor condition of ruleS
           to see if ruleS can execute;
      If so, start ruleS with appropriate parameters
           and execute in parallel;
    }
}
```

Figure 2: Scheduling algorithm for the execution of a rule structure

to events of interest that can trigger rules. A distributed CORBA object can post an event, which is transferred through the event service to the appropriate subscribers of the event. The rule processor, as a subscriber of those events that have been defined in trigger specifications, will be notified of the occurrence of an event. Once notified, the rule processor looks up its corresponding triggers and starts executing each trigger. The rules specified in the trigger are executed in parallel. During execution, a rule can call the methods of other distributed objects. Thus, several rules may call the methods of different distributed objects in parallel.

5 Conclusion

We have provided a specification language that allows rule structures of different complexity to be specified. The scheduling algorithm and the implementation of the rule processor as an Orbix server has also been presented. The notable features of the implemented rule processor, including those not mentioned above, can be summarized as follows. First, the rule processor can schedule the execution of complex rule structures that are composed of sequential, parallel, AND synchronization, and OR synchronization constructs. Second, rules can activate distributed objects and thus perform distributed actions. Third, rules are compiled into Java code, which provides platform independence. Fourth, rules can be dynamically changed and reloaded during run-time of the system, which makes the system more flexible. Fifth, grouping of rules is supported to allow the enabling/disabling groups of rules. Sixth, rules can return data values to the method call that posts an event. Lastly, an accompanying GUI Editor and a Metadata Manager have been developed to assist in specifying, editing, and storing events, triggers, and rules. The implemented rule processor clearly demonstrates its capability to add active features into a distributed object environment, and the efficiency that can be gained by parallel rule processing. Our planned future work includes the expansion of parallel rule processing into other distributed object environments, such as RMI and enterprise JavaBeans. Also, additional transaction-related features will be included.

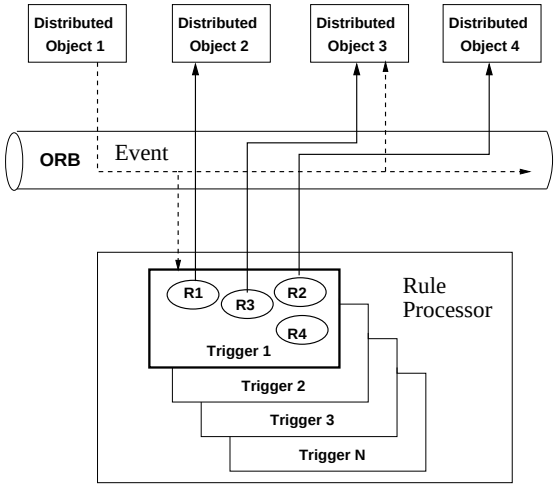

Figure 3: Framework for Parallel Rule Processing in the Distributed Object Environment

References

[1] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra. Design of Sentinel: an Object-Oriented DBMS with Event-Based Rules. *Information and Software Technology*, 39(9):555–568, London, Sept. 1994.

[2] U. Dayal, et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM Sigmod Record*, 17(1):51–70, March 1988.

[3] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. *Proc. 17th Int'l Conf. on Very Large Data Bases*, pp. 327-336, Barcelona, Spain, Sept. 1991.

[4] J. Kiernan, C. de Maindreville, and E. Simon. Making Deductive Databases a Practical Technology: A Step Forward. *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pp. 237-246, Atlantic City, NJ, May 1990.

[5] J.N. Amaral and J. Ghosh. A Concurrent Architecture for Serializable Production Systems. *IEEE Trans. on Parallel and Distributed Processing*, 7(12):1265–1280, Dec., 1996.

[6] S.Y.W. Su, H. Lam, et. al. An Extensible Knowledge Base Management. System for Supporting Rule-based Interoperability among Heterogeneous Systems. *Proc. of the Conference on Information and Knowledge Management*, pp. 1-10, Baltimore, MD, Nov. 28 - Dec. 2, 1995.

[7] S.Y.W. Su, R. Jawadi, P. Cherukuri, Q. Li, and R. Nartey. OSAM*.KBMS/P: A Parallel, Active, Object-oriented Knowledge Base Server. *IEEE Trans. on Knowledge and Data Engineering*, 10(1):55–75, Jan./Feb. 1998.

[8] *OrbixWeb Programmer's Reference.* IONA Technologies PLC, Dublin, Ireland, 1997.