Incorporating Flexible and Expressive Rule Control in a Graph-Based Transaction Framework

Ramamohanrao S. Jawadi
Informix Software Inc.
Portland, OR 97205
Email: jawadi@informix.com

Stanley Y.W. Su
Database Systems Research and Development Center
University of Florida, Gainesville, FL 32611
Email: su@cis.ufl.edu

Abstract

The need for user-defined execution orders (or control structures) for rules is well recognized by researchers of active database management systems. Priority-based approaches (e.g., numeric priorities) have been used to specify a desired control structure among rules. However, due to the fact that fixed priorities are assigned to rules, independent of different contexts in which they may be triggered, the existing approaches are not able to allow rules to be executed following different control structures when they are triggered by different events. More flexible and expressive control mechanisms are often needed for rules in advanced database applications. Since rules in database environments are executed in a transaction framework, an expressive transaction model is needed to model complex control structures among rules uniformly. In this work, we separate the event part from the condition-action parts of a rule and associate it with a rule graph which represents a set of rules (actually a set of condition-action pairs) sharing the same control structure. Different rule graphs can be defined under different event specifications thereby enabling a set of rules to follow different control structures when triggered by different events. We also use an expressive graph-based transaction model to incorporate the control structures of rule graphs uniformly in a transaction framework. The proposed rule and transaction modeling and execution techniques have been implemented and verified on a shared-nothing multiprocessor computer nCUBE2 which exploits the parallel execution properties of independent rules (tasks) in a rule graph (transaction graph).

1 Introduction

Database management systems (DBMSs) coupled with Event-Condition-Action (ECA) rules [9], which are known as active DBMSs, are becoming increasingly popular because of their added features for supporting a wide spectrum of applications [13, 17, 18, 12, 20, 3, 16]. In contrast to the passive DBMSs, active systems monitor a variety of events (e.g., external events, user-defined operations, DB operations) and react to them automatically by triggering and processing the condition and the action parts of ECA rules. In many advanced DB application domains such as CAD/CAM, CASE, CIM and Flexible Manufacturing Systems (FMS), multiple events may trigger the processing of a set of rules and these rules need to follow different execution orders (or control structures) for different events. For example, in the diagnosis and test stage of car manufacturing, the event “Engine_overheating”, may require the steps defined by rules r1-r6 be followed in the order shown in Figure 1.a which is different from one of many possible control structures (e.g., Figure 1.b) for the event, say, "General_diagnosis".

The simple example above serves to stress several important points which motivate our work. First, an event may trigger not just one rule (one condition and action pair) but a number of rules which represent a number of necessary testings and corrective actions. These rules have to follow a specific control structure during their execution. This should not be surprising to any one because any complex application program is full of procedural implementations of structured testings and operations. The advantages of capturing the procedural semantics buried in application programs by rules are well-understood by the database community (e.g. semantic query optimization, knowledge base validation, logical deduction, etc.). They need not be justified here. Second, different events may trigger the same set of rules but require the rules to follow different control structures during their execution due to different event semantics (e.g., Engine_overheating vs. General_diagnosis). In those existing active DBMSs in which fixed priorities are assigned to ECA rules (e.g. POSTGRES [17], HiPAC [9], Ariel [12], Alert [1] and Starburst [20]), dif-
Figure 1: Rules for Diagnosing an Engine

The remainder of this paper is organized as follows. In Section 2, we first describe rule graphs and show how they can provide expressive and flexible rule control. In Section 3, we describe the graph-based transaction model and illustrate how it can naturally incorporate rule graphs. In Section 4, we describe the architecture and parallel implementation of an active DBMS server which supports the proposed rule and transaction models. In Section 5, we analyze and evaluate the system's performance. Section 6 relates our work to some existing work and a conclusion is given in Section 7.

2 Graph-based Rule Control

In existing active DBMSs, control structure for multiple rules triggered by a single event is specified using rule priorities [17, 12, 1, 6] and these rules follow the same control structure (which is derived from rule priorities) during their execution, independent of the event that triggered them. This is because event and priority are tied together with C and A parts of a rule. More flexible control structures among CA rules can be specified if the specification of rule control and triggering events is separated from the definition of the CA rule. We introduce rule graphs to specify control structures among CA rules in a more flexible fashion than with fixed priorities.

A rule graph is defined by a set of CA rules (condition-action pairs or simply rules) having a graph-based control structure among them. A rule may participate in multiple rule graphs and may have different execution orders relative to other rules in the graphs. In addition, using rule graphs, new rules with desired control requirements can be inserted, and independent rules can be explicitly specified. We also adopt the graph-based transaction model used in DOM [4], ACTA [7] and ConTract [19] to model the control structure among operations which define a transaction. The expressive power of the graph-based model allows rule graphs to be incorporated in the transaction framework in a natural and uniform fashion. We shall call a transaction with a graph-structure a "transaction graph". In this work, we also exploit the parallel execution properties of independent rules (DB operations) in a rule (transaction) graph. We define correctness criterion for executing DB operations and rules concurrently in the graph-based transaction framework. The proposed transaction and rule modeling and execution techniques have been implemented on nCUBE2 - a parallel computer, for the purpose of verifying their implementability and studying their performance improvement.
A rule graph is always associated with some triggering event(s). For example, Figure 1.b can be viewed as a rule graph associated with an event "General_diagnosis". The semantics of the rule graph are that it is triggered by the event "General_diagnosis", and once it is triggered, rule execution has to follow the control structure depicted by the DAG, i.e., rule r5 is executed first and is followed by r6, then by rules r2 and r3 in parallel, and so on. It can be observed that the graph representation of rule control provides enough flexibility to allow a set of rules to follow different control structures when they are triggered by different events. This can be done by defining different rule graphs having the same set of rules with different control structures, and associating them with different events. For example, using rules r1-r6, two different rule graphs RG1 and RG2, having the control structure depicted in Figures 1.a and 1.b respectively, can be specified. RG1 and RG2 are associated with events Engine_overheating and General_diagnosis respectively so that they follow appropriate control structure when triggered.

The rule graph representation is also more expressive than the rule priority specification because its general graph-structure captures more complex structural relationships among CA rules. It also allows parallel rules to be explicitly specified. We have provided different control constructs namely precede, precede-set and follow-set, to specify the graph-based control structure in a user-friendly fashion. For detailed syntax and semantics of these constructs refer to [14]. Additionally in rule graphs, new rules with desired control requirements can be added easily. For example, consider the following ECA rules (with the numeric priorities in parentheses) R1(1), R2(2) and R3(3) each of which has E1 as its event. They can be represented by a rule graph: (E1 : r1→r2→r3) where r1, r2 and r3 represent condition-action pairs of R1, R2 and R3 respectively. ECA rules R7(7), R8(8) and R10(13) each of which has E2 as its event, can be represented by the rule graph: (E2 : r7→r8→r10). Consider the case where a new rule R20 with "E1 or E2" as its event needs to be added to the rule base and its control requirements are that it has to be executed before R2 when it is triggered by E1 and after R7 when it is triggered by E2. Note that it is not possible to satisfy the above control requirements using numeric priorities. However, with rule graphs, the above control requirements can be satisfied by adding r20 which represents the condition-action pair of R20 in the rule base of the appropriate class and adding the name r20 to the above rule graphs as given below:

E1: r1→r20→r2→r3;  E2: r7→r20→r8→r10

It must be observed that the specification of control structures among rules is clearly separated from the specification of rules themselves. This provides an opportunity for a rule designer to clearly demarcate the control behavior of rules from their individual definitions.

Trigger Times: Each rule graph is associated with a trigger operation(s) and a trigger time. The trigger time specifies when a triggered rule graph should be executed relative to the trigger operation (or triggering event). The following trigger times can be specified in our rule specification language: i) before: the associated rule graph is executed before the trigger operation, ii) immediately-after: the associated rule graph is executed immediately after the trigger operation, iii) after: the associated rule graph is executed just before the commit time, and iv) parallel: the associated rule graph is executed as a separate transaction graph.

The above trigger times provide the rule designer with some flexibility in specifying different execution options for a rule graph. However, an expressive transaction execution model is needed to uniformly incorporate the execution of complex rule graphs with different trigger times in a transaction framework.

3 Transaction Graph Model

We introduce a transaction graph model (TGM) which has a more expressive control structure to model and execute rule graphs in a natural and uniform fashion. In addition, control points corresponding to different trigger times can be easily identified in the structure of a transaction graph. In TGM, a transaction is viewed as a control graph of DB operations (or tasks) as shown in Figure 2.a. In this Figure, t1-t5 are different DB tasks. Boxes represent the spheres of control and the directed edges represent the control structure among the tasks. Each task in turn can have a graph structure.
3.1 Expressiveness of a Transaction Graph

Since a rule graph and a transaction graph (TG) have the same structure, rule graphs can be uniformly incorporated into the transaction graph’s control structure. For example, if a transaction T triggers the rule graph shown in Figure 3.a, the rules r1–r5 are treated as subtransactions (or tasks) of T, as shown in Figure 3.b. The tree grows further as the rules trigger more rules.

Another important feature of the TG is its ability to support the control semantics of different trigger times. During the execution time, triggered rule graphs are included in the dynamically expanding control structure of a TG according to their trigger times. In the case of before or immediately-after trigger times, rules of the triggered rule graph are treated as subtransactions of the triggering task. Since our model captures the control structure among subtransactions, the control structure of the rule graph can be easily incorporated, e.g., if task t2 of the TG shown in Figure 2.b triggers the rule graph in Figure 4.c, it is added to the TG as shown in Figure 4.a. In the case of after as the trigger time, the triggered rule graph is added to the triggering TG just before the commit task and the rules are treated as subtransactions of the TG as shown in Figure 4.b. In the case of parallel as the trigger time, the triggered rule graph is executed as a separate TG in parallel with the triggering TG. It is completely detached from the triggering TG in all respects except a causal relationship. A parallel transaction can be causally dependent or causally independent of the triggering TG. In the first case, the failure of the parallel TG causes the triggering TG to abort and, in the second case, the failure does not affect the triggering TG. However, the failure of a triggering TG causes all triggered TGs including parallel TGs to be aborted.

It can be observed that, for all trigger times, the control structure of rule graphs uniformly fits into the proposed transaction control structure. This obviates a transaction manager’s need to treat rules differently from other subtransactions with respect to the scheduling, concurrency control and the correctness criterion.

3.2 Concurrency Control

In a TG model, DB tasks and rules maintain atomicity and isolation. In addition, they also need to maintain the partial order defined by their rule or transaction graphs. Rule (task) serializability alone (any serializable execution of rules (tasks)) is not a sufficient correctness criterion for rules (tasks) in a rule (transaction) graph. We now define the correctness criterion for the concurrent execution of rules and tasks within a TG and discuss two scheduling algorithms and a locking scheme which are useful for maintaining the correctness.

3.2.1 Correctness Criterion

At the transaction-level, the correctness criterion for concurrent execution of several TGs is the standard serializability [10] which states that the interleaved (concurrent) execution of several TGs is correct when it is equivalent to some serial execution. Within a TG, the control structure among tasks and triggered rules plays a role in defining correctness. Serializability alone cannot be adopted as a correctness criterion for rules and tasks because an arbitrary serial order can violate the control structure (e.g., for the rule graph shown in Figure 3.a, the serial order r1–r4–r2–r3–r5 is not correct). The concurrent execution of rules (tasks) in a rule (transaction) graph is correct if and only if the resulting serial order is equivalent to some topological order [8] of execution.

To summarize, when rules in a rule graph are being executed concurrently, the correctness criterion includes the following conditions: (1) the execution of a rule must be atomic, (2) the execution of a rule must be isolated from the execution of all other rules that are active in the system, (3) the concurrent execution of rules in a rule graph must be equivalent to a sequential execution of these rules in a specified topological order. We will call the above criterion topological serializability. The same correctness criterion applies for tasks in a transaction graph.
3.2.2 Topological Scheduling

A set of rules in a rule graph can be executed in parallel if they are independent of each other. A rule in a rule graph is independent if it does not have any incoming edge. To maintain a topological order, rules in a rule graph can be divided into topological groups so that the rules in each group are independent of each other and can be executed in parallel. The groups can be formed as follows. All the rules in a rule graph can be given a level number in such a way that every rule's level is lower than the levels of all of its successors. A rule r2 is a successor to rule r1 if there is a directed path from r1 to r2 in the rule graph, and r1 becomes r2's predecessor1. Rules can be partitioned into topological groups by clustering them according to their level numbers. For example, the groups for the rule graph shown in Figure 3.a are (r1:1), (r2:2, r3:2), (r4:3, r5:3). The number beside a rule indicates its level in the rule graph. Within each group, the rules can be executed concurrently, however the serial order of the topological groups must be maintained. The following algorithm schedules the rules of a rule graph in a topological order and exploits the parallelism among independent rules in each topological group.

Repeat

Select: This step selects the set of independent rules which is unique for a given rule graph (e.g., in the first iteration, all the rules at level 1 are selected.).

Schedule: This step assigns the independent rules to appropriate processing nodes for parallel execution2. The isolation among rules is maintained by a locking scheme which is explained in Section 3.2.4.

Wait: This step synchronizes all the rules scheduled in one iteration by waiting for their completion. In the actual implementation, which is explained later in Section 4, the transaction manager switches to another transaction graph during the wait step, thereby interleaving the execution of multiple transaction graphs.

Remove: This step removes all the completed rules as well as the outgoing edges of those rules from the rule graph, so that schedule gets the next set of independent rules in the succeeding iteration.

Until the rule graph is completely executed

3.2.3 Asynchronous Scheduling

The above scheduling algorithm waits (in the wait step) for all the scheduled rules to complete before scheduling the next set of independent rules. This can make some rules wait unnecessarily. For example, consider the rule graph shown in Figure 3.a. It is scheduled in the following sequence of groups \{r1→(r2, r3)→(r4, r5)\} and note that r4 waits until the completion of r2 and r3 despite the fact that it is eligible for execution as soon as r2 is completed.

We avoid the unnecessary waiting of the rules by scheduling them asynchronously. We modify the algorithm explained in the previous section to monitor all the scheduled rules for their completion so that, if any of them completes its execution, all the rules that become independent due to the completion can be scheduled for execution immediately. In the modified algorithm, the completed rules place their acknowledgments in a queue asynchronously. The scheduler monitors the queue for the acknowledgment from at least one of the scheduled rules and it waits only when the queue is empty, otherwise it proceeds to the remove step. In the remove step, all the rules corresponding to the acknowledgments in the queue and their outgoing edges are removed from the rule graph as are the acknowledgments from the queue. This approach avoids the unnecessary waiting of some rules and thereby increases concurrency.

3.2.4 Locking Scheme

Although the above two scheduling algorithms enforce the control structure, they do not ensure the isolation of concurrently executing rules, e.g., parallel execution of two independent rules need not be serializable. The following locking rules are introduced for maintaining the atomicity and isolation of rules as well as transaction graphs.

1. A transaction/rule may hold a lock in write mode (read mode) if all other transactions/rules holding the lock in any mode (in write mode) are ancestors of the requesting transaction/rule. Note that, for a rule, the triggering transaction/rule becomes the parent and an ancestor is any rule above in the triggering line of hierarchy.

2. When a transaction/rule aborts, all its locks, both read and write, are released. If any of its ancestors hold the same lock, they continue to do so in the same mode as that of the committed child.

3. When a transaction/rule commits, all its locks, both read and write, are inherited by its parent (if any). This means that the parent holds each of the locks in the same mode as that of the committed child.

Note that the above scheduling algorithms and locking rules which are defined for rules in a rule graph, can also be used for tasks in a transaction graph, because a rule graph and a transaction graph have the

1 Note that this is different from ancestor-descendant relationship.
2 On a uniprocessor machine, the rules can be executed as child processes or threads.
same control structure. The theorems that prove the correctness of the above scheduling algorithms and locking rules are not given here because of the space limitation and the reader is referred to [14] for the theorems. The recovery of a transaction graph is out of the scope of this paper.

4 Parallel Implementation

The proposed rule and transaction execution models have been implemented on an nCUBE2 computer as part of the implementation of a parallel active object-oriented knowledge base management system named OSAM*KBMS/P [14]. The objective of this implementation is to test the feasibility of implementation of the proposed models, and demonstrate that the parallel execution property of graph structures can be exploited to achieve efficient transaction and rule executions. The overall architecture of OSAM*KBMS/P, as shown in Figure 5, is based on the standard client/server architecture explained in [15]. The clients C1, C2, ..., Cn, each of which is running on a workstation connected to the server by an interconnection network (Ethernet), each have an X-Motif graphical user interface (GUI) for editing and browsing TGs, rule graphs, queries and DB schemas. A client translates a TG into an intermediate form and sends it to the server through the interconnection network. For efficient processing of transaction and rule graphs, we have implemented the server on a 64-node nCUBE2 computer. The server's architecture is designed to be scalable and asynchronous. As shown in Figure 5, the server has a global transaction manager (GTM) and a group of local transaction managers (LTMs). All computation intensive functionalities, namely, data processing, lock management, recovery management and rule processing are assigned to the LTM so that the system becomes scalable. The server processes transactions asynchronously. All the modules in the server (e.g., transaction scheduler, task processor) operate asynchronously and they interact with each other using messages to cooperatively process transactions.

In our system, rule processors (RPros) are dynamically launched when there are rules to be executed and they are terminated automatically as soon as the assigned rules have been executed. The scheduler identifies independent rules in the rule graph and depending on the number of independent rules, dynamically launches the same number of RPros to process them in parallel. This is implemented using the nCUBE's remote launching facility which allows new processes at remote nodes to be launched at run time. Each RProc is responsible for processing the assigned rule.

Figure 5: System Architecture of OSAM*KBMS/P

Figure 6: Typical Transaction and Rule Graphs

If an operation generated by a rule in turn triggers a rule graph, the rule graph is passed to the GTM for scheduling at an appropriate time. An RProc is automatically terminated when the assigned rule completes its execution. The technique of launching RPros only when there are rules to be processed reduces the load on the system significantly. It also provides the scheduler with the flexibility to dynamically choose an ideal node for processing a triggered rule.

5 Performance Evaluation

For performance evaluation, we used a Registration transaction which has the graph structure shown in Figure 6.a, and a rule graph shown in Figure 6.b. The latter is triggered by the task Calculate Debt.

We ran 640 Registration transactions on the server with one global unit or GU (a GTM launched on a cube of 4 nodes) and one local unit or LU (an LTM launched on a cube of 4 nodes). Later, by keeping the number of transactions and GUs constant, we gradually increased the number of LUs to 15 and noted
the speedups due to the parallel execution properties of the rule and transaction graphs as well as the asynchronous execution model. The speedup is defined as \( \frac{t_4}{t_1} \) where \( t_1 \) is the time taken by 640 transactions in a system with one GU and one LU, and \( t_4 \) is the time taken by 640 transactions in a system with one GU and \( n \) LUs. It was observed that the increase in speedup was linear up to 10 LUs and it gradually tapered as the number of LUs reaches 15. The gradual reduction in the speedup is due to the communication time for distributing tasks and rules to different LTMs and the time for converting each of them into a message format suitable for nCUBE2's interprocessor communication.

We also evaluated the scaleup of the system by running 40 Registration transactions on the system with one GU and one LU, then increased the number of transactions and the number of LUs in the same proportion (e.g., 80 transactions on (1 GU and 2 LUs), 120 transactions on (1 GU and 3 LUs), etc.). The formula used for the scaleup is \( \frac{t_9}{t_8} \) where \( t_9 \) is the time taken for processing \( 1^*40 \) transactions on one LU, and \( t_8 \) is the time taken for processing \( n^*40 \) transactions on \( n \) LUs. The system scaled up well because most of the computation intensive functionalities are distributed to LTMs instead of being centralized at a GTM. The scaleup was almost linear up to '520 transactions on (1 GU and 13 LUs)'. At this point, the GTM becomes a bottleneck serving multiple LTMs at one side and several clients at another side. The scaleup therefore gradually tapered as the number of transactions and the number of local units increased further. The scaleup can be further improved by launching multiple GTMs to receive incoming transactions in parallel and to schedule them to the underlying LTMs.

6 Comparison with Related Work

The rule subsystems of active DBMSs. POSTGRES [17], Ariel [12], HiPAC [9], Starburst [20] and Alert [1] support priorities to define complex control structures among rules. However, priority-based rule control is not flexible and expressive enough to support the control requirements of rules in advanced DB applications. Simon, Kiernan and deMaindreville [16] separate the rule control from rules by defining them in two different sections of a module, and propose a set of expressive control constructs namely 'sequence', 'disjunction' and 'saturate', for specifying rule control for the active DBMS RDL1. However, for a set of rules defined in a module to follow N different control structures when they are triggered by different events, the same rules must be defined N times in N different modules. Similar to RDL1, we also define rule control and rules in two different sections. In addition, we allow different control specifications for the same set of rules according to the semantics of triggering events. Since rule graph (or control) specification contains only rule names, rules need not be defined repeatedly if they participate in a number of rule graphs.

In an active DB environment, the rule execution must be uniformly incorporated into a transaction framework. In POSTGRES and Ariel, the execution of rules has been incorporated into a flat transaction model. Some other research efforts have [13, 3, 5] described more expressive models which are basically extended nested transaction models (NTMs) to capture rules and nested triggerings of rules uniformly. However, the tree-based structure of NTM is also not expressive enough to uniformly model graph-based control structures among rules. The transaction graph model (TGM) not only incorporates the triggered rule graphs uniformly but also places them at an appropriate control point of the transaction structure according to their trigger times. NTM is a special case of TGM, i.e., the case when triggered rules are not connected by a control structure. Several expressive transaction models worth mentioning have been reported in [7, 19, 4, 2] However, they do not focus on the uniform incorporation of rules, rule control and trigger times in the transaction framework.

7 Conclusions

The majority of contemporary active DBMSs adopt priority-based approaches for specifying and enforcing control structures among rules. Priority-based approaches lack the flexibility and expressiveness needed for many advanced applications. In this paper, we have provided a more powerful rule control mechanism which enables rules to follow different control structures when they are triggered by different events and allows new rules with desired control requirements to be added easily. Using rule graphs, parallelism among rules can be explicitly specified. To execute rule graphs in a transaction framework, we have used a graph-based transaction model and showed that its dynamically expanding structure can uniformly model dynamically triggered rule graphs at different trigger times. The presented graph-based transaction and rule execution models have been implemented on an nCUBE2 parallel computer using a client/server architecture. Some implementation techniques and the results of a performance evaluation have been described to demonstrate the speedup and scaleup of the implemented system. It should be noted that the structure of a transaction graph models all the trigger times, and is independent of the complexity of an event specification. Therefore, it can be readily adapted for the
execution of rules with complex events [11]. Furthermore, it is general enough to have rules executed at arbitrary points of a transaction’s life time as suggested in [5].

Acknowledgments:
The authors would like to thank Dr. Eric Hanson for the detailed comments on the previous version of this paper and Dr. Sharma Chakravarthy for several useful discussions on the graph-based transaction model. Thanks to Kurt Engel for his proof reading.

References

3That is before or after any task in the transaction.