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

Ramamohanrao Jawadi *Stanley Y.W. Su*
Technical Report TR94-030
Database Systems Research and Development Center
Department of Computer and Information Sciences
University of Florida, Gainesville, FL 32611
rsj,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 such as CAD/CAM, CASE, CIM and flexible manufacturing systems. 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 [11], which are known as *active DBMSs*, are becoming increasingly popular because of their added features for supporting a wide spectrum of applications [18, 33, 34, 17, 36, 3, 12, 31]. 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.

An ECA rule consists of an event (E), a condition to be checked (C), and an action (A) to be executed if the condition is true. In this paper, we separate the event part from the condition

and action parts of the ECA rule. We use "event" and "trigger operation" interchangeably and use "CA rule" (or simply rule) to refer to the C and A parts of the ECA rule. We observe that, 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 rule 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".

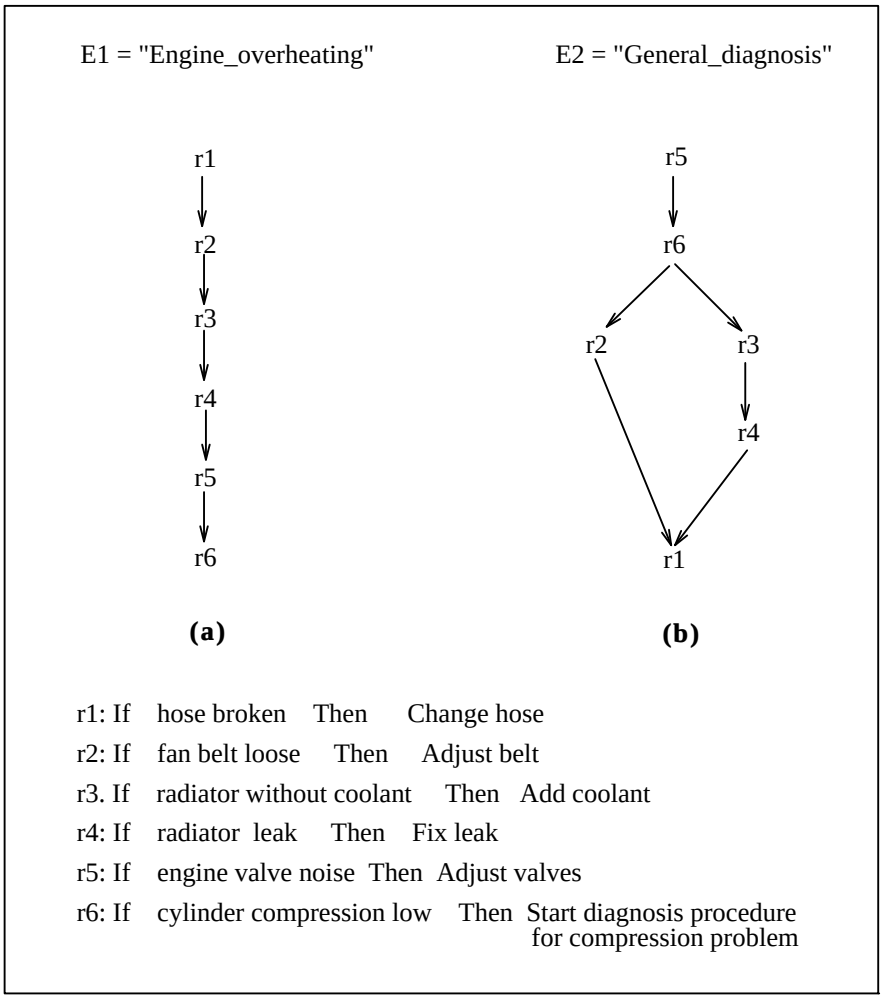

Figure 1: Rules for Diagnosing an Engine

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 [33], HiPAC [11], Ariel [17], Alert [1] and Starburst [36]), different control structures for a single set of rules cannot be accommodated. Furthermore, ECA rules with fixed priorities (e.g., numeric priorities) cannot always accommodate a new rule with the desired priority. For example, consider the following ECA rules with their numeric priorities given in parentheses: R1(1), R2(2), R3(3), R7(7), R8(8) and R10(13). Assume that R1, R2 and R3 have a common event E1 and R7, R8 and R10 also have a common event E2. Consider the case where a new rule R20 has to be inserted into the rule base. Its event is "E1 or E2" which means that it is triggered by the occurrence of either E1 or E2. The control requirements of R20 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 there is no numeric priority (which is less than 2 and greater than 7) that can satisfy the above control requirements. Third, the control structure associated with rule execution can be quite complex (not just a linear or tree structure but a more complex acyclic graph). Since rules can be triggered by the operations in a DB transaction and rules may in turn generate DB operations, the execution of rules need to be incorporated into a transaction execution framework in a uniform fashion. Although, the execution of rules having complex control structures can be supported by extended implementations of flat [16] or nested transaction models [24], these models are not expressive enough to model graph-based control structures among operations in a transaction and structured rules uniformly. To uniformly capture graph-based control structure among transaction operations and rules, a more expressive transaction model (e.g., a graph-based model) would be needed. Lastly, some rules in a rule graph may be independent of each other (e.g., r2, r3 in Figure 1.b), they can be executed in parallel. The existing active DBMSs which are not implemented on parallel computer systems do not take advantage of parallelism among the independent rules. Since rule processing in active systems represents a significant burden on system performance, it is important to exploit the parallel execution properties of independent rules during their execution.

In this paper, we introduce the concept of *rule graph* to provide more flexible and expressive rule control mechanism. In our approach, event part is separated from condition, action parts of the ECA rule and associated with the rule graph which represents a set of rules (condition-action pairs) 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 [8] and ConTract [35] to model the control structure of operations which define a transaction. The graph-based model not only has higher expressive power than the linear (flat) or tree-structured (nested) transaction models, but also 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 the structure of a transaction graph, different control points corresponding to different trigger times (or coupling modes) can be clearly identified for placing a triggered rule graph at an appropriate control point according to its trigger time. For example, the rule graph with trigger time 'after' is placed just before the commit node of the triggering transaction graph, so that the rule graph is executed just before the commit operation. In this work, we also exploit the parallel execution properties of independent rules (DB operations) in a rule (transaction) graph. We define correctness criteria 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.

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, then discuss four different trigger times that provide the flexibility for a triggered rule graph to execute at different control points. In Section 3, we describe the graph-based transaction model and illustrate how it can naturally incorporate rule graphs and trigger times. 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. A conclusion is given in Section 7.

2 Flexible and Expressive Rule Control

In active DB environments, a single event can trigger multiple rules and a set of rules can be triggered by multiple events. In existing systems, control structure for multiple rules triggered by a single event is specified using rule priorities [33, 17, 1, 6]. In a priority-based approach, when a set of rules is triggered by an event, rules in that set have to follow the same control structure (which is derived from rule priorities) during their execution, independent of the event that triggered them. This is because events as well as priorities 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 the concept of a *rule graph* to specify control structures among CA rules in a more flexible fashion than with fixed

priorities. In our approach, events are associated with rule graphs instead of individual rules so that CA rules can follow different control structures when triggered by different events. 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.

2.1 Rule Graphs

A rule graph is defined by a set of CA rules having some control structure among them. The control structure is represented by a directed acyclic graph (DAG). 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.

The control structure of a rule graph can be specified by the following control associations among CA rules.

- *Sequential association:* denoted by "S" and the semantics of

```
{  r1:
     S: r2; }
```

  is that rule r2 should *sequentially* follow r1.

- *Precede-set association:* denoted by "P" and the semantics of

```
{  r:
     P: R; }
```

  where R is a set of rule names, is that all the rules named in R should begin only after the completion of rule r and they are independent of one another (they may be executed in parallel in a parallel computing environment).

- *Sync-at association:* denoted by "Y" and the semantics of

```
{  r:
     Y: R; }
```

  where R is a set of rule names, is that all the rules named in R should *synchronize* before rule r and they are independent of one another.

Note that "P" and "Y" associations are equivalent to "S" when R contains a single rule name. However, "S" is included as a control association to allow the specification of a simple precedence relationship between a pair of rules. An example rule graph specification corresponding to Figure 1.b is given below.

```
Rule_graph        RG1  is

 Triggered:       Before General_diagnosis()

    r5:
       S:         r6;
    r6:
       P:         r2, r3;
    r3:
       S:         r4;
    r1:
       Y:         r2, r4;

End               RG1;
```

In our object-oriented data model [34] and its extension, rule graphs applicable to instances of an object class are defined in the RULE GRAPHS section of the class definition. An example class definition is given in Figure 2. Rule graphs applicable to multiple classes are defined in the superclass of these classes. They are accessible to their instances through the inheritance mechanism. Thus, rule graphs are distributed among object classes and are readily available for use in object processing. In the above rule graph specification, r1, r2, .. r6 are rule names. The actual rules are separately defined in the RULES section of a class. A rule defined once can be named in multiple rule graphs. A rule has a rule name and Condition, Action and Otherwise clauses. Every rule must have a rule name but each of the three clauses is optional. C-A, A, and C-O are meaningful combinations. When a rule is processed, the condition is evaluated first, and based on the result, the action part or the otherwise part is executed.

2.1.1 Flexibility

It can be observed that the rule 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, the earlier defined rule graph RG1 specifies that rules r1-r6 should follow the control structure shown in Figure 1.b when they are triggered by the method General_diagnosis(). As shown below, another rule graph RG2 can be defined to specify that the same rules should follow the control structure shown in Figure 1.a when they are triggered by the method Engine_overheating().

```
/* Each class definition consists of five sections:
   ASSOCIATIONS, METHODS, RULE GRAPHS, RULES, and IMPLEMENTATIONS */

Entity_class  Vehicle  is
  ASSOCIATIONS:
     Generalization of Car, Truck, Van;
      Aggregation of
          public: /* definition of public attributes */
            V_Series: integer;
            V_Make: date;
            V_Engine: Engine;

          protected: /* definition of protected attributes */
            Vehicle_No: integer;

  METHODS: /* the signature of methods */
      public:
          method test_hose() : string;
          method test_fanbelt() : string;
             .
             .
   RULE GRAPHS: /* Definitions of rule graphs */
       Rule_graph  RG1  is

            Triggered: Before  General_Diagnosis()

             r5:
                 S:  r6;

             r6:
                 P:  r2, r3;

             r3:
                 S:  r4;

             r1:
                 Y:  r2, r4;

       End  RG1;
          .
          .
          .

  RULES:
    Rule       r1  is
    Condition:  test_hose()  = "broken"
    Action:    change_belt();
    Otherwise: -----
         .
         .
         .
 IMPLEMENTATIONS :    /* actual coding of methods */

    method test_hose(): string is
     {
       begin ..... end
     }
         .
         .
         .
 END Vehicle;
```

Figure 2: An Example Class Definition

```
Rule_graph        RG2  is

 Triggered:       After Engine_overheating()

    r1:
       S:         r2;
    r2:
       S:         r3;
    r3:
       S:         r4;
    r4:
       S          r5;
    r5:
       S          r6;

End               RG2;
```

2.1.2 Expressiveness

In addition to being flexible, the specification of rule control using a general DAG structure is also expressive enough to define the parallelism among rules explicitly and to accommodate new rules with the desired control requirements. For example, using the control associations P and Y, the independent nature of a set of rules can be explicitly defined. The explicit specification of independent rules is useful in parallel environment since it enables the rule processor to decide which rules to process in parallel. Additionally in rule graphs, new rules with desired control requirements can be inserted easily. For example, consider the following ECA rules (with the numeric priorities in parentheses) R1(1), R2(2) and R(3) each of which has E1 as its event. They can be represented by a rule graph: $E1 : r1 \rightarrow r2 \rightarrow r3$ 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: $E2 : r7 \rightarrow r8 \rightarrow r10$. Consider the case where a new rule R20 with "E1 or E2" as its event needs to be inserted into 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 inserting r20 which represents the condition-action pair of R20 in the RULES section of the appropriate class and adding the name r20 to the above rule graphs as shown below:
E1: r1→r20→r2→r3;
E2: r7→r20→r8→r10

Note that only the definitions of the above rule graphs are changed and the actual rule definitions (Condition, Action and Otherwise parts) remain intact. In this way, the specification of control structures among rules is clearly separated from the specification of rules themselves. This also provides an opportunity for a rule designer to clearly demarcate the control behaviour of rules

from their individual definitions.

2.2 Trigger Times

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.

- *Before*: The associated rule graph is executed before the trigger operation. This is useful for performing several tasks defined by rules before an operation (e.g., checking the access privileges of a user before performing a retrieval operation).
- *Immediately-after*: The associated rule graph is executed immediately after the trigger operation. This is useful for processing several integrity constraints in a specific (partial) order, immediately after an update operation.
- *After*: The associated rule graph is executed just before the commit time. This is used to delay a set of interrelated tasks defined by rules right before the commit time, to make sure that they are performed only when the transaction will surely commit.
- *Parallel*: The associated rule graph is executed as a separate transaction graph. This is used to execute some rule graphs as separate transaction graphs in parallel to the triggering 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 complex rule graphs with different trigger times.

3 Transaction Graph Model

In a DB environment, all operations are executed in a transaction framework. Rule graphs are not an exception. Although it is possible to implement the execution of rule graphs using a flat or nested transaction model, the linear and tree control structures of these cannot adequately model the more general structures of rule graphs.

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 3.a. In this Figure, t1-t5 are different DB tasks. Boxes represent the spheres of control [10] and the directed edges represent the control structure. This can be shown in a tree form as

depicted in Figure 3.b, in which solid lines represent parent-child relationships and dashed lines represent the control structure among the tasks. Each task in turn can have a graph structure. In this paper, we will call such a transaction a *transaction graph* (TG).

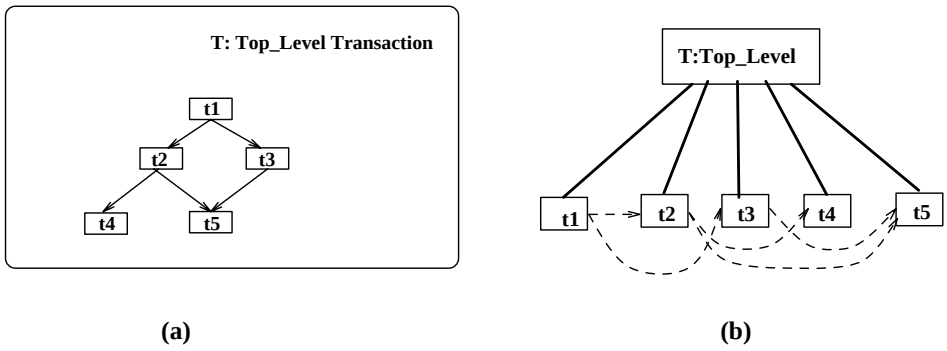

Figure 3: Transaction Graph Model

3.1 Expressiveness of a Transaction Graph

3.1.1 Modeling Rule Graph Execution

Since a rule graph and a transaction graph have the same structure, rule graphs can be easily incorporated into the transaction graph's control structure. For example, if a transaction T triggers the rule graph shown in Figure 4, the rules r1-r5 are treated as subtransactions (or tasks) of T, as shown in Figure 5. The TGM also captures the nested triggering of rule graphs. Rules of a triggered rule graph are modeled as subtransactions of the triggering rule and they have their own control structure. For example, if an operation activated by rule r2 (of the TG shown in Figure 5) triggers the rule graph shown in Figure 6.a, the rule graph is incorporated into the transaction graph structure as shown in Figure 6.b. The correctness criteria and the concurrency control method for the execution of a transaction graph is discussed in Section 3.2.

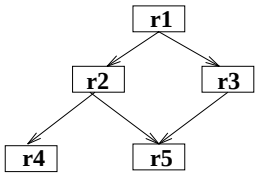

Figure 4: A Rule Graph

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.

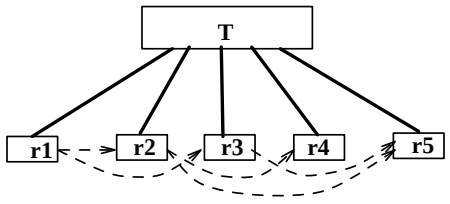

Figure 5: Active Rules as part of a Transaction Graph

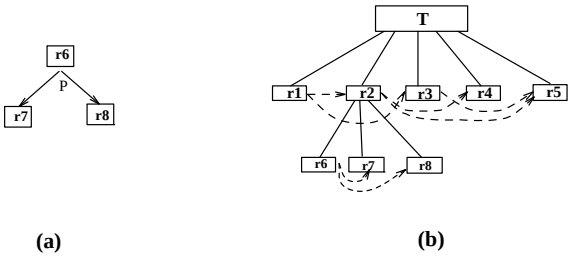

Figure 6: Modeling Nested Triggerings of Rules

3.1.2 Modeling Trigger Times

To include a rule graph with the trigger time *after* at the end of the TG (or just before the commit operation), the TG should have beginning and ending points. We add a *begin-task*, which performs the initialization, before the beginning of the other tasks and a *commit-task*, which commits the transaction atomically, after all the other tasks, as shown in Figure 7.a. This can be represented using the tree form as shown in Figure 7.b. A triggered rule graph is incorporated into the triggering TG's structure according to its trigger time, as explained below.

- 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 7.b triggers the rule graph in Figure 8.c, it is added to the TG as shown in Figure 8.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. In the above example, if the rule graph has "after" as the trigger time, it is added to the TG as shown in Figure 8.b. Before the triggered rule graph can be executed, another rule graph (with trigger time "after") may be triggered by another task, say t4. In this case, the newly triggered rule graph is added just before the commit task of the TG, however, after the rule graphs triggered earlier. In this way, all the rule graphs whose execution is delayed until just before the commit time are executed in the same order in which they were triggered. It is important for some applications to maintain such order of execution [18].

- In 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 the trigger times, the control structure of rule graphs uniformly fits into the proposed transaction control structure. This obviates a transaction manager's treating rules differently from other subtransactions with respect to the scheduling, the correctness criterion and the concurrency control techniques associated with transaction execution.

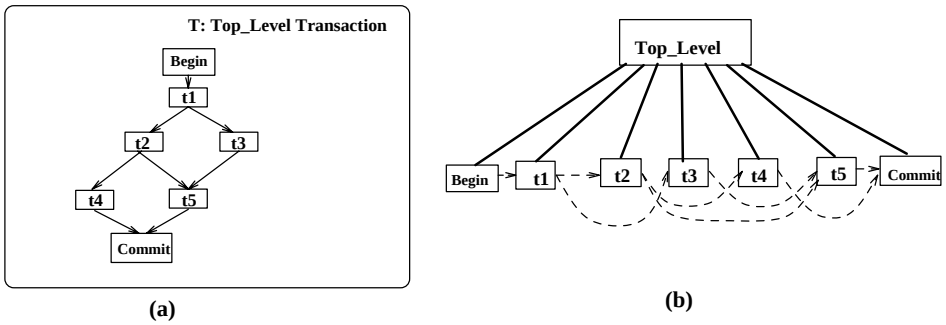

Figure 7: Adding begin and end points to a TG

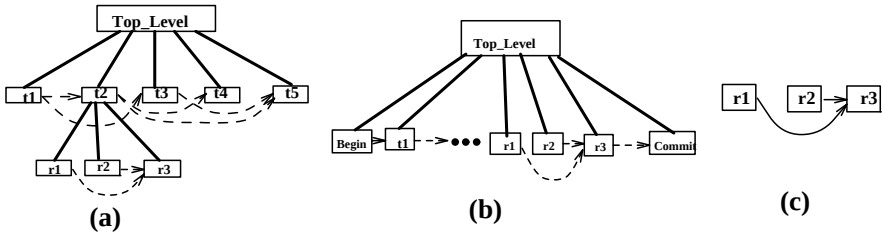

Figure 8: Incorporating Triggered Rule Graphs into the TG's Structure

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. Serializability alone (any serializable execution) 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 correctness.

3.2.1 Correctness Criterion

At the transaction-level, the correctness criterion for concurrent execution of several TGs is the standard serializability [13] 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 4, the serial order $r1 \rightarrow r4 \rightarrow r2 \rightarrow r3 \rightarrow r5$ is not correct). The concurrent execution of rules (tasks) in a rule (transaction) graph is *correct* if and only if the resulting serial order does not violate the control structure of the rule (transaction) graph, (e.g., the correct execution orders for the TG shown in Figure 3.a are $r1 \rightarrow r2 \rightarrow r3 \rightarrow r4 \rightarrow r5$ or $r1 \rightarrow r3 \rightarrow r2 \rightarrow r4 \rightarrow r5$). To be more general, the concurrent execution of rules is correct when it is equivalent to some *topological serial order* [9] of rules.

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 those 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.

We now describe a scheduling algorithm that exploits the parallelism among independent rules while enforcing the topological order and a locking scheme that maintains atomicity and isolation of rules in the following three subsections.

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. In other words, a node with a zero *indegree* (number of incoming edges) in a rule graph represents an independent rule. For example, in the rule graph shown in Figure 4, r1 is independent and, r2 and r3 become independent after the completion of r1.

To maintain a topological order, rules in a rule graph may be executed in a topological order sequentially, at the expense of not being able to take advantage of the potential inter-rule parallelism. To achieve inter-rule parallelism, 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 *predecessor*[1]. 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 4 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. Although the sequentiality among the groups may reduce the degree of inter-rule parallelism, it is necessary for maintaining the correctness. However, the sequentiality can be ameliorated by scheduling the rules *asynchronously*, which we shall discuss in the next section.

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 execution[2]. 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

In the above scheduling algorithm, the `wait` step waits for all the scheduled rules to complete before scheduling the next set of independent rules. This synchronization step can make some rules wait *unnecessarily*. For example, consider the rule graph shown in Figure 4. It is scheduled in the following sequence of groups $\{r1 \rightarrow (r2, r3) \rightarrow (r4, r5)\}$ 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 wait-step monitors

[1] Note that this is different from child-parent or ancestor-descendant relationships, which are denoted by solid lines in the tree representation used.

[2] On a uniprocessor machine, the rules can be executed as child processes or threads.

the queue for the acknowledgment from at least one of the scheduled rules and the scheduler 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 are removed from the rule graph as are their outgoing edges. The `remove` step also removes the acknowledgments from the queue. This approach avoids unnecessary waiting 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 before the abort.*

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 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 [20] for the proofs.

The recovery of a transaction graph can be done using the variants of standard recovery methods used for the nested transaction model [25, 28]. The only difference is that, when a set of rules (tasks) in a rule (transaction) graph are being undone, they need to follow *reverse* topological order. A detailed discussion of recovery techniques 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 a parallel active OO knowledge base management system - OSAM*.KBMS/P

implementation [7, 23, 26]. The objective of this implementation is to test the implementability of the proposed models and demonstrate that the parallel execution property of graph structures can be exploited to achieve efficient transaction and rule execution. In this section, we describe the following features of our implementation: (i) its client/server architecture in which the transaction server is designed to be scalable and asynchronous, (ii) its transaction ID system, which assigns unique IDs to transactions and rules, helps maintain the atomicity of transactions and rules, and helps process lock requests efficiently, and (iii) dynamic launching of rule processors for processing dynamically triggered rules.

4.1 Client/server Architecture

The overall architecture of OSAM*.KBMS/P, as shown in Figure 9, is based on the standard client/server architecture explained in [29]. The clients C1, C2,...Cn, each of which is running on a workstation connected to the server by an inter-connection network (Ethernet), each have an X-Motif graphical user interface (GUI) for editing and browsing TGs, rule graphs, queries and DB schemas [22]. A client translates a TG into an intermediate form and sends it to the server through the inter-connection network. For efficient processing of transaction and rule graphs, we have implemented the server on a 64-node nCUBE2. The server's architecture is designed to be scalable and asynchronous.

As shown in Figure 9, the server has a global transaction manager (GTM) and a group of local transaction managers (LTM) which are launched on different processors of the nCUBE2. GTM receives the incoming TGs and supervises their asynchronous execution. All computation intensive functionalities, namely, data processing, lock management, recovery management and rule processing are shifted to the LTM module so that the processing is distributed among multiple LTMs. Several replicas of the LTM are launched on different nodes of nCUBE2. With a minor change in the global data dictionary, any number of LTMs can be launched to scale up the system's performance.

The server processes transactions asynchronously. All the modules (e.g., transaction scheduler, task processor) in the server interact with each other using messages to cooperatively process transactions. For example, when the transaction scheduler (TSC) schedules a set of tasks for execution, it does not wait for their completion. Instead, it will switch to another transaction graph. All waiting tasks are kept in a wait queue. The TSC resumes the processing of a transaction graph in the wait queue as soon as it receives an acknowledgment from at least one of the scheduled tasks. All the acknowledgments are automatically queued in the acknowledgment queue. Note that, the TSC is effectively following the asynchronous scheduling algorithm explained in Section 3.2.3. The synchronization among the transactions and rules that are being processed asynchronously is achieved by our transaction ID system and the locking scheme. None of the modules waits for a message.

Instead, all the messages directed to a module are automatically stored in a queue associated with that module. It can be observed from Figure 9 that the GTM maintains a transaction queue and the LTM maintains a task queue. Each module processes the messages in its queue in FIFO order.

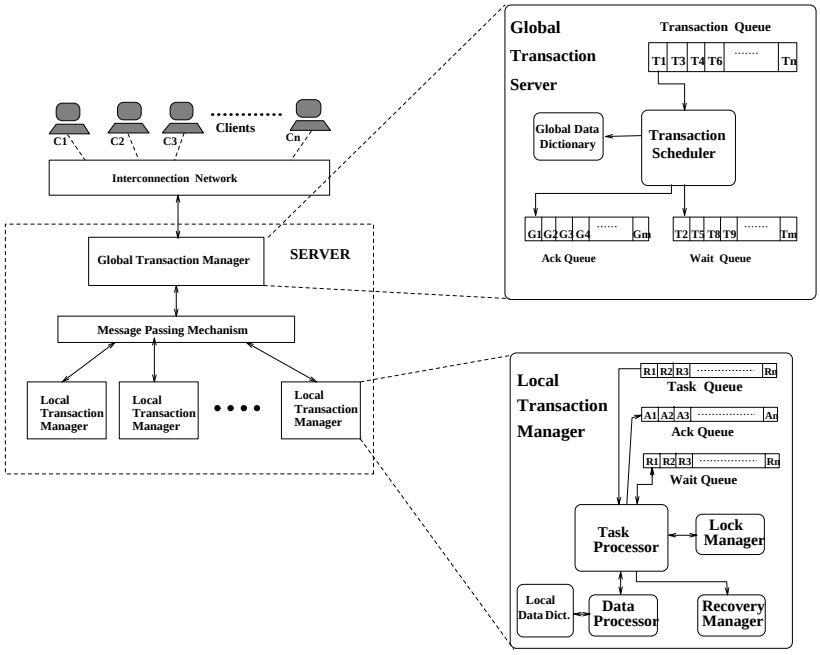

Figure 9: System Architecture of OSAM*.KBMS/P

4.2 Transaction ID System

During the execution, the structure of a transaction graph can become very complex when the number of rule graphs with different trigger times are triggered and executed as a part of the transaction graph. The execution of different tasks and rules generated by the transaction graph may be spread over a number of nodes. It is important to bind all the operations generated by a transaction graph together to maintain its atomicity. Also, it is important to know the ancestors of a given rule/task. We use a transaction ID system in which an ID helps in grouping all related operations together as one logical unit.

A transaction ID (or TRID) is given to each TG. The TRID is the glue that binds all the actions of a TG together. All TRIDs assigned to currently executing transactions are unique. All processing nodes in an nCUBE2 parallel computer have unique node_id numbers. Globally unique TRIDs are generated by concatenating the node_id of the node where the transaction scheduler (TSC) is executing (in case there are multiple TSCs) with a sequence number which is guaranteed to be unique for each TG executing at a particular node.

All TGs contain tasks, and in order to maintain the atomicity and isolation of tasks, all tasks within a TG are assigned unique identifiers (TASK_IDs). The TASK_ID contains TRID of the

parent TG as a prefix. Each rule is given a unique RULE_ID. In order to allow other modules (e.g., lock manager) to independently determine the lineage of a rule, a rule's family history is incorporated into its RULE_ID in the following manner. For each rule in a triggered rule graph, its RULE_ID is formed by appending a number which is unique among all the rules in the rule graph to the ID of the triggering task or rule. A linked list data structure is used to represent RULE_IDs which are of variable length. It should be noted that from a TASK_ID (RULE_ID) one can discern where in the system the task (rule) was created, what LTM is responsible for its execution, and also the IDs of all its ancestors. Such information allows modules such as the Lock and Recovery Managers to act based on an ID, without cluttering the communication network with inquiries about tasks. In addition, all the operations generated by a logical unit (e.g., rule, task, transaction) can be processed in a single sphere of control even though the operations are distributed among multiple nodes.

4.3 Dynamic Launching of Rule Processors

Another important feature in our implementation is that several rule processors (RProcs) are dynamically launched when there are rules to be executed and they are terminated automatically as soon as the assigned rules have been executed.

Before scheduling a rule graph, a TSC assigns unique RULE_IDs to triggered rules. This RULE_ID contains ID of the triggering task which in turn has the IDs of all its ancestors, as explained earlier. The TSC identifies independent rules in the rule graph and depending on the number of independent rules, dynamically launches the same number of RProcs 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. Each rule may consist of a condition, an action and an otherwise part. A condition can be a boolean expression or a DB query expression. Action and otherwise clauses can be DB query expressions or error messages. If the condition evaluates to true or returns a non-null data set, then the action part is executed. Otherwise, the otherwise part is executed. 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 RProc processes only when there are rules to be processed reduces the load on the system significantly. It also provides TSC with the flexibility to choose an ideal node for processing a rule dynamically.

5 Performance Evaluation

For performance evaluation, we used a "Registration transaction" which has the graph structure shown in Figure 10.a, and a rule graph shown in Figure 10.b. The latter is triggered by the task "Calculate Debt".

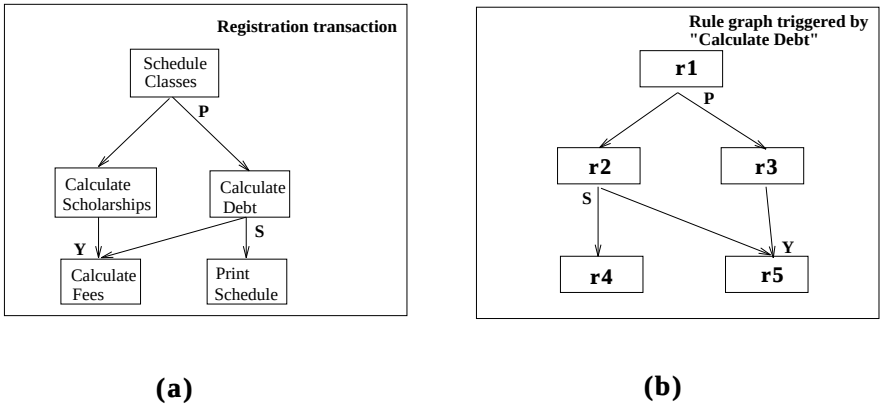

Figure 10: Transaction and Rule Graphs used for Performance Evaluation

We run a large number of Registration transactions on the server with one global unit (a GTM launched on a cube of 4 nodes) and one local unit (an LTM launched on a cube of 4 nodes). Later, by keeping the number of transactions and global units constant, we gradually increased the number of local units 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 follows:

$$Speedup = \frac{Time\_taken\_by\_640\_transactions\_in\_a\_system\_with\_one\_global\_unit\_and\_one\_local\_unit}{Time\_taken\_by\_640\_transactions\_in\_a\_system\_with\_one\_global\_unit\_and\_n\_local\_units}$$

It was observed that the increase in speedup was linear up to 10 local units and it gradually tapered as the number of local units 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 inter-processor communication.

We also evaluated the *scaleup* of the system by running 40 Registration transactions on the system with one global unit and one local unit, then increased the number of transactions and the number of local units in the same proportion (e.g., 80 transactions on (1 global unit and 2 local units), 120 transactions on (1 global unit and 3 local units), etc.). The formula used for the scaleup is :

$$Scaleup = \frac{Time\_taken\_for\_processing\_1*40\_transactions\_on\_1\_local\_unit}{Time\_taken\_for\_processing\_n*40\_transactions\_on\_n\_local\_units}$$

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 global unit and 13 local units)". 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 Related Work

There have been several research efforts on rules, rule control and transaction framework in both AI and DB areas [18, 33, 11, 17, 30, 36, 3, 31, 5, 14, 27]. We shall relate our work on graph-based rule and transaction models to them.

The rule subsystems of active DBMSs: POSTGRES [33], Ariel [17], HiPAC [11], Starburst [36] 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.

In [31], RDL1 supports a set of control constructs namely 'sequence', 'disjunction' and 'saturate', for specifying the control structures among rules. In RDL1, rules are defined in modules. Each module contains a rule section in which rules are defined, and a rule control section in which the control structure among these rules is defined. However, a set of rules defined in a module cannot follow different control structures when they are triggered by different events. For a set of rules to follow N different control structures when they are triggered by N different events in RDL1, the same rules must be defined N times in N different modules. We have presented the concept of rule graphs in which we have associated events with rule graphs instead of individual rules so that different control structures can be specified for the same rules when they are triggered by different events. Since rule graphs contain only rule names, rules need not be defined repeatedly if they participate in 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. In [18], Hsu, Ladin and McCarthy describe a more expressive model which is basically an extended nested transaction model to capture rules and nested triggerings of rules uniformly. Other significant works that use variants of the nested transaction model for modeling the execution of rules are [3, 5, 27]. However, the tree-based structure of the nested model is not expressive enough to capture graph-based control structures among rules in a uniform fashion. Our graph-based transaction model not only captures the triggered rule graphs uniformly but also places them at an appropriate control point of the transaction structure according to their trigger times. The nested transaction model is a special case of the graph-based model, i.e., the case when triggered rules are not connected by a control structure. Several expressive transaction models have

been reported in [8, 35, 4, 2]. However, they do not focus on the uniform incorporation of rules, rule control and trigger times in the transaction framework.

The majority of existing active DBMSs have been implemented in sequential and centralized environments. In these systems, a single rule is selected in the conflict resolution phase, although multiple rules can be executed (or fired) without violating priorities. In the field of Artificial Intelligence, a significant amount of research has focused on parallel rule execution and rule control [30, 19, 32, 21]. Similar to our system, RUBIC [21] and PARULEL [32] provide several control constructs to control the parallel execution of rules, nevertheless, they do not guarantee the serializable execution of rules. Rule systems presented in [30, 19] guarantee serializability during the parallel execution of rules by the static analysis of rules. However, they do not provide any rule control constructs. Our system has been implemented on a shared-nothing parallel computer. It exploits the parallelism by executing independent rules (tasks) of a rule (transaction) graph in parallel. The serializability among rules is enforced dynamically by a locking scheme which is more relevant to DB environments when compared with the static analysis of some AI production systems.

7 Conclusions

The majority of contemporary active DBMSs adopt priority-based approaches for specifying and enforcing control structures among rules. Priority-based approaches are not flexible enough to specify a set of rules that follow different control structures when they are triggered in different contexts. Also, they are not expressive enough to accommodate new rules with more complex control requirements and to express parallel properties of rules. A more flexible and expressive rule control mechanism is needed for many advanced DB application domains such as CAD/CAM, CASE, CIM and FMS. In this paper, we have provided a flexible and expressive 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 be inserted. Using the control associations proposed in this paper, parallelism among rules can be explicitly specified. To execute graph-based rules 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 model has 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 [15]. Furthermore, it is general enough to have rules

executed at arbitrary points[3] 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

[1] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 479–487, Barcelona (Catalonia, Spain), September 1991.

[2] P. Attie, M. Singh, M. Rusinkiewicz, and A. Sheth. Specifying and enforcing intertask dependencies. In *Proc. 19th Int'l Conf. on Very Large Data Bases*, August 1993.

[3] C. Beeri and T. Milo. A model for active object oriented database. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 337–349, Barcelona, September 1991.

[4] A. Buchmann, M. T. Ozsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A transaction model for active distributed object systems. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 123–158. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[5] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proc. 18th Int'l Conf. on Very Large Data Bases*, pages 339–352, Vancouver, September 1992.

[6] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *to appear in Proc. of 20th Int'l Conf. on Very Large Data Bases*, Santiago, Chile, September 1994.

[7] P. V. Cherukuri. A task manager for parallel rule execution in multi-processor environments. Master's thesis, Department of Electrical Engineering, University of Florida, 1993.

[8] P. K. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 349–398. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, pages 536–538. McGraw Hill Book Company, New York, 1990.

[10] C.T. Davies. Recovery semantics of a db/dc system. In *Proc. of ACM national conference*, 1973.

[11] U. Dayal, B. Blaustein, A. Buchmann, S. Chakravarthy, et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[12] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: A uniform approach. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 317–326, Barcelona, September 1991.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in database systems. *Communications of the ACM 19*, 10(11), Nov. 1976.

[14] O. Etzion. PARDES-a data-driven oriented active database model. *SIGMOD Record*, 22(1):7–14, 1993.

[15] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 81–90, San Diego, CA, June 1992.

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

[16] J. Gray and A. Reuter. *Transaction processing: Concepts and techniques*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[17] E. N. Hanson. An initial report on the design of Ariel: a DBMS with an integrated production rule system. *SIGMOD Record*, 18(3):12–19, September 1989.

[18] M. Hsu, R. Ladin, and D. McCarthy. An execution model for active database management systems. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, pages 171–179, Washington, DC, June 1988.

[19] T. Ishida. Parallel rule firing in production systems. *IEEE Trans. Knowledge Data Eng.*, 3(1):11–17, March 1991.

[20] R.S. Jawadi. *Graph-based rule and transaction execution in an Object-oriented knowledge base management system*. Ph.D. dissertation, Department of Computer and Information Sciences, University of Florida, 1994.

[21] S. Kuo and D. Moldovan. Implementation of multiple rule firing production systems on hypercube. *Journal on Parallel and Distributed Computing*, 13(4):383–394, December 1991.

[22] H. Lam and S.Y.W. Su. GTOOLS: An active graphical user interface toolset for an object-oriented KBMS. *International Journal of Computer System Science and Engineering*, 7(2):69–85, April 1992.

[23] Q. Li. Design and Implementation of a Parallel Object-Oriented Query Processor for OSAM*.KBMS/P. Master's thesis, Department of Electrical Engineering, University of Florida, 1993.

[24] E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, 1985.

[25] E. Moss. Log-based recovery for nested transactions. In *Proc. 13th Int'l Conf. on Very Large Data Bases*, pages 427–432, Brighton, England, September 1987.

[26] R. Nartey. The design and implementatation of a global transaction server and a lock manager for a parallel knowledge base management system. Master's thesis, Department of Electrical Engineering, University of Florida, 1994.

[27] L. Raschid, T. Sellis, and A. Delis. A simulation-based study on the concurrent execution of rules in a database environment. *Journal on Parallel and Distributed Computing*, 20(1):20–42, Jan 1994.

[28] K. Rothermal and C. Mohan. ARIES/NT: A recovery method based on write ahead logging for nested transactions. In *Proc. 15th Int'l Conf. on Very Large Data Bases*, pages 337–346, Amsterdam, The Netherlands, August 1989.

[29] N. Roussopoulos and A. Delis. Modern client-server DBMS architectures. *SIGMOD Record*, 20(3):52–61, September 1991.

[30] J. G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal on Parallel and Distributed Computing*, 13(4):348–365, December 1991.

[31] E. Simon, J. Kiernan, and C. deMaindreville. Implementing high level active rules on top of a relational DBMS. In *Proc. 18th Int'l Conf. on Very Large Data Bases*, pages 315–326, Vancouver, 1992.

[32] S. J. Stolfo, O. Wolfson, P. K. Chan, H. M. Dewan, L. Woodbury, J. S. Glazier, and D. A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal on Parallel and Distributed Computing*, 13(4):366–382, December 1991.

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

[34] S. Y. W. Su, V. Krishnamurthy, and H. Lam. An object-oriented semantic association model (OSAM*). In S. Kumara, A. L. Soyster, and R. L. Kashyap, editors, *Artificial intelligence: Manufacturing theory and practice*, pages 463–494. Institute of Industrial Engineers, Industrial Engineering and Management Press, Norcross, GA, 1989.

[35] H. Wachter and A. Reuter. The ConTract model. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 219–264. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[36] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 275–285, Barcelona (Catalonia, Spain), September 1991.