OSAM*.KBMS/P: A Parallel, Active, Object-oriented Knowledge Base Server

Stanley Y.W. Su *Ramamohanrao Jawadi*
Prashant Cherukuri *Qiang Li* *Richard Nartey*

Technical Report TR94-031

Database Systems Research and Development Center
Department of Computer and Information Sciences
University of Florida, Gainesville, FL 32611
su,rsj,pvc,ql,rn@cis.ufl.edu

Abstract

An active object-oriented knowledge base server can provide many desirable features for supporting a wide spectrum of advanced and complex database applications. Knowledge rules, which are used to define a variety of DB tasks to be performed automatically on the occurrence of some events, often need much more sophisticated *rule specification and control mechanisms* than the traditional priority-based mechanism to capture the control structural relationships and parallel execution properties among rules. The underlying object-oriented knowledge representation model must provide a means to model the structural relationships among data entities and the control structures among rules in a uniform fashion. The transaction execution model must provide a means to incorporate the execution of structured rules in a transaction framework. Also, a parallel implementation of an active knowledge base server is essential to achieve the needed efficiency in processing nested transactions and rules. In this paper, we present the design and implementation of a parallel active OO knowledge base server which has the following features. First, the server is developed based on an extended OO knowledge representation model which models rules as objects and their control structural relationships as association types. This is analogous to the modeling of entities as objects and their structural relationships as association types. Thus, entities and rules, and their structures can be uniformly modeled. Second, the server uses a graph-based transaction model which can naturally incorporate the control semantics of structured rules and guarantee the serializable execution of rules as subtransactions. Thus, the rule execution model is uniformly integrated with that of transactions. Third, it uses an asynchronous parallel execution model to process the graph-based transactions and structured rules. The server named OSAM*.KBMS/P has been implemented on a shared-nothing multiprocessor system (nCUBE2) to verify and evaluate the proposed knowledge representation model, graph-based transaction model, and asynchronous parallel execution model. The results of a performance evaluation are presented.

1 Introduction

There are three important trends in the current database management research: (i) parallel database servers, (ii) active database management systems (DBMSs), and (iii) object-oriented (OO) DBMSs. Each focuses on enriching some aspects of the traditional DBMS technology [5, 14, 22]. *Parallel data servers* aim to achieve high performance by exploiting maximum parallelism using

multiprocessor architectures [14, 47]. The main feature of an *active DBMS* is to react autonomously (without user intervention) to different events that can occur in a DBMS [16, 30, 29]. The main goal of an *object-oriented DBMS* is to model and process the structural and behavioral properties of complex real world objects naturally and efficiently [4, 44, 5]. Although significant research has been carried out in the above areas separately, little attention has been given to the integration of all their features in a unified system. It is necessary to combine the modeling concepts and implementation techniques introduced in these three research focuses for coping with the modeling, functional, and performance requirements of many advanced and complex DB applications. In this paper, we present the features, design and implementation of OSAM*.KBMS/P which is a parallel active OO knowledge base server. It uses a parallel version of an Object-oriented Semantic Association model (OSAM*/P) as its underlying knowledge representation model.

Traditionally, event-condition-action (ECA) rules are used to implement active features in a DB environment [13]. 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 satisfied. In this paper, we shall separate the event part from the condition and action parts of an ECA rule. We shall use "event" and "trigger operation" interchangeably and use "CA rule" (or simply rule) to refer to the C and A parts of an 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 execution of a set of semantically interrelated rules, and different rule execution orders (or control structures) need to be followed when triggered by different events. For example, in the diagnosis and test stage of car manufacturing, the event "Engine_overheating" may require the diagnosis steps defined by CA rules r1-r6 (shown in Figure 1) to be followed in the order of Figure 1.a which is different from one of many possible control structures (e.g., Figure 1.b) for the event, "General_diagnosis", due to the different semantics of the events. The priority-based rule control mechanisms adopted by the existing active and/or object-oriented DBMSs (e.g. POSTGRES [42], HiPAC [13], Alert [1], Starburst [8], Ariel [21] and Sentinel [9]), are not sufficient for capturing the complex control semantics of CA rules in current DB applications, because priorities are assigned to ECA rules and a set of CA rules associated with a number of events cannot follow different execution orders based on the occurrence of different events. In the example above, if fixed rule priorities are assigned to ECA rules R1-R6 (which have "Engine_overheating OR General_diagnosis" as their E part and, CA rules r1-r6 as their C and A parts), they follow the same execution order (specified by the priorities) when they are triggered by two different events namely "Engine_overheating" and "General_diagnosis". Also, using priorities, parallel execution properties of rules cannot be specified explicitly and the rule control specification cannot be separated from the rule specification.

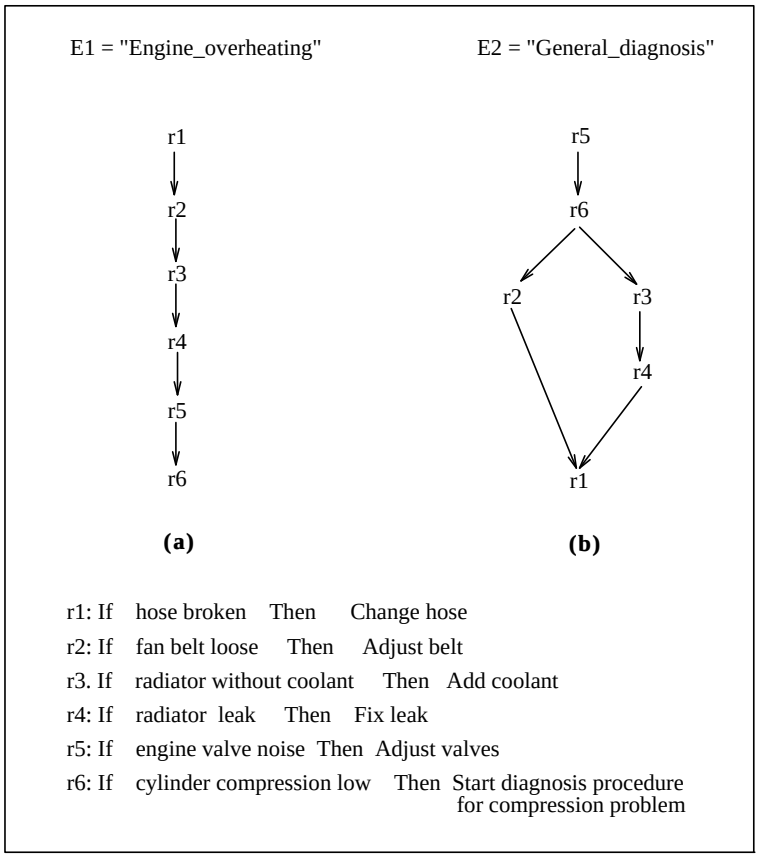

Figure 1: Rules for Diagnosing an Engine

In an active OODBMS, a transaction may be composed of several application/database tasks sharing a specific control structure (e.g., see the "Registration transaction" shown in Figure 2 which consists of a set of tasks to be done, following a specific control structure). Each task may trigger a set of rules which may also share a specific control structure, and the condition/action of each rule in turn may involve operations which can trigger more rules and these rules may be related by another control structure. Traditional transaction models such as the flat transaction model [20] and the nested transaction model [31], and their variants [50] are not expressive enough to model the above scenario in a uniform fashion. A more expressive and powerful transaction model (e.g., a graph-based model) is needed for defining and processing graph-structured transactions and rules uniformly.

An OODBMS is complex because of its powerful semantic features. Incorporating active rules into an OODBMS increases its complexity further. The performance of the system may not be able to satisfy real time requirements of many applications if it is implemented on a sequential computer. Even among parallel computers, a shared-memory computer limits the scalability of the system. It is therefore ideal to implement the system on a shared-nothing parallel computer for achieving

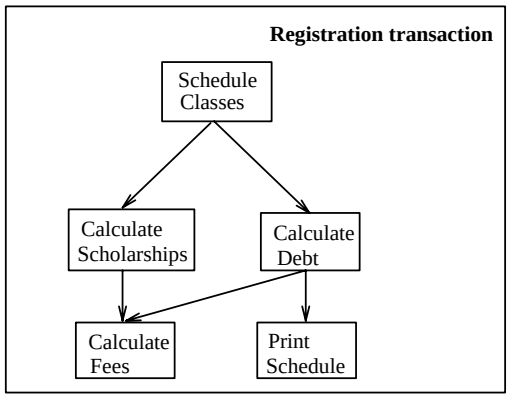

Figure 2: An Example Transaction

good speedups and scaleups. To achieve these, the software architecture of the system must take advantage of the scalability of the shared-nothing architecture. In a shared-nothing computer, all modules loaded on different nodes interact with each other using a message-passing mechanism. An asynchronous execution model in which all modules process their messages asynchronously (without waiting for messages from other modules) is useful for achieving high speedups.

The remainder of this paper is organized as follows. In Section 2, we survey some existing related works. In Section 3, we describe the knowledge representation model OSAM*/P which provides among other properties the facility for defining complex control structures among rules in its OO framework. In Section 4, we discuss how an expressive graph-based transaction model can uniformly incorporate the execution of structured rules. The graph-based transaction model also supports different coupling modes in transaction and rule execution. The correctness criterion for the execution of graph-based transactions is also presented. In Section 5, we present a scalable client/server-based architecture for OSAM*.KBMS/P and describe the asynchronous processing capability of each module in the architecture. We also present some techniques used in the implementation of OSAM*.KBMS/P on an nCUBE2 parallel computer. Finally, in Section 6, we evaluate the system's performance, and give conclusions and directions for future research in Section 7.

2 Related Work

The rule subsystems of the existing active and/or object-oriented DBMSs (e.g., POSTGRES [42], Ariel [21], HiPAC [13], active OODB [6], ADAM [15], DOM [7], Starburst [49], Ode [19], Alert [1] and SAMOS [18]) use either *priorities*, to define the control structures among rules, or support no rule control mechanism at all. However, the priority-based control mechanism is not flexible and

expressive enough to meet the requirement of structured rule execution found in some advanced DB applications. As explained in the introduction, when using priorities, it is not possible to specify different control structures for rules triggered by different events. Additionally, since the control structures of rule execution are implicitly specified by rule priorities, it is quite difficult to understand or modify the rule structures. In [26, 39], Simon et al emphasize the need for a more powerful rule control mechanism for making a rule system more practical. They propose a set of control constructs, namely 'sequence', 'disjunction' and 'saturate', for specifying the control structures among rules. In RDL1 (the active DBMS proposed by Simon et al), rules are defined in 'modules'. Rule control is clearly separated from rules. 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, in RDL1, 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, say, N different control structures when they are triggered by N different events, the same rules must be defined N times in N different modules. In OSAM*.KBMS/P, we use *rule graphs* to specify rule control structures. Since a rule graph specification contains only a structure of rule names, and the actual CA rules are defined separately, these CA rules need not be redefined if they participate in a number of rule graphs. We separate events from CA rules and associate them with rule graphs instead of individual rules, so that the same CA rules can follow different control structures when they are triggered by different events. In our OO model, rules are modeled as objects and the control relationships among rules as *control associations*. This is analogous to modeling semantic relationships among entity classes as *semantic associations*. This allows the specifications of rule control structures as an integral part of a knowledge base schema.

Many existing DBMSs use different transaction models to capture the complex execution of several interdependent DB tasks and rules. In POSTGRES and Ariel, the execution of rules is incorporated into a flat transaction model [20]. Hsu, Ladin and McCarthy [24] describe a more expressive model, which is basically an extended nested transaction model (NTM) [31], to capture rules and nested triggerings of rules uniformly. Other significant works that use variants of NTM for modeling the execution of rules are reported in [6, 8, 36]. The tree-based structure of NTM is not expressive enough to capture graph-based control structures among rules in a uniform fashion. , Buchmann et al [7] present an extended NTM in which subtransactions of a transaction can have a graph-based control structure (specified by a precedence graph). Other significant works that propose powerful and expressive transaction models are reported in [11, 48, 23, 3]. However, the above works do not deal with the uniform incorporation of rules, rule control and rule trigger times in a transaction framework. In our work, we use a graph-based transaction model which has the

same control structure as the models used in DOM [7] and ConTract [48]. In addition, we show how the graph-based transaction model can uniformly incorporate complex control structures among rules. We also show how rules can be placed at different control points in the transaction structure based on different trigger times. Furthermore, we discuss the correctness criterion and scheduling techniques for graph-based transaction executions.

The majority of the existing active and/or object-oriented DBMSs have been implemented in sequential and centralized environments. In these systems, when a set of rules are triggered, a single rule is selected for execution even though multiple rules can be executed (or fired) without violating rule priorities. In the field of Artificial Intelligence, a significant amount of research has been focused on executing rules in parallel and at the same time maintaining their control structures [38, 25, 40, 27]. Similar to our system, RUBIC [27] and PARULEL [40] 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 [38, 25] guarantee serializability during the parallel execution of rules by a static analysis of rules. However, they do not provide any rule control mechanism. Our system, which is implemented on a shared-nothing parallel computer, nCUBE2, exploits the parallel execution properties of rules by executing independent rules of a rule graph in parallel. The serializability among rules is enforced dynamically, by a locking scheme which is more relevant to database environments when compared with the static analysis of some AI production systems.

To summarize, the novel features of OSAM*.KBMS/P are the following: i) its expressive and flexible rule control mechanism for modeling and processing complex control structures among rules in an OO fashion, ii) its expressive transaction model for carrying out not only the execution of the traditional DB operations, but also structured rules, and iii) its scalable architecture and asynchronous execution model for enhancing scaleup and speedup.

3 OSAM*/P Knowledge Representation Model

In OSAM*/P, all items of interest in an application world such as physical entities, abstract entities, functions, events, processes, methods and rules, are uniformly modeled as objects, and various semantic and structural relationships among these objects are modeled as semantic associations. An application world is modeled by a *schema graph* which is basically a network of object classes and their semantic associations. An example schema graph for a university database is shown in Figure 3. Rectangular boxes and circles represent two different categories of object classes, namely, entity classes (E-Class) and domain classes (D-class), respectively. The sole function of a

D-class is to define a domain of possible values from which descriptive attributes of objects draw their values (e.g., integer, real, string). An E-class, on the other hand, forms a domain of objects which occur in an application's world (e.g., Person, Student). OSAM*/P also provides a number of semantic associations to model different relationships among E-classes. Among them are the *aggregation association* (A) and the *generalization association* (G). In Figure 3, the labels A and G stand for aggregation association and generalization association, respectively. An A-association signifies either a value attribute or a reference attribute depending upon whether its constituent-class is a D-class or an E-class respectively. For example, the E-class 'Transcript' in Figure 3 is defined as having three aggregation associations with object classes 'Student', 'Course' and 'Grade' (which are called constituent classes). The two A-associations with E-classes 'Student' and 'Course' define two reference attributes of 'Transcript' and the A-association with D-class 'Grade' defines a value attribute of 'Transcript'. A G-association represents a superclass-subclass relationship. For example, the E-class 'Person' is a superclass of E-class 'Student' and also of 'Teacher'.

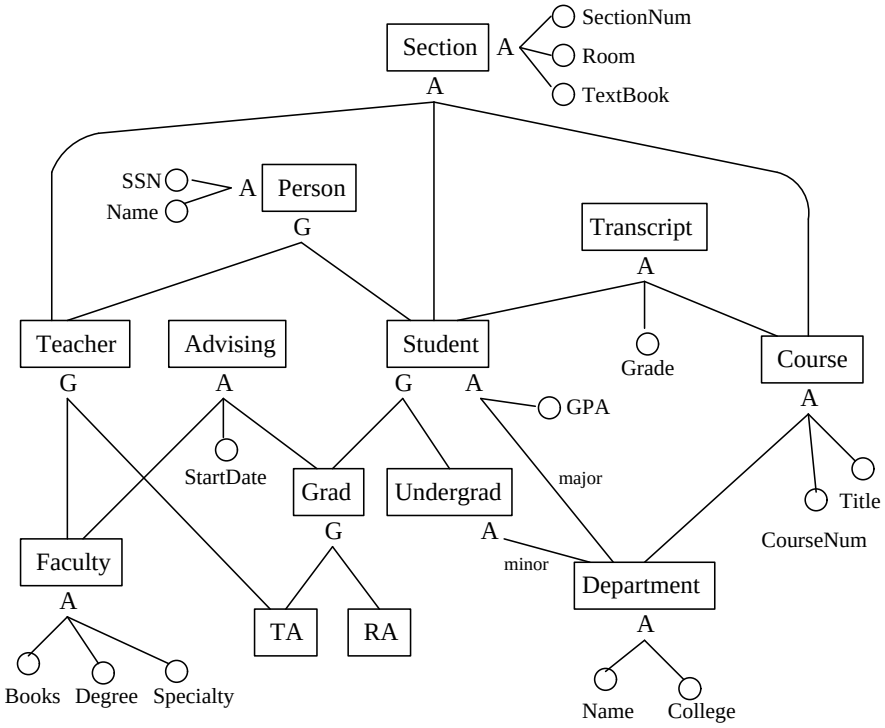

Figure 3: Example Schema of a University Database

OSAM*/P models the structural properties of entity types in an application domain using *classes* and *associations*, and their behavioural properties using *rules* and *methods*. In addition, OSAM*/P models complex control structures among rules using *rule graphs*. Since the other modeling features are explained in a previous paper [45], we describe only rules and rule graphs

in detail, here. Mainly, we want to show the flexibility and expressiveness of rule graphs and how they are incorporated in OSAM*/P's OO model and language framework.

Rule Associations: In OSAM*/P, rules are modeled as objects. The control relationships among rule objects are modeled by a number of rule (or control) associations. This is analogous to modeling semantic relationships among entity classes using class associations (such as generalization and aggregation). The syntax and semantics of the rule associations are given below.

Rule association S: An "S" association defines a sequential execution order between two rules. The syntax for S is as follows:

```
rule r1:
    S: r2; /* r1 precedes r2 */
```

Rule association P: A "P" association is defined between a single rule and a set of rules to specify that the single rule should precede all the rules in the set during their execution and the rules in the set can be executed in parallel. The syntax for P is as follows:

```
rule r1:
    P: r2, r3, r4; /* r1 precedes r2, r3 and r4 */
```

Rule association Y: A "Y" association is defined between a set of rules and a single rule to specify that the rules in the set should precede the single rule during their execution. The syntax for Y is as follows:

```
rule r4:
    Y: r1, r2, r3; /* r1 r2, r3 precede r4 */
```

Rule Graphs: A rule graph is a network of CA rules and their control associations. An example rule graph is shown in Figure 4. The semantics is as follows. First, r1 must be executed. Then, the label P at r1 indicates that r2, r3 and r4 can be executed in parallel. The label S at r2 indicates that r5 has to sequentially follow r2. And, the label Y at r6 indicates that r3 and r4 have to synchronize at (or precede) r6.

In our approach, the event part is separated from the CA parts of an ECA rule and is associated with a rule graph instead of individual rules. Different events can be associated with different rule graphs. This provides rule designers with the flexibility to allow the same set of rules to follow different control structures when triggered by different events. For example, CA rules r1-r6 given

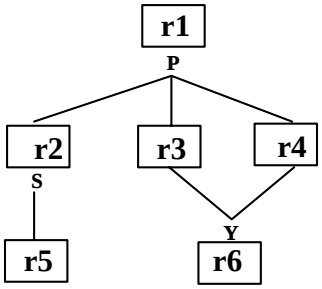

Figure 4: An Example Rule Graph

in Figure 1 can be specified to follow two different control structures (shown in Figures 1.a and 1.b) when they are triggered by two different events, namely, "Engine overheating" and "General diagnosis", by simply defining two different rule graphs over the same set of CA rules.

Trigger Times: In addition to the event part, the trigger time is also associated with the rule graph. It specifies when the triggered rule graph must be executed relative to the triggering operation. Our system supports the following four trigger times to provide a rule designer with the flexibility to specify four different execution options for a triggered rule graph.

- *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 integrity constraints by following a specific control structure, immediately after an update operation.
- *After*: The associated rule graph is executed after the transaction execution, but before the commit time. This option is used to delay a set of interrelated tasks defined by rules until 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 useful to execute some rule graphs as separate transaction graphs in parallel to the triggering transaction graph.

Syntax of a Rule Graph: The general syntax for the rule graph shown below. Strings enclosed by $'<'$ and $'>'$ are identifiers, strings enclosed by quotes are expressions, and the others are keywords used in the rule graph specification.

```
rule_graph     <rule graph name>  is
triggered      'trigger_conditions'
<rule name>:
       S: <rule name>;
       P: <rule name>, <rule name> .... ;
       Y: <rule name>, <rule name> .... ;
<rule name>:
       S: <rule name>;
       P: <rule name>, <rule name> .... ;
       Y: <rule name>, <rule name> .... ;
 :
 :
end            <rule graph name>;
```

Every rule graph must have a name and some `trigger_conditions`. The rule_graph body consists of a list of rules and their associations, if any. Each rule can have S, P or Y association(s) with other rules. All associations are optional. The keyword "triggered" specifies the trigger conditions. Trigger conditions are specified as follows:

```
trigger_conditions: trigger_condition | trigger_conditions, trigger_condition
trigger_condition:  trigger_time trigger_operation
trigger_time:       Before | Immediately-after | After | Parallel
```

A trigger operation can be a DB operation or a user-defined method. For example, the rule graph shown in Figure 4 can be defined as follows:

```
rule_graph  RG1  is
triggered   Before Update Employee.BasicSalary
     r1:
       P: r2, r3, r4;
     r2:
       S: r5;
     r6:
       Y: r3, r4;
end  RG1;
```

Syntax of a Rule: As mentioned earlier, a rule graph specification contains only rule names and the actual rules are specified in the RULES section of the class definition. The general syntax for a rule is as follows:

```
rule          <rule name>  is
condition     'rule_condition'
action        'statements'
otherwise     'statements'
end           <rule name>;
```

Every rule must have a rule name. The condition clause is optional. The action and otherwise clauses are also optional, but one of them must exist. The condition clause specifies a state of the database using a condition expression which evaluates to true or false. If the condition evaluates to true, then the operation(s) in the action clause is executed. If the condition evaluates to false, the operation(s) in the otherwise clause is executed.

OSAM*/P Class Definition: Rules and rule graphs are specified in RULES and RULE GRAPHS sections of a class definition, respectively. Additionally, an OSAM*/P class definition has some more sections, namely, ASSOCIATIONS, METHODS and IMPLEMENTATIONS. In OSAM*/P, the class definition is the main building block of an application schema. An application schema is defined by a set of class definitions. An example OSAM*/P class definition is given in Figure 5.

It should be noted that, by using a rule graph, any complex execution flow of rules can be easily specified, and the interrule parallelism and synchronization points can be explicitly defined. In addition, a CA rule has the flexibility to participate in multiple rule graphs as it can be triggered by multiple trigger operations. In a class definition, the specification of rule control is clearly separated from the specification of rules. This makes the complex control semantics of rules easy to understand and modify.

4 Graph-based Transaction Model

In DB environments, all operations are executed in a transaction framework. Since DB operations of a transaction may trigger rule graphs, and rules in a rule graph may generate additional DB operations which may in turn trigger other rule graphs, the execution of rule graphs must be incorporated into a transaction framework.

```
/* 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 5: An Example Class Definition

4.1 Graph-based Transactions

To model control structures of triggered rule graphs uniformly, it is logical to view a transaction as a control graph of subtransactions (each having its own control sphere) as shown in Figure 6.a. In this figure, boxes represent control spheres and the directed edges represent the control flow. This can be viewed as an extended tree control structure in which siblings can form a Directed Acyclic Graph (DAG) depicting the (partial) order in which they should be executed. The extended tree structure is shown in Figure 6.b, in which solid lines represent parent-child relationships and hyphenated lines represent the control structure among subtransactions. Each subtransaction in turn may have a graph structure, which means that the tree can expand dynamically as the nesting depth of the triggered rule graphs increases. We shall call a transaction with the graph-structure a *transaction graph* (TG).

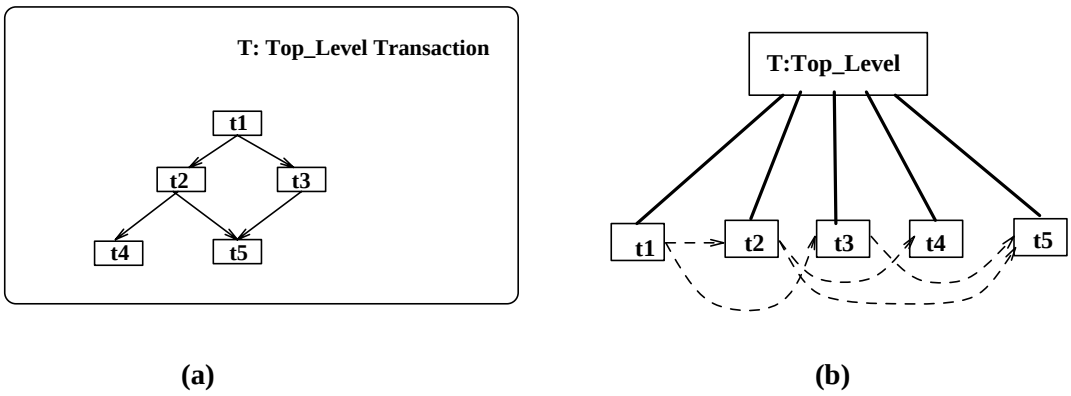

Figure 6: Transaction Graph Model

The structure of a transaction graph is expressive enough to *uniformly* model control structures of rule graphs. For example, if a top-level transaction T1 triggers the rule graph shown in Figure 7.a, rules in the rule graph are modeled as subtransactions of T1, and the control structure of the rule graph is modeled as the control relationships among subtransactions. The resultant transaction graph is shown in Figure 7.b. Note that, the rule graph shown in Figure 7.a is translated into an equivalent directed acyclic graph (shown in hyphenated lines) and added to the transaction graph. The graph-based model also captures the nested triggering of rule graphs. The rules in a triggered rule graph are modeled as subtransactions of the triggering rule. For example, if an operation activated by rule r2 (of the TG shown in Figure 7.b) triggers the rule graph shown in Figure 8.a, the rule graph is incorporated into the transaction graph structure as shown in Figure 8.b.

Another important feature of the transaction graph (TG) is its ability to support the control semantics of different trigger times (or coupling modes). During execution, a triggered rule graph is added to the triggering transaction at an appropriate place according to its trigger time. To

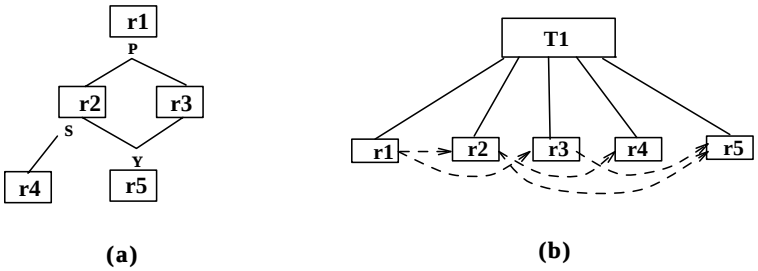

Figure 7: Modeling a Triggered Rule Graph

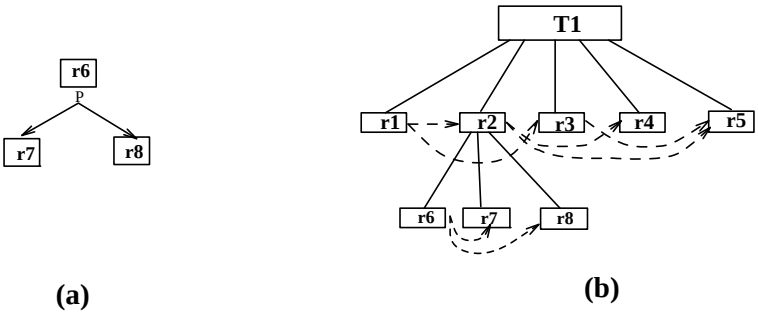

Figure 8: Modeling Nested Triggerings of Rules

include the rules with trigger time *after* at the end of the TG, the TG should have "begin" and "end" points. We add a *begin-task* which performs the initialization in the beginning of the control flow and a *commit-task* which commits the transaction atomically, at the end of the control flow, as shown in Figure 9.a. This can be represented using the tree form shown in Figure 9.b. A triggered rule graph is added to the triggering transaction graph 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 (or rule). For example, if an operation of t2 of the TG shown in Figure 9.b triggers the rule graph in Figure 10.c, the rule graph is added to the TG as shown in Figure 10.a. Modeling rules as subtransactions of the triggering task enables arbitrary nesting of rules. The isolation property of the rules being executed in parallel is maintained by a locking scheme, and the control structure among rules is maintained by a scheduler. The scheduler starts scheduling the rules before or immediately after performing the triggering operation, depending on the trigger time. Nevertheless, rules are executed as subtransactions of the triggering tasks in both cases.

In the case of *after* as the trigger time, the triggered rule graph is added just before the commit task. In the above example, if the trigger time is *after*, the rule graph (in Figure 10.c) is added to the triggering TG just before the commit task as shown in Figure 10.b, and the triggered rules

are treated as subtransactions of the top-level transaction. In addition, the *triggering order* of the *after* rule graphs is also captured. Assume that the task t2 triggers another rule graph with the trigger time "after", the rule graph is added just before the commit task but after the rule graph which was triggered earlier. In this way, the "after" rule graphs will be executed in the order they were triggered. The importance of maintaining such order is stressed in [24].

In the case of *parallel* as the trigger time, the triggered rule graph is executed as a separate transaction in parallel with the triggering transaction. It is completely detached from the triggering transaction/rule in all respects except a causal relationship. A parallel transaction can be causally dependent or causally independent of the parent transaction. In the first case, the failure of the parallel transaction causes the triggering transaction to abort and, in the second case, the failure does not affect the triggering transaction. However, the failure of a triggering transaction causes all triggered transactions, including parallel transactions, to be aborted.

It can be observed that, for all trigger times, the rule graph structure uniformly fits into the proposed TG structure, which obviates the transaction manager's need to treat rules differently from other subtransactions with respect to scheduling and concurrency control techniques and the correctness criterion. A transaction graph is expanded as the number of triggered rule graphs are added at different levels of the graph.

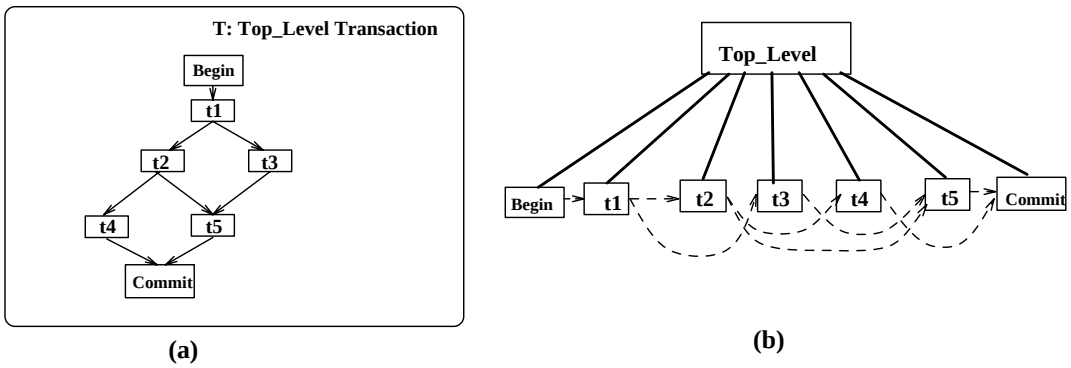

Figure 9: Adding begin and end points to a TG

4.2 Concurrency Control

In a TG model, rules (or subtransactions) should maintain atomicity and isolation. Additionally, they need to maintain the partial order defined by the control structure. A random execution order (although serial) can lead to semantically incorrect results. In this section, we define a correctness criterion for the concurrent execution of rules within a TG and describe a scheduling algorithm to enforce the order imposed by the control structure.

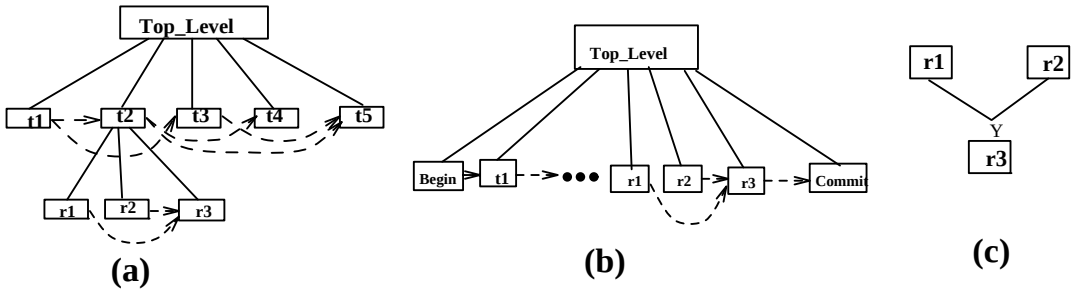

Figure 10: Coupling a Rule Graph to TG

4.2.1 Correctness Criterion

In our transaction model, each task (transaction or rule) is executed in its own control sphere. Transactions are executed in different transaction-level control spheres. Each transaction-level control sphere may generate a set of rule-level control spheres when a set of rules is triggered for execution within the transaction control sphere. Each rule-level control sphere may in turn generate a set of rule-level control spheres when an operation of the rule triggers another set of rules.

At the transaction-level, the correctness criterion for a parallel execution of several TGs is standard serializability [17] which states that the interleaved execution of several TGs is correct when it is equivalent to some serial execution. This means that all transaction-level control spheres are clearly isolated from one another. Within a TG, the control structure among subtransactions and triggered rule graphs plays a role in defining correctness. The serializable execution of rules alone is not sufficient because an arbitrary serial order can violate the control structure among rules. For example, for the rule graph shown in Figure 7.a, the order $r1 \rightarrow r4 \rightarrow r2 \rightarrow r3 \rightarrow r5$ is not correct. The parallel/concurrent execution of rules is *correct* if and only if the resulting order does not violate the control structure of the rule graph. The correct execution orders for the rule graph shown in Figure 7.a are $(r1 \rightarrow r2 \rightarrow r3 \rightarrow r4 \rightarrow r5)$ and $(r1 \rightarrow r3 \rightarrow r2 \rightarrow r4 \rightarrow r5)$. To be more general, the parallel/concurrent execution of rules is correct when the resultant execution order is equivalent to some *topological order* [12]. Since the above correctness criterion requires a topological serial order instead of an arbitrary serial order, we shall call it *topological serializability*. To summarize, the requirements of the topological serializability are: (i) the execution of a rule must be atomic, (ii) the execution of a rule must be isolated from the execution of other rules in the system, and (iii) the execution of rules in a rule graph must be equivalent to a sequential execution of the rules in the topological order specified by the rule graph. The same correctness criterion applies to both rules in a rule graph and tasks in a transaction graph.

4.2.2 Topological Scheduling

To maintain the topological serializability during the execution of a rule graph, it is sufficient to execute rules one after another in a topological order, even at the expense of losing the potential parallelism among independent rules. For example, rules r2, r3 in the rule graph shown in Figure 7.a can be executed in parallel without violating the topological order because they are independent of each other. In general, a rule in a rule graph is independent if it does not have any incoming edge, or its *indegree* (number of incoming edges) is equal to zero. To exploit parallelism, rules in a rule graph can be divided into *topological groups* such that the rules in each group can be executed in parallel. For example, the topological groups for the rule graph shown in Figure 7.a are (r1), (r2, r3) and (r4, r5). Within each topological group, the rules can be executed in parallel. 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.

Repeat

Select: This step selects the set of independent rules which is unique for a given rule graph. Independent rules are identified by their indegree.

Schedule: This step assigns the independent rules to appropriate processing nodes for parallel execution[1]. The isolation among rules is maintained by a locking scheme which will be explained later in Section 4.2.3.

Wait: This step synchronizes all the rules scheduled in one iteration by waiting for their completion. In the actual implementation, which will be explained later in Section 5, the transaction manager switches to another transaction during the `wait` step, thereby interleaving the execution of multiple transactions.

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

Until the rule graph is completely executed;

The following theorem proves that the above algorithm schedules the rules of a given rule graph in a topological order specified by its control structure.

Theorem 1: *The above algorithm schedules the rules of a given rule graph without violating the control structure.*

Proof: *We prove the theorem by contradiction. Assume that there is a scheduled rule r which violates the control structure. This implies that it is scheduled before the completion of r's predecessor say r1. According to the predecessor definition, there is a directed path from r1 to r in the rule graph. This implies that there is at least one incoming edge to r when it is scheduled. This*

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

means r's indegree is not zero. This contradicts with the hypothesis that the algorithm schedules only independent rules. Hence the proof. Note that the scheduler waits until the completion of all scheduled rules and in the remove-step, it removes the completed rules and their outgoing edges from the rule graph.

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 7.a. It is scheduled in the following sequence of groups $\{r1 \rightarrow (r2, r3) \rightarrow (r4, r5)\}$ and note that the rule r4 waits until the completion of r2 and r3 in spite of the fact that it is eligible for execution as soon as r2 is completed. To avoid the unnecessary waiting of rules, the above algorithm is modified. The modified algorithm does not wait until the completion of all scheduled rules. Instead, after the completion of each scheduled rule, the algorithm schedules the rules which subsequently become independent (if any). The completed rules place their acknowledgments in a queue asynchronously. The scheduler waits only when the queue is empty, otherwise 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 of the scheduler and thereby increases concurrency among rules.

4.2.3 Locking Scheme

Although the above scheduling algorithms enforce the control structure, they do not take care of the isolation property of the rules that are being executed in parallel, e.g., the parallel execution of two independent rules need not be serializable. The following locking rules are used for maintaining rule isolation as well as transaction (TG) isolation.

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, triggering transaction/rule becomes parent and an ancestor is any transaction/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.*

The following theorem proves that the proposed concurrency control method which is comprised of the locking scheme and the topological scheduling algorithm is correct.

Theorem 2: *The above locking scheme and the scheduling algorithm satisfy the following requirements for correctness: i) the parallel execution of rules in a triggered rule graph is equivalent to some topological order, and ii) the interleaved execution of transactions is equivalent to some serial order of execution.*

Proof: *It is proved in Theorem 1 that the proposed topological scheduling algorithm maintains the topological order for a given rule graph. But, according to the scheduling algorithm, there can be several rules being executed in parallel. For example, when a scheduler identifies several independent rules, all of them are scheduled for the execution at the same time. In the asynchronous approach, there can be some new rules being scheduled when there are rules being executed still. The isolation among all these rules is maintained by the above locking scheme. The first locking rule does not allow two rules to share the locks in a conflicting mode, as long as they do not have an ancestor-descendant relationship. The fact that no two rules in a rule graph have an ancestor-descendant relationship (not to be confused with predecessor-successor relationship) ensures that co-executing rules are isolated. The isolation among top_level transactions is maintained because the rule locks are inherited by the top-level transaction and they are released only when it commits or aborts so that the other top-level transactions cannot see the partial results until its completion.*

The recovery of the graph-based transactions can be done using the standard recovery methods used for NTM [32, 37]. The only difference is that when a set of rules are being undone, they need to follow *reverse* topological order. A detailed discussion of recovery techniques is out of the scope of this paper.

5 Implementation on a Shared-nothing Computer

The OSAM*.KBMS/P, a parallel active OOKBMS server, has been implemented on an nCUBE2 computer [10, 28, 33]. The parallel computer is suitable for parallel knowledge base servers because of its shared-nothing (distributed) architecture, scalability, availability and high performance [41]. The main objectives of our implementation effort were to demonstrate the implementability of the presented transaction and rule modeling scheme, the knowledge definition language, the topological scheduling scheme and its corresponding locking scheme, the parallel rule and data processing algorithms and to gather performance data from the implemented system. The main functionality of the server is to receive transactions from several clients, process them asynchronously and return

results to the clients. The architecture of the server is shown in Figure 11. Multiple clients C1,C2,...Cn which are connected to the server by an interconnection network (Ethernet), reside on different Sun workstations. Each client contains an X-Motif graphical user interface which has advanced features for editing TGs, rule graphs, queries and application schemas.

The OSAM*/P knowledge base, which is defined by a set of interrelated classes, is partitioned class-wise, i.e., the instances of each class are stored in the disk of a processing node of nCUBE2. Multiple classes can be assigned to a single processing node. A global data dictionary which contains the overall schema information and the mapping between classes and the processing nodes is maintained. A local data dictionary, which contains the structure of local classes, is stored along with the classes in a node. It should be noted that rules are also distributed among the classes.

As shown in Figure 11, the server has a global transaction server (GTS) and a cluster of local transaction managers (LTMs). The GTS receives transaction graphs from different clients and supervises their asynchronous executions. The GTS uses the message passing mechanism provided by `nread` and `nwrite` commands [34] to communicate with LTMs. Each LTM receives tasks from the GTS and processes them in an interleaved fashion.

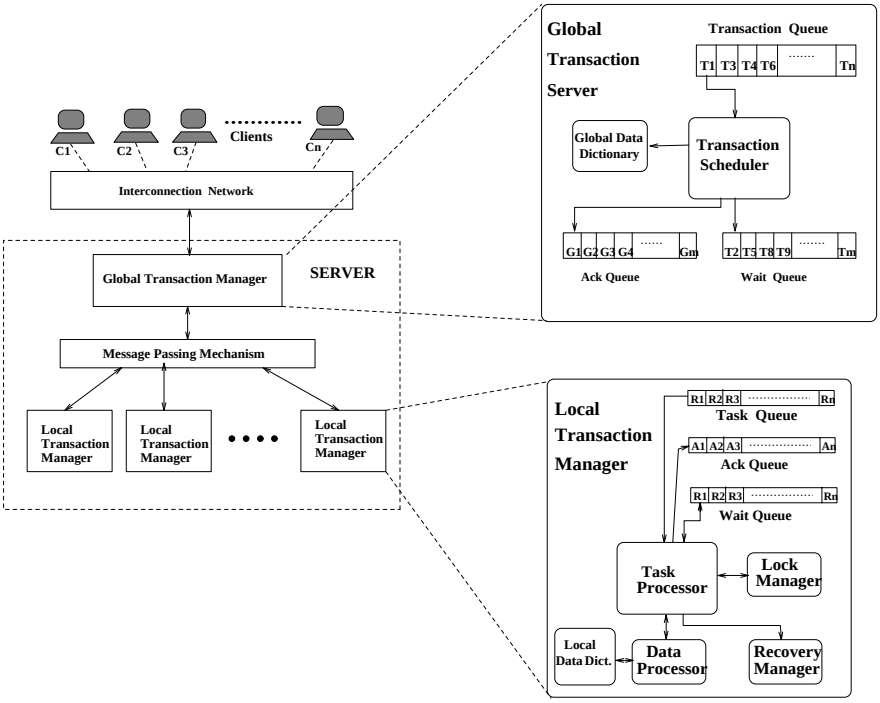

Figure 11: System architecture of OSAM*.KBMS/P

5.1 Global Transaction Server

The components of GTS are shown in Figure 11. GTS consists of a transaction queue (TQ), a transaction scheduler (scheduler), a wait queue (WQ) and an acknowledgment queue (AQ). In Figure 11, T1,T2 .. are transaction graphs (TGs) and G1,G2 .. are acknowledgments from LTMs. GTS keeps the incoming TGs in the rear of TQ. TQ acts as a buffer to offset the mismatch between the rate at which TGs are arriving from clients and the rate at which TGs are being scheduled by the scheduler. The scheduler runs the asynchronous scheduling algorithm explained in Section 4.2.2. It gets a TG from the head of TQ, selects the independent tasks, distributes them to the underlying LTMs, and keeps the remaining tasks in WQ. For assigning a task to an appropriate LTM, GTS identifies the classes that are used by the task and assigns the task to the LTM that handles the classes. In case if there are multiple classes handled by multiple LTMs, a load balancing strategy is used to select the appropriate LTMs. WQ consists of TGs that are partially processed and waiting for further processing. When a TG is in WQ, the scheduler gets another TG from TQ and schedules it. The scheduler resumes the processing of a suspended TG, when it sees an acknowledgment received from an LTM indicating that a task of the TG is completed/aborted. A TG's execution is completed when the scheduler reaches the commit node. GTS requests all relevant LTM's to commit the transaction using the 2 phase commit protocol [35]. The committed TG is removed from WQ.

It can be observed that TGs are being interleaved by the scheduler and also several tasks are being processed in parallel by the underlying LTMs. Since the `nread` command (for reading messages) provided by nCX (nCUBE2 OS) is blocking, the scheduler uses the `ntest` command to check whether there are any new messages. In the same way, all the other modules in the system also use queues and the `ntest` command to process assigned tasks asynchronously.

5.2 Local Transaction Manager

GTS puts a task in the LTM's task queue for execution. It is the LTM's responsibility to execute the tasks efficiently and consistently. In addition, an LTM also passes the rule graphs triggered by tasks' operations to GTS for their scheduling. As shown in Figure 11, an LTM consists of a task processor, a parallel data processor, a lock manager (LM), a recovery manager, a task queue, a wait queue (WQ) and an acknowledgment queue (AQ).

When a task is being processed, an operation of this task can trigger a rule graph. Since a rule graph has the same control structure as a TG, the task processor utilizes the GTS's scheduler for enforcing the control structure among rules. The trigger time associated with each rule graph is maintained by placing it at an appropriate control point of the top_level transaction. The task

processor handles triggered rule graphs, depending on their trigger times, as follows: i) In the case of *before*, the rule graph is kept in WQ of GTS, and an acknowledgment which includes the rule_graph_id is kept in GTS's AQ. The task is suspended and kept in the local WQ. GTS schedules the rule graph and sends an acknowledgment to the LTM when the rule graph completes its execution. The task processor resumes the task as soon as it sees the acknowledgment in its AQ. ii) In case of *immediately-after*, the operation is performed and the task is kept in the local WQ. The task processor then follows the same steps as (i). iii) In the case of *after*, the rule graph is kept in GTS's WQ and the task processor resumes the processing of the task. GTS appends the rule graph to the TG just before the commit task as shown in Figure 10.b. iv) In the case of *parallel*, the triggered rule graph is kept in the transaction queue. The rule graph is scheduled like a separate TG.

The LTM processes tasks and the rules in an asynchronous fashion. When the task processor has to suspend the processing of a task, it keeps this task in WQ and mark the control point to resume processing later. The task processor calls the underlying parallel data processor for performing data processing operations. It acquires the locks for the data being accessed before sending the operations to the data processor. A local Lock Manager (LM) receives the requests for locks from the local task processor and sometime from the remote task processors. The local LM is responsible for the locks on the entity class that is stored in that particular node and follows the locking rules explained in Section 4.2.3. The detailed algorithm for the LTM is given in the appendix.

5.3 Parallel Data Processor

As shown in Figure 11, each LTM has a parallel data processor for processing operations on the local database. When a DB operation involves remote data, e.g. the operation is to form a subdatabase which involves multiple classes (analogous to joining multiple relations), the data processor uses asynchronous multiple wavefront algorithms [43, 46] to form the subdatabase in parallel.

In an OO view, a database is viewed as a *schema graph* intensionally and an *object graph* extensionally. In a schema graph, nodes represent classes and edges represent the associations among the classes. In an object graph, nodes represent objects and edges represent the connections between the objects. For example, the object graph shown in the Figure 12.a depicts objects in the classes (RA, Grad etc.) and their interconnections. The subgraph corresponding to the subdatabase of interest is specified by a Context expression [2], e.g., the Context `RA * Grad * Student * AND (Section,Department)` means: Identify all object instances of RA, Grad, Student, Section and

[2] See [2] for full details on the syntax and semantics of the Context expression.

Department which satisfy the query pattern given in Figure 12.c, i.e., the sections and departments of graduate students serving as RAs. Following the Context specification, a query would specify the system-defined or user-defined operations which are to be performed on the object instances in the subdatabase. The resulting subdatabase for the above context expression is shown in Figure 12.b.

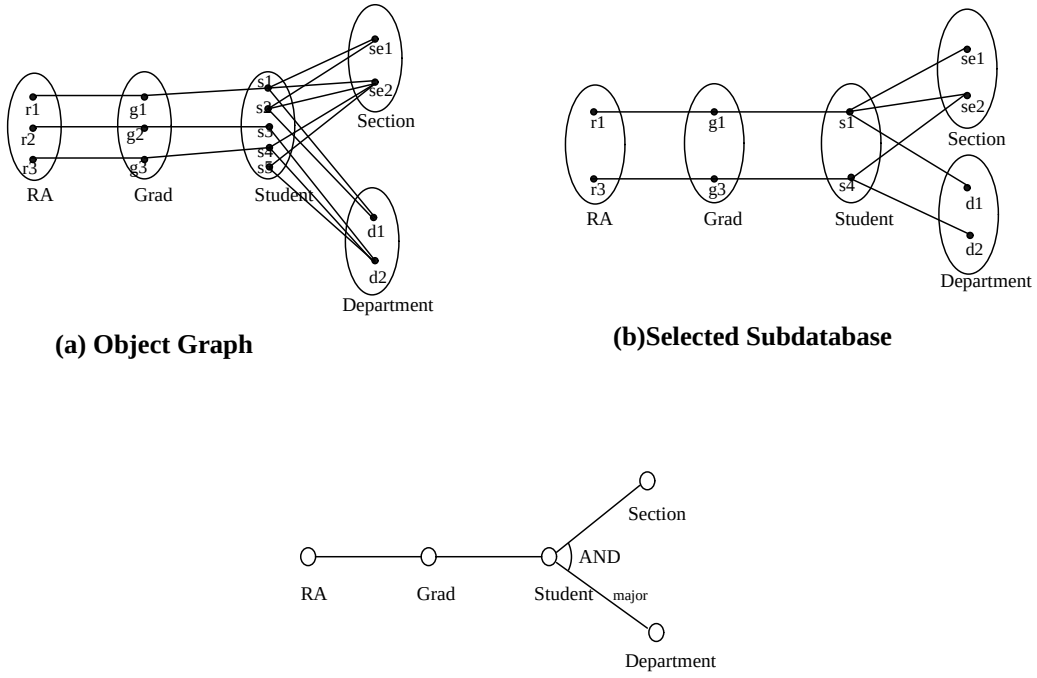

Figure 12: Graph-representation of OODB operations

In OSAM*.KBMS/P, two parallel multiple wavefront algorithms, namely, identification [46] and elimination [43] algorithms, are used for establishing a subdatabase specified by a Context expression. Since the papers describing these two algorithms have been published, we shall only briefly describe them here to give the reader a general idea of their functionalities. Object classes are assigned to a number of data processors which manage the instances of these classes. The instances of each class, stored in a data processor, are connected to the instances of the neighboring classes of an object graph. All connections using instance identifier (IID) references are bi-directional. Given a query, the query graph which represents the Context expression of the query is sent to all the data processors that manage the involved classes. The identification algorithm starts from all the terminal nodes (i.e., RA, Section and Department in the above example) which simultaneously propagate the IIDs of the neighboring classes' instances, which are associated with their own object instances, to their neighbors. The propagation of each terminal node forms a wavefront, thus multiple wavefronts of IIDs go across one another. Each data processor receiving IIDs looks up its

local tables to identify the IIDs of some other neighboring nodes that are connected with the IIDs just received, and propagates the IIDs to its neighbors. Set intersection or set union operations are performed on the IIDs of multiple waves depending on AND or OR branching operator used in the query graph. When all the terminal nodes have received and processed all the waves of IIDs, object instances which satisfy the query pattern have been marked, the subdatabase has been established for further processing. The algorithm then terminates.

The elimination algorithm also uses the multiwavefront strategy except that wavefront propagations start from all the nodes (instead of only terminal nodes). Each node eliminates instances that do not satisfy the query pattern instead of identifying those that do. The remaining instances form the final subdatabase.

Each data processor is capable of running both identification and elimination algorithms which perform differently under different data and query conditions. Data processors select the appropriate algorithm based on these conditions. Both algorithms use a general data structure which stores the object graph information in such a way that it can be efficiently used by both algorithms. The connections between the object instances belonging to two classes are represented by an adjacency matrix. Rows and columns of the matrix are stored as adjacency lists of IIDs in the data processors that manage those two classes. The adjacency list for each instance contains IIDs of the connected objects in the associated class. For supporting both algorithms, a set of adjacency lists are stored in a data structure that uses *backward pointers*. Instead of storing connected IIDs of each instance in the class, we store the list of local IIDs that are connected to each instance of a neighbouring class. This data structure is useful for both identifying the connected objects and eliminating the objects that are not connected.

Each data processor is implemented as a finite state automata controller (FSAC), which enables the interleaved processing of multiple queries concurrently. When a query sends its wavefront of IIDs to a neighboring class and waits for other wavefronts of IIDs to come to it, the data processor suspends the query by saving its state and switches to another (new or old) query. It moves from one state to another depending on the message received. For example, FSAC resumes a suspended query (by loading the query's state) only when it receives the message the query is waiting for, e.g., a wavefront from a neighbouring class. Each message has message_type, query_id, destination and source information. Message_type indicates whether the query is new or partially processed. If it is new, a fresh query block which contains information about that query is created, otherwise, the current state of the suspended query is loaded from the query block and the processing is resumed. The query_id is used to identify the query. The query blocks are hashed on the query_id. Query information contains the message contents, e.g., a set of IIDs. FSAC selects a method to perform

the appropriate action from a FSA table in which, for a given message type and FSA state, the appropriate method to be invoked is stored.

5.4 Lock Manager & Recovery Manager

In OSAM*.KBMS/P, consistency and correctness of parallel/concurrent executions of tasks, transactions, and rules, are maintained using a distributed Lock Manager (LM). Each LTM residing at a node contains a local LM which is responsible for granting/rejecting locks on the data at that particular node. Each local LM is comprised of a Lock Communication Manager (LCM) and a Lock Logic Manager (LLM). LCM is responsible for dispatching the lock requests to the appropriate LLM depending on the location of the data item for which lock is requested. LLM is responsible for processing lock requests using the locking rules given in Section 4.2.3. The distributed implementation of the lock manager is scalable and prevents the lock manager from becoming a bottleneck.

Our approach for recovery is as follows. Since our transaction is decomposed into several tasks and task isolation is maintained, a finer control over recovery is possible. When a task is aborted, it can be recovered without affecting other tasks or transactions. Our recovery is based on a write ahead logging scheme and an UNDO/REDO strategy which is more efficient than the version approach. Similar recovery techniques have been used for nested transactions in [32, 37]. However, the recovery of a graph-based transaction is different from that of a nested transaction in that the tasks (or subtransactions) need to be undone in reverse of the order in which they were scheduled (reverse topological order), to maintain semantic consistency. The above approach is efficient because several recovery managers (at different nodes) can undo tasks in parallel. In the current system, the recovery manager is the only module that has not been implemented.

6 Performance Evaluation

There are two performance metrics in use for parallel systems. The first metric, called *speedup*, measures the ability of the system to grow in order to reduce the execution time of a fixed number of tasks. Speedup assesses how effective a system is in allocating its resources. The second metric, called *scaleup*, measures the ability of a system to be scaled up to a larger system. In quantitative terms, an n-times larger system is expected to perform an n-times larger job in the same amount of time that it took the original (or smaller) system to process the smaller task. In an ideal system, as the size of the system is increased, the speedup should increase while the scaleup remains constant.

For performance evaluation, we used a "Registration transaction" which has the graph structure shown in Figure 2, and the rule graph shown in Figure 7.a. The latter is triggered by the

task "Calculate Debt". We ran a large number of Registration transactions on the server with one global unit (a GTS 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 formula used for calculating the speedup is given below.

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

It can be observed from the curve shown in Figure 13 that the increase in speedup is almost linear up to 7 local units and gradually tapers 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 interprocessor communication. However, it can be observed from the speedup curve that the speedup can be increased further, although not linearly, by increasing the number of local units. The optimal number of local units is 7 to achieve maximum speedup for 640 transactions.

We also evaluated the *scaleup* of the system by running 40 Registration transactions on the system with one global unit and one local unit, and increasing 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 calculating the scaleup is given below.

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

It can be observed from the scaleup curve in Figure 14 that the system scaled up well because most of the computation-intensive functionalities are distributed among LTMs instead of being centralized at a GTS. The scaleup is almost linear up to 480 transactions. At this point, GTS becomes a bottleneck serving multiple LTMs on one side and several clients on the other side. The scaleup therefore gradually tapers as the number of transactions and the number of local units increase further. The scaleup can be further improved by launching multiple GTSs to receive incoming transactions in parallel and schedule them to the underlying LTMs.

We also analyzed the execution time of the rule graph in Figure 15 with the increase in the number of local units (LUs). The variation of execution time as the number of LUs increases is shown in Figure 16. Up to 3 nodes, the execution time decreases almost linearly, then it gets saturated; in fact it slowly increases. This is because, according to the control structure of the rule graph, at most three rules can be executed in parallel except in the last phase. Therefore, only three LUs can be utilized and any additional LU creates overhead. This is the reason why the execution time increases as the number of processors exceeds 3. When 4 processors are allocated,

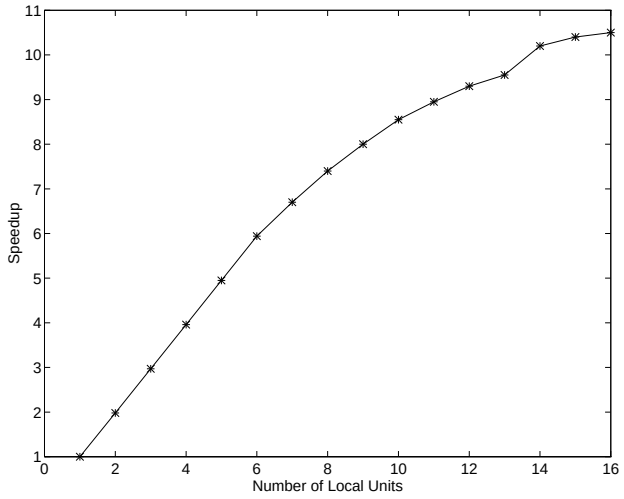

Figure 13: Speedup of the Transaction Server

at least one processor will be idle until the last phase, then R9, R10, R11, R12 can be executed in parallel. Performance results show that the overhead generated by the 4th processor is more than the speedup achieved.

7 Conclusions

A parallel, active and object-oriented knowledge base server is needed for meeting the functional and performance requirements of many advanced and complex database applications. We believe that such a server must have a flexible and expressive rule control mechanism for capturing complex control structures among rules in these applications. In this paper, we have presented the rule control mechanism of OSAM*.KBMS/P - a parallel active object-oriented knowledge base server. We have proposed the use of rule graphs to explicitly specify complex control structures among rules. The rule graph representation provides the needed flexibility, expressiveness and comprehensibility for defining and understanding the complex structural relationships among rules. We have extended an existing OO knowledge representation model (OSAM*) with modeling and language constructs for supporting the modeling and specification of rule graphs in an application schema. We have designed a comprehensive execution model using a graph-based transaction model to uniformly incorporate the execution of rules in rule graphs. Additionally, the proposed execution model also captures the different execution options of the triggered rule graphs. Besides its expressiveness and flexibility, the proposed graph-based representation can also be used to explicitly specify the parallelism among rules and database tasks. To exploit the parallel execution properties of our graph-representation, we have implemented the proposed OO knowledge representation model and

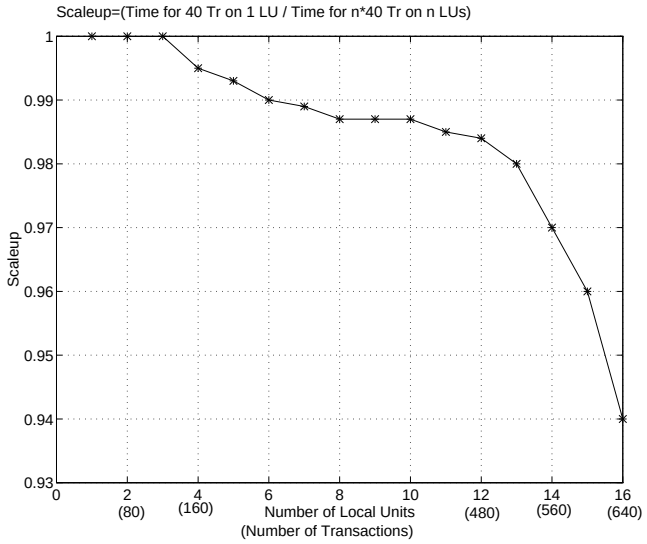

Figure 14: Scaleup of the Transaction Server

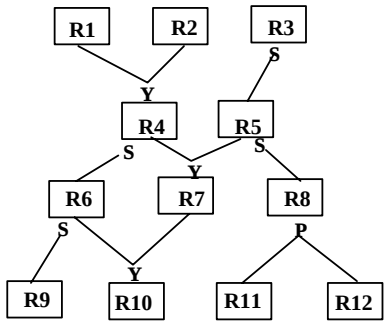

Figure 15: A Sample Rule Graph

graph-based execution model on an nCUBE2 computer. Our implementation uses a scalable software architecture and an asynchronous execution model. We have evaluated the speedup and the scaleup of the implemented system with a large number of transaction graphs and rule graphs. The system has achieved good speedups because of the parallel execution properties of rules and tasks, and their asynchronous executions. The system scales up well because of the scalability of both hardware and software architectures.

The proposed rule control mechanism, OO modeling and language constructs, graph-based transaction modeling and execution techniques, and the parallel architecture and implementation techniques, can be used as independent mechanisms or as a combination for the implementation of other advanced DB functions. For example, the proposed graph-based transaction model and the concurrency control methods can be used in a multi-DB environment to capture intertask and intertransaction dependencies, and the proposed OO extensions can be used to extend an OO data model with rules and rule control relationships.

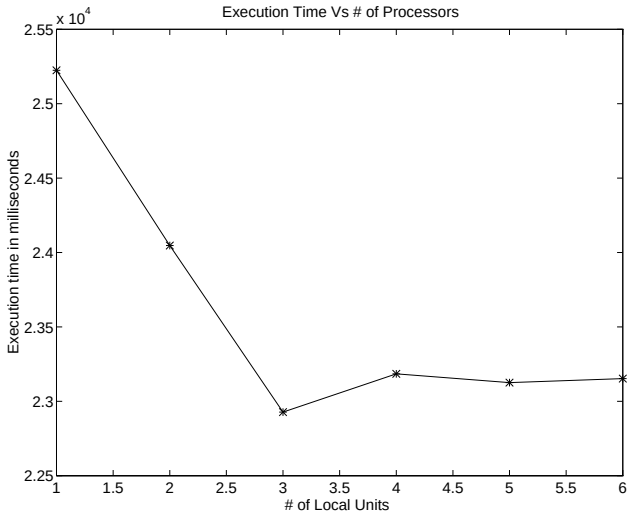

Figure 16: Rule Graph Execution

Our future work involves the implementation of the recovery subsystem and the extension of the system to support high-level complex event specifications.

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] A. M. Alashqur, S.Y.W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proc. 15th Int'l Conf. on Very Large Data Bases*, pages 433–442, Amsterdam, The Netherlands, August 1989.

[3] 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.

[4] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H. J. Kim. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[5] C. Beeri. Formal models for Object Oriented Database Systems. In W. Kim, J-M. Nicolas, and S. Nishio, editors, *First Intl. Conf. on Deductive and Object Oriented Databases*, pages 370–395, 1989.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), June 1992.

[15] 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.

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

[17] 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.

[18] S. Gatziu and K. R. Dittrich. SAMOS: an active, object-oriented database system. *in IEEE Quarterly Bulletin on Data Engineering*, 15(1):23–26, December 1992.

[19] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona (Catalonia, Spain), Sep. 1991.

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

[21] 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.

[22] Eric N. Hanson and Jennifer Widom. Rule processing in active database systems. *International Journal of Expert Systems*, 6(1):83–119, 1993.

[23] S. Heiler, S. Haradhvala, S. Zdonik, B. Blaustein, and A. Rosenthal. A flexible framework for transaction management in engineering environments. In A. Elmagarmid, editor, *Database transaction models for advanced applications*, pages 87–122. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[24] 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.

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

[26] G. Kiernan, C. deMaindreville, and E. Simon. Making deductive database a practical technology: a step forward. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, 1990.

[27] 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.

[28] 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.

[29] D. R. McCarthy and U. Dayal. The architecture of an active, object-oriented database system. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 142–151, Portland, Oregon, June 1989.

[30] M. Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In *Proceedings 9th International Conference on Very Large Data Bases*, pages 34–42, 1983.

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

[32] 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.

[33] 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.

[34] nCUBE. *nCUBE 2 Programmer's Guide*, 1992. nCUBE Corporation, Foster City, CA.

[35] M.T. Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

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

[43] S. Y. W. Su, Y.-H. Chen, and H. Lam. Multiple wavefront algorithms for pattern-based processing of object-oriented databases. In *Proc. 1st Int'l Conf. on Parallel and Distributed Inf. Syst.*, pages 46–55, Miami, FL, December 1991.

[44] 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.

[45] S. Y.W. Su, H. Lam, S. Eddula J. Arroyo, N. Prasad, and R. Zhuang. OSAM*.KBMS: an object-oriented knowledge base management system for supporting advanced applications. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Washington, D.C., May 1993.

[46] A.K. Thakore, S.Y.W. Su, and H. Lam. Algorithms for asynchronous parallel processing of object-oriented databases. *to appear in IEEE Trans. Knowledge Data Eng.*, October 1994.

[47] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases*, 1(1):137–165, 1993.

[48] 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.

[49] 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.

[50] D. R. Zertuche and A. P. Buchmann. Execution models for active database systems: A comparison. Technical Report TM-0238-01-90-165, GTE Laboratories, Waltham, MA, January 1990.

Appendix

```
ALGORITHM GlobalTransactionServer
 LOOP
   IF  Ack_Queue is Empty THEN
      Get transaction T from the Transaction_Queue;
   ELSE
      Get the transaction T corresponding to the acknowledgment
      from the Wait_Queue;
   IF T is a rule graph and the trigger time is "after"
   THEN Attach the rule graph before the commit
        node of the parent transaction;
        Exit the loop;
   ELSE
      IF T is a partially processed transaction
      THEN Mark the completed task;

   Select the independent tasks of T;

   FOR each independent task DO
     Determine the appropriate LTM
     and assign the task.
   ENDFOR
 FOREVER;
END GlobalTransactionServer;
```

```
ALGORITHM LTM
 LOOP
   IF  Ack_Queue is Empty THEN
     Get Task R from the Task_Queue;
   ELSE
     Get the Task R corresponding to the acknowledgment
     from the Wait_Queue;
   ENDIF;
    LOOP
     Get the next operation for processing and
     Check if operation triggers any rules;
     IF a rule graph is triggered THEN
       CASE trigger_time OF
         before: Mark the operation and keep the
              task in wait queue and place the
              rule graph in the GTS wait queue
              and keep an acknowledgment in the GTS Ack
```

```
            queue; Exit;
        immediate after: Obtain locks, perform
            the operation, and keep the
            task in wait queue and place the
            rule graph in the GTS wait queue
            and keep an acknowledgment in the GTS Ack
            queue; Exit;
        after: Keep the triggered rule graph in
            the wait queue and keep an acknowledgment
            in the GTS Ack queue;
        parallel: Keep the triggered rule graph
            in the transaction queue;
      ELSE  Obtain locks for the operation and
          send the operation to the data processor;
    UNTIL end of the task;
    Send an acknowledgment to the GTS Ack queue to indicate that the
    task is completed;
  FOREVER;
END LTM;
```