A Graph-based Transaction Model for Active Databases and its Parallel Implementation

Ramamohanrao Jawadi
Stanley Y.W. Su
Database Systems Research and Development Center
Department of Computer and Information Sciences
University of Florida, Gainesville, FL 32611
rsj,su@cis.ufl.edu

Abstract

Databases coupled with active rules, which are known as *active databases* are becoming increasingly popular because of their added features that for supporting a wide spectrum of applications. The need for user-defined control over the rule execution order, particularly in parallel rule execution models, is well recognized in both AI and DB rule systems. In database rule systems, ideally, the rule execution should be uniformly integrated with the transaction execution framework. In active databases, little attention has been given to the provision of expressive control constructs for rule execution and their uniform integration with the transaction model. In this paper, we provide several control constructs which can be used to control the parallel execution of rules. These constructs are translated to a directed acyclic graph representing the control flow among the rules. To merge the graph-based rules with the database transactions uniformly, we introduce a graph-based transaction model that can naturally model complex structures among database operations and rules. While we follow the standard serializability as a correctness criterion for the concurrent execution of transactions, a new correctness criterion which involves the control flow among the rules, is introduced for concurrent execution of rules within a transaction. Two different concurrency control methods are presented for concurrent and correct execution of rules with user-defined control structures. In addition, we discuss the design and implementation of a parallel active database server which supports the proposed rule control and graph-based transaction model. The server has been implemented and tested on a shared-nothing multiprocessor architecture.

1 Introduction

Databases coupled with active rules, which are known as *active databases*, are becoming increasingly popular because of their added features for supporting a wide spectrum of

applications [27, 47, 48, 23, 8, 29, 50, 3, 14, 18, 24, 44]. In contrast to the traditional database systems, active systems monitor a variety of events that can occur and react to them automatically. In the majority of these systems, the active behavior is incorporated using production rules or simply rules. Although, there have been several efforts toward the development and implementation of sequential and centralized execution models for active database systems, little attention has been focused on the parallel execution model and implementation. This paper focuses on control and execution models of rules and transactions and their implementation in a shared-nothing parallel environments.

In shared-nothing parallel machines (e.g. Teradata, Tandem, nCUBE), since each node has a local processor, primary memory, and secondary memory, the best way of exploiting parallelism is by decomposing a task into several independent subtasks and processing them in parallel. In active systems, since multiple rules can trigger simultaneously, they can be executed in parallel on different processors [30]. Several AI researchers have worked on the execution semantics of such parallel firing systems [41, 45, 28, 30]. Two important problems have been identified in such systems [40]: i) controlling the execution order of rules, ii) guaranteeing the serializable execution of rules. In database systems, the second problem can be easily solved by viewing rules as database tasks, for which the serializability is automatically enforced by a database transaction manager [27]. Regarding the first problem, several active systems including POSTGRES [47], Ariel [23] and Starburst [50, 1] provide *user-defined priorities* to control the execution order. However these rule systems are sequential in the sense that only one rule (with the highest priority) is fired at a time which is a performance bottleneck in parallel rule systems [40]. On the other hand, HiPAC [8] and Raschid's work [39] support parallel firing of rules but do not provide any control over the rule execution order. For parallel active databases, there is a need for a parallel rule system that s supports the parallel firing of rules as well as provides the user-control over rule execution.

In active databases, rules and transactions are closely related in the sense that rules are treated as a part of the triggering transactions in most of the situations or they act as separate transactions [27]. The transaction control structure of an active system can be much more complex than a linear or hierarchical control structure because of the nested triggering of rules and the control relationship among rules can be expressed by a network structure. We believe that a well-integrated parallel active database system should meet the following two

important requirements: i) providing expressive control constructs for specifying complicated control semantics among rules, and ii) having an expressive transaction model that can accommodate complex structural relationships among database tasks and rules triggered by them.

In our work, we first enhance the active database rule system with several control constructs which can be used to define the parallel control structure of the rules and then propose a transaction model whose control structure can naturally incorporate the rule control semantics. In addition, the transaction model automatically guarantees the serializable execution of rules by treating them as its sub-units or subtransactions. Thus, the parallel rule execution model is uniformly integrated with that of transactions. Finally, we describe the design and implementation of an integrated parallel active database server for a shared-nothing multiprocessor system (nCUBE2) [20, 33, 9].

Our work is different from the advanced transaction models presented in [4, 49, 42, 25, 10, 2] due to the following reasons: i) the proposed graph-based model maintains the traditional ACID properties [21] which are useful in most applications for maintaining consistency and are difficult to be enforced in application programs, ii) we propose a new correctness criterion and concurrency control schemes for the parallel and interleaved execution of several graph-based transactions and rules based on the semantics of the proposed control structure, and iii) we present the detailed design and implementation of a parallel active database server running on a *shared-nothing* parallel computer.

The remainder of the paper is organized as follows. Section 2 defines the graph-based control structure to capture the control semantics of rules and transactions and describes how the proposed transaction structure includes the rule structure uniformly. Section 3 discusses the correctness criterion for the concurrent execution of rules and transactions and discusses two different schemes for concurrency control. Section 4 describes the design and implementation of the proposed transaction model on a shared-nothing multi-processor architecture. Section 5 compares our work with other related systems. Section 6 concludes our work and discusses its impact on other database environments.

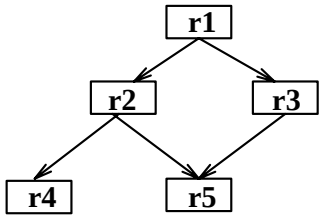

Figure 1: An example rule graph

2 Graph-based Control Structure

The transactions of active database systems are different from those of traditional systems mainly because they involve dynamically triggered rules which can be executed at various control points during the transactions' life time. Because of the close relationship between rules and transactions, the control structure among rules (which specifies the execution flow including the available parallelism) necessitates new control requirements for the active database transactions. In this section, we introduce several graph-based control constructs to capture the various control structures among rules and propose a control structure for transactions that can naturally support a hierarchical control structure due to nested triggerings of rules and a graph-based control structure due to the complex control relationships among rules.

2.1 Control Structure among Rules

We adopt a general graph-based approach to define the control structure among rules. The basic idea is that any complex control structure and inter-rule dependencies can be captured by a control flow represented by a directed acyclic graph (DAG). The graph-based control structure is more expressive than the parallel programming language (PPL) model in the sense that it allows several rules to synchronize at a point which is not possible with the hierarchical control structure provided by PPL constructs such as the `cobegin-coend` construct. Also, the control constructs provided by RDL1 [44] (*sequence*,*disjunct*,*saturate*) have the same disadvantage. For example, the control flow graph shown in Figure 1 cannot be expressed using the two mentioned approaches because rules r2, r3 synchronize at r5.

A set of rules with a graph-based control structure is called a *rule graph*. In a rule graph, nodes represent rules and edges represent the control flow. A rule graph contains all

the rules that are supposed to execute at a single control point[1]. While the control structure of a rule graph can be specified using a DAG, a set of *control constructs* is introduced for the convenience of rule designers to define the rule structure which is subsequently converted to a DAG form. The control constructs are as follows: i) *Precede construct:* It is denoted by $\stackrel{s}{\rightarrow}$ and the semantics of r1 $\stackrel{s}{\rightarrow}$ r2 means that rule r2 should *sequentially* follow r1, ii) *Precede-set construct:* It is denoted by $\stackrel{p}{\rightarrow}$ and the semantics of r $\stackrel{p}{\rightarrow}$ R where R is a set of rules, is that all the rules of R should begin only after the completion of rule r and they can be executed in *parallel*, and iii) *Sync-at construct:* It is denoted by $\stackrel{y}{\rightarrow}$ and the semantics of R $\stackrel{y}{\rightarrow}$ r, where R is a set of rules, is that all the rules of R should *synchronize* before rule r and they however can be executed in any order. Note that $\stackrel{p}{\rightarrow}$ and $\stackrel{y}{\rightarrow}$ constructs are equivalent to $\stackrel{s}{\rightarrow}$ when R contains a single rule.

The rule graph can be formally defined as a 2-tuple (R,D) where R is a set of rules and D is a set of control constructs among the rules. For example, consider a rule graph RG=(R,D) where R={r1,r2,r3,r4,r5} and D = {r1 $\stackrel{p}{\rightarrow}$ (r2,r3); (r2,r3) $\stackrel{y}{\rightarrow}$ r5; r2 $\stackrel{s}{\rightarrow}$ r4}. The rule graph RG can be graphically viewed as a DAG shown in Figure 1. In the DAG the nodes represent rules and edges represent the control flow among the rules. An edge r1 to r2 means r1 precedes r2, more specifically r2 can execute only after r1 is completed. Note that the DAG can capture any arbitrary execution flow. In the remaining part of this paper, we shall use the graphical representation for ease of understanding.

2.2 Control Structure of Transactions

A specific control structure is needed for active database transactions to capture the triggered rules. If a transaction T triggers a set of rules {r1,r2,r3,r4,r5}, a flat transaction model used in active systems [47, 23] would linearly expand itself to include the triggered rules [51] as shown in Figure 2. In the nested transaction model (NTM) [34, 8, 38], the rules are viewed as subtransactions of T as shown in Figure 3. NTM does not capture the control structure among the rules. The reason is, in the NTM, a transaction is viewed as a composition of an *independent* set of subtransactions. To capture the control structure among the rules uniformly, it is logical to view a transaction as a composition of a set of *inter-dependent*

[1] We assume that the control relationships exist only between those rules that are semantically defined to execute together (i.e., the trigger time and trigger condition(s) of all those rules are the same). This is reasonable because two different sets of rules that are supposed to execute at two different control points are very unlikely to have any relationships.

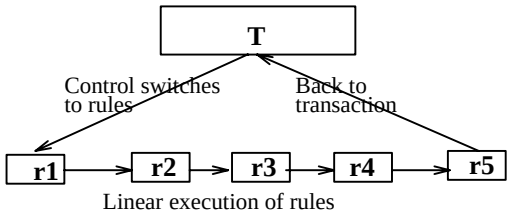

Figure 2: Flat Transaction Model

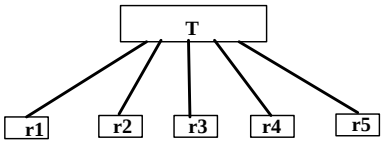

Figure 3: Nested Transaction Model

subtransactions. In other words, a transaction should be viewed as a control graph of subtransactions. In this paper, we call such a transaction a *transaction graph* and to be more general, we use the term *task* to refer to a subtransaction or a rule.

A transaction graph (TG) consists of a set of tasks in a DAG form as shown in Figure 4.a. In this figure, boxes represent the spheres of control [12] and the directed edges represent the control flow. This can be shown in a tree form as depicted in Figure 4.b where thick lines represent "contains" relationship similar to an NTM tree and dashed lines represent the control flow among the subtransactions. Each subtransaction in turn can have a similar structure. It can be observed that the TG structure *uniformly* captures the rule graphs. For example, if a transaction T triggers rules r1..r5 with the control flow shown in Figure 1, the rules are treated as subtransactions with the graph-based control flow as shown in Figure 5. Note that the control graph representation neither imposes a *strict sequential* execution on one extreme nor allows *total parallelism* that leads to a completely non-deterministic execution on the other extreme. It allows the user to define the control relationships among the rules from which it exploits the available parallelism and at the same time preserves the user-defined control.

Another important feature of the TG is its ability to support the control semantics of different trigger times[2]. An event is a combination of two components namely trigger time and trigger operation. The trigger time specifies *when* the triggered rule should be executed

[2] They are referred to as *coupling modes* in some literature [13]

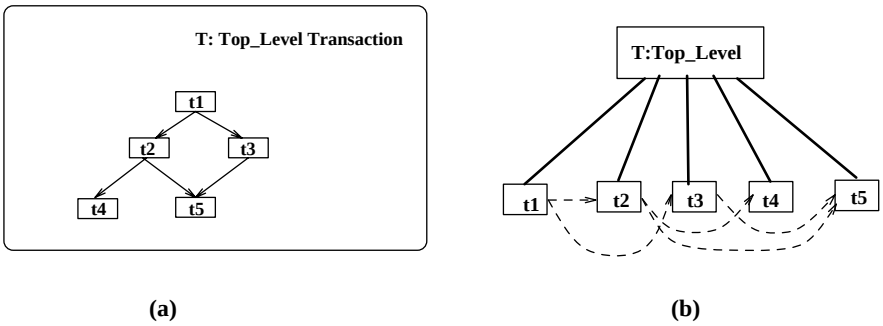

Figure 4: Transaction Graph Model

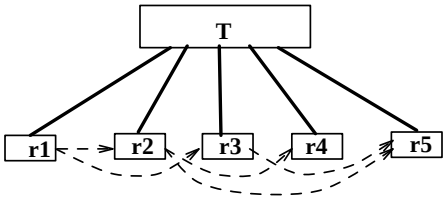

Figure 5: Active Rules as part of a Transaction Graph

relative to a trigger operation. For example, the following trigger times can be specified in our OSAM*'s rule specification language [43]: i) *before*: the rule is executed before the trigger operation, ii) *immediately-after*: the rule is executed immediately after the operation, iii) *after*: the rule is executed just before the commit time and iv) *parallel*: the rule is executed as a separate transaction. Some research is [7, 19] focused on developing powerful event specification languages in which a variety of trigger operations can be defined. Note that we are modeling only the trigger times but not the triggering operations. Therefore our approach can model rules with powerful events also. Furthermore, it is general enough to have rules executed at arbitrary points [3] of a transaction's life time as it is suggested in [6].

A TG can capture the control flow among all the tasks that constitute a transaction. To have a unique beginning and ending points for the transaction 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 6.a [4]. This can be represented using the tree form as shown in Figure 6.b. With this view of TG we explain the coupling of rule graphs to the TG according to the semantics of the trigger time.

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

[4] In the NTM, begin and commit tasks are embedded in the parent itself, they cannot be brought out because NTM does not have any control structure among the subtransactions.

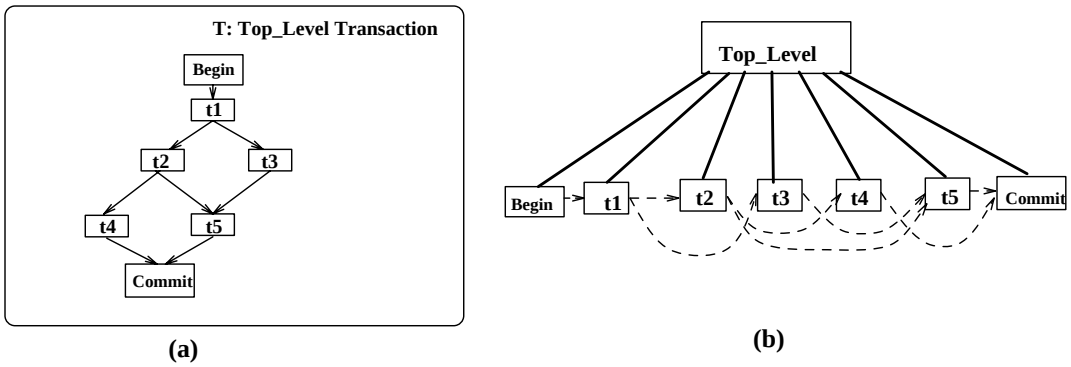

Figure 6: Adding begin and end points to a TG

In case of *before* or *immediately-after*, the triggered rules are treated as subtransactions of the triggering task. Since our model captures the control structure among subtransactions, the control structure among the rules is naturally modeled. For example, if an operation of t2 of the TG shown in Figure 6.b triggers the rule graph in Figure 7.c, it is coupled to the TG as shown in Figure 7.a. In case of *after* as the trigger time, in addition to the control structure, the *triggering order* of the *after* rules[5] is also captured. Although HiPAC does not capture the triggering order, the importance of such order is mentioned in [27]. In the above example, if the trigger time is *after*, the rule graph is coupled to the triggering transaction as shown in Figure 7.b. The triggered rules are treated as subtransactions of the top-level transaction and they are attached just before the commit node. In case of *parallel* as the trigger time, the triggered rule graph is executed as a separate transaction in parallel. It is completely detached from the triggering transaction 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. In the second case, the failure of parallel transaction does not affect the triggering transaction. It can be observed that, for all trigger times, the control structure among the rules uniformly fits into the proposed transaction control structure, which obviates a transaction manager treating rules differently from other subtransactions.

In addition to having a powerful control structure, the proposed transaction model follows the traditional ACID properties. Although, at the transaction level, we follow the standard serializability, within a transaction, we define a new correctness criterion that

[5] HiPAC refers them as *deferred* rules.

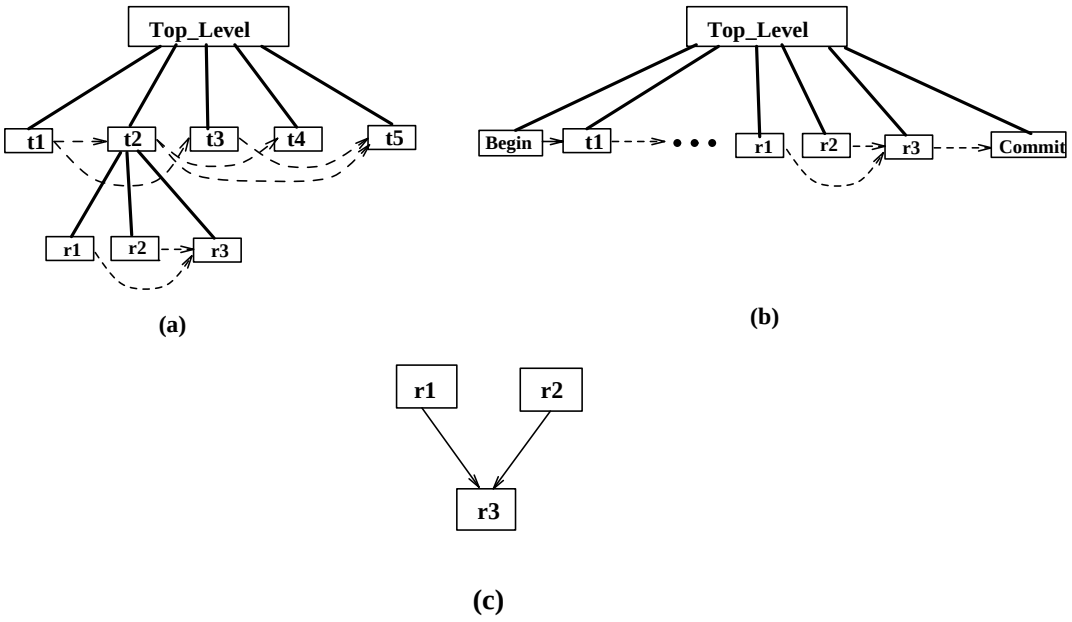

Figure 7: Coupling the rule graphs to a TG

enforces the control structure among the tasks in addition to the isolation.

3 Concurrency Control

In the TG structure, the subtransactions must follow a specific partial execution order in addition to being isolated from each other. A random execution order (although serial) can lead to semantically incorrect results. In this section, we define a correctness criterion for the parallel execution of tasks within a TG. We introduce two different schemes for maintaining the correctness during the parallel execution of tasks. The first scheme uses a scheduling algorithm and nested locking rules, and the second scheme uses graph-based locking rules that automatically enforce the control structure among the tasks. The second scheme exploits maximum parallelism if the tasks are operating on disjoint sets of data.

3.1 Correctness Criterion

In our model, at the transaction-level, the correctness criterion for parallel execution of several TGs is the standard serializability [15] which states that the interleaved (concurrent) execution of several TGs is correct when it is equivalent to an execution in some serial order. Within the transaction, the control flow among the tasks plays a role in defining the

correctness. We cannot adopt the task serializability as a correctness criterion because an arbitrary serial order can violate the control semantics. For example, for the TG shown in Figure 4.a, the order $t1 \rightarrow t4 \rightarrow t2 \rightarrow t3 \rightarrow t5$ is not correct. The parallel execution of tasks of a TG is *correct* if and only if the resulting order does not violate the control relationships. For example, for the TG shown in Figure 4.a, any of the orders $t1 \rightarrow t2 \rightarrow t3 \rightarrow t4 \rightarrow t5$ or $t1 \rightarrow t3 \rightarrow t2 \rightarrow t4 \rightarrow t5$ is correct. To be more general, the parallel execution of the tasks of a TG is correct when it is equivalent to some *topological order* [11] of the tasks. Since the above correctness criterion requires a topological serial order rather than an arbitrary serial order, we shall call it *topological serializability*.

Although the topological serializability prevents the tasks to execute in a complete parallel fashion, there is a scope for intra-transaction parallelism. A set of tasks can be executed in parallel if they are *independent* of each other. A task in a TG is independent if it does not have any incoming edge. In other words, the node with a zero *indegree* (number of incoming edges) represents an independent task. For example, for the TG shown in Figure 4.a, t1 is independent and the tasks t2,t3 become independent after the completion of t1 and they can be executed in parallel. In the next section we shall present scheduling and locking schemes that achieve the intra-transaction parallelism and, at the same time, maintain the topological serializability.

3.2 Topological Scheduling

The main goal of a concurrency control scheme is to achieve as much parallelism as possible and, at the same time, maintain the correctness criterion. The scheme we propose here achieves inter and intra-transaction parallelism and also maintains the correctness by enforcing topological serializability at the task level and standard serializability at the transaction level.

To maintain the topological order, the tasks in a control graph can be executed in a topological order sequentially, nevertheless with the expense of potential intra-transaction parallelism. To achieve the parallelism, the tasks can be divided into topological groups such that the tasks in each group are independent of each other and can be executed in parallel. The groups can be formed as follows. All the tasks in the control structure can be given a *level* number such that every task's level is lower than the levels of all its *successors*. A

task t2 is a *successor* to task t1 if there is a direct path from t1 to t2 in the control graph (formed by dashed lines), and t1 becomes t2's *predecessor*. Note that, this is different from child-parent or ancestor-descendant relationships, which are denoted by thick lines in the tree representation used. The tasks can be partitioned into topological groups by grouping them according to their level number. For example, the groups for the control graph shown in Figure 4.a are (t1:1),(t2:2, t3:2),(t4:3, t5:3). The number beside a task indicates the level. Within each group the tasks can execute in an arbitrary serial order but the order of the groups must be maintained. Although the sequentiality among the groups is undesirable, it is necessary to maintain the correctness with respect to the control flow. It can be ameliorated by executing the tasks *asynchronously* which we shall discuss later in this section. The following algorithm gives the topological groups for a given control graph.

```
Algorithm Topological-grouping
Input: A control graph
Output: Groups with levels

BEGIN
   Initialize v.indegree for all vertices (e.g. by DFS);
   level = 1;
   FOR i= 1 TO n DO    /* n is the total number of tasks */
      IF v(i).indegree = 0
      THEN  v.level = level;
            put v(i) in Queue;

   REPEAT
      remove vertex v from the front of the Queue;

      FOR all edges (v,w) DO
        w.indegree = w.indegree - 1;
        IF w.indegree = 0
        THEN w.level = v.level + 1;
             put w in the rear of the Queue;

   UNTIL Queue is empty
END;
```

The following simple algorithm schedules the tasks of a control graph in a topological order and exploits the parallelism among independent tasks.

```
Repeat
  Select
  Schedule
  Wait
  Remove
Until the control graph is completely executed
```

In the above algorithm, the `select` step simply selects the set of independent tasks in the given control graph. As shown in the topological_grouping algorithm, the independent tasks are identified by the indegree. In the `schedule` step, the independent tasks are assigned to appropriate processing nodes for parallel execution. The isolation among the tasks is maintained by a locking scheme which is explained later in this section. The `wait` step synchronizes all the tasks scheduled in one iteration by waiting for their completion. In the actual implementation, the transaction manager switches to another transaction during the `wait` step which is explained later in Section 4. In the `remove` step, all the completed tasks as well as the outgoing edges of those tasks are removed from the TG. This process is repeated until the whole control graph is executed. The theorem that proves the correctness of the above algorithm is given as Theorem 1 in the Appendix.

Asynchronous approach:

In the above scheduling algorithm the `wait` step waits for all the scheduled tasks to complete before scheduling the next set of independent tasks. This synchronization step can make some tasks wait *unnecessarily*. For example, consider the TG shown in Figure 4.a. It is scheduled in the following sequence of groups $\{t1 \rightarrow (t2, t3) \rightarrow (t4, t5)\}$ and note that the above algorithm waits until the completion of t2 and t3 to schedule t4 despite the fact that t4 is eligible for the execution as soon as the task t2 is completed. To avoid the unnecessary waiting of the tasks, the `wait` step in the above algorithm is modified and a *queue* data structure is introduced. The completed tasks place their acknowledgments in the queue asynchronously. The algorithm waits only when the queue is empty, otherwise proceeds to the `remove` step. In the `remove` step, all the tasks corresponding to the acknowledgments in the queue are removed from the control graph and also the outgoing edges. The `remove` step also removes the acknowledgments from the queue. This approach avoids the unnecessary waiting of the tasks and thereby increases the parallelism. The queue operations are synchronized to avoid inconsistent queue updates.

Locking scheme:

Although the above scheduling algorithms enforce the dependencies, they do not take care of the *isolation* of independent tasks. For example, parallel execution of two indepen-

dent tasks need not be serializable. In addition, we need to maintain the isolation of the TGs to facilitate the interleaved execution at the transaction level. We use the following nested locking rules[6] for maintaining the task isolation as well as the transaction isolation. The nested locking rules combined with the topological scheduling maintain the hierarchical control structure as well as graph-based control structure of the proposed transactions.

1. A task/transaction may hold a lock in write mode (read mode) if all other tasks/transactions holding the lock in any mode (in write mode) are ancestors of the requesting task/transaction.

2. When a task/transaction 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 task/transaction 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 child.

Theorem 2 of the Appendix proves that the proposed concurrency control method, which comprises of the topological scheduling algorithm and the nested locking rules, is correct.

3.3 Graph-based Locking

The concurrency method explained in the previous section is restrictive in that, even though two tasks are operating on *disjoint* sets of data items, they will not be executed in parallel if there is a dependency between them. In this section, we discuss a locking scheme which allows complete parallelism among all the tasks in a control structure. In this approach, there is no task scheduling. The control structure is enforced by the locking scheme automatically. As a result, the locking scheme is more complex. All the tasks in a control structure are allowed to execute simultaneously. This approach optimistically assumes that all the tasks will operate on disjoint sets of data or there is no read-write or write-write conflict in accessing the common data. In case two tasks have common data items in their write domains (or conflicting domains in general) and they violate the order, one of the tasks is aborted. This approach identifies the control flow violations by characterizing the task relationships. Two

[6] The first version of these rules appeared in [36]

tasks t1, t2 in a control graph are related in one of the following ways: i) t1 is a successor of t2, ii) t1 is a predecessor of t2, iii) t1 and t2 are independent of each other. In case a task tries to lock an item which is locked in a conflicting mode by a predecessor task (which is supposed to execute before the requesting task) then the requesting task waits. If it is locked by a successor task in a conflicting mode, then the holding task is aborted. In addition, all its successors are aborted. If it is locked by a task which is independent of the requesting task, then the requesting task waits. The detailed rules of the graph-based locking scheme are as follows:

1. A task/transaction may hold a lock in *write mode* if all other tasks/transactions holding the lock (in any mode) are either the *ancestors* or those *predecessors* of the requesting task/transaction that have been committed of the requesting task/transaction. In case the lock is held by any *successor* task, the holding task and all its *successors* are aborted and rolled back.

2. A task/transaction may hold a lock in *read mode* if all other tasks/transactions holding the lock in *write mode* are either the *ancestors* or those *predecessors* of the requesting task/transaction that have been committed of the requesting task/transaction. In case the lock is held by any *successor* task, the holding task and all its *successors* are aborted.

3. When a task/transaction aborts, all its locks, both read and write, are released. In addition, all the *descendant* tasks as well as the *successor* tasks are aborted. If its *ancestors* hold the same lock, they continue to do so in the same mode as before the abort.

4. When a task/transaction 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 child. A task *delays* the commit until all its *predecessors* are committed.

This scheme allows all the tasks to run in parallel if they are operating on disjoint data items. The properties of atomicity and isolation hold for each task. If there is an overlap in the data that is used by different tasks in a directed path of the control structure, the

tasks need to be aborted whenever there is a dependency violation. Like any other locking schemes, there is a possibility of dead lock in this scheme also. Typically, two tasks can be waiting for each other to release the locks. In this case, after a *time limit*, one of the tasks is aborted and rolled back. All the successor tasks are also aborted and the locks are released. The theorem that proves the correctness of the above locking scheme is given in Theorem 3 of the Appendix.

The recovery of the graph-based transactions can be done using the standard recovery methods used for NTM [35, 22]. The only difference is that, when a set of tasks are being undone, they need to follow the *reverse* of the topological order. A detailed discussion on the recovery techniques is out of the scope of this paper.

4 Implementation on a Shared-nothing Architecture

The proposed transaction model has been implemented as part of the OSAM*.KBMS/P [20, 9, 33] project which is being carried out at the Database Systems R&D Center, University of Florida. A DB server has been implemented on nCUBE2 - a high-performance parallel computer with shared-nothing architecture. The architecture of the system is shown in Figure 8 where C1,C2,...Cn are clients. All the clients which are SUN workstations, are connected to the server by an inter-connection network. A client contains window-based graphical user interfaces which have advanced features to edit TGs, rule graphs, queries and database schema containing classes and associations [32]. The server processes the transactions *asynchronously* and queues are used to hold the messages which can be transactions, tasks or acknowledgments, that are received at each node. The underlying data model for this parallel system is OSAM*/P - an extension of the object-oriented data model OSAM* [48].

In OSAM*.KBMS/P, a database application is defined by a set of classes and their various types of associations. Each class is defined in terms of its structural associations with other classes, a set of method specifications, a set of structured rules or rule graphs and the implementation of its methods. It is stored at and processed by a processing node. Multiple classes can be assigned to a processing node. A global data dictionary containing the overall schema information and the mapping between classes and the processing nodes

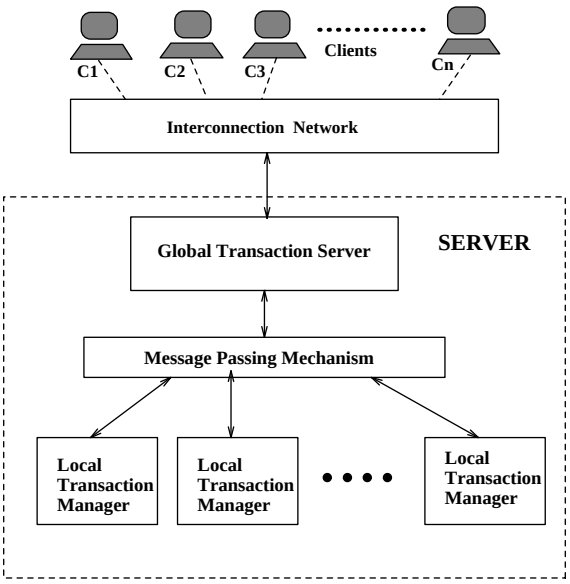

Figure 8: System architecture of OSAM*.KBMS/P

is maintained. A local dictionary that contains the structure of the local class and the rule graphs that can be triggered by the local operations (so that operations at a node triggers only the local rule graphs) is stored along with the instances of the object class in a node. It should be noted that rules are also distributed among the classes. As shown in Figure 8, the server has a global transaction server (GTS) and a group of local transaction managers (LTM) which are launched on different processors of nCUBE. The GTS receives the incoming transactions and supervises their interleaved execution. It also schedules the tasks and assigns them to the underlying LTMs for parallel execution based on the class-to-processor mapping and a load balancing strategy. In addition, it interacts with LTMs to schedule the triggered rule graph at appropriate time (immediately, during commit time and etc.) depending upon the trigger times. Each LTM receives the tasks from GTS and processes them in an interleaved fashion. It utilizes the GTS scheduler for scheduling the rule graphs according to their control structure and the local lock manager for maintaining the task serializability. Since database operations of a task are much more complex and time-consuming when compared with the topological scheduling at GTS, the LTMs seldom wait for the scheduler. The detail functionality and algorithm of the GTS and LTM are given in the following sections.

4.1 Global Transaction Server

The components of the GTS are shown in Figure 9.a. The GTS consists of a transaction queue (TQ), transaction scheduler (TS), wait queue (WQ) and a signal queue (SQ). In Figure 9.a, T1,T2 .. are TGs and G1,G2 .. are signals from LTMs. The GTS keeps the incoming TGs in the rear of the TQ. The 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 TS. The TS runs the asynchronous scheduling algorithm explained in Section 3.2. It gets the TG from the head of TQ, selects the independent tasks, distributes them to the underlying LTMs, and keeps the remaining tasks of the TG in the WQ. The GTS analyzes the task operations to identify a *hot class* for that task which is the class that is being used by most of the operations in that task. The task is assigned to the node corresponding to the hot class. GTS looks into the global data dictionary to find the node corresponding to the hot class. In case there are several hot classes, a load balancing strategy is used to select the appropriate node. The WQ consists of TGs that are partially processed and waiting for further processing. When a TG is in the WQ, TS gets another TG from TQ and starts scheduling it. It can be observed that the TGs are being interleaved by TS and also several tasks are being processed in parallel by the underlying LTMs. In this approach, the GTS has to be informed of when a task completes its execution, since it can make some other tasks of the TG eligible for execution. This is achieved by LTM sending a signal to SQ of the GTS. The SQ contains the signals coming from LTMs each of which indicates that an assigned task is completed/aborted. TS periodically monitors the SQ to check if any task is completed and schedules the next set of independent tasks of the corresponding TG. The completed tasks are marked and the TG is kept in the WQ. A TG's execution is completed when TS reaches the commit node of the TG. The GTS requests all relevant LTM's to commit the transaction using 2PC. The committed TG is removed from the WQ.

The TS, in addition to the TGs, schedules the rule graphs also. The rule graphs with trigger time *before* or *immediately-after* wait in WQ of the GTS and they are scheduled for processing when TS sees the signal from the LTM that triggered the rule graph. This is explained further in the next section. For the rule graph with trigger time *parallel* wait in TQ and is scheduled as a top-level transaction. The rule graph with trigger time *after*,

TS attaches it just before the commit node of the TG as shown in Figure 7.b thereby they become part of the control graph and will eventually be scheduled for processing just before the commit of the transaction. The detailed algorithm of the TS is as follows.

```
Algorithm TransactionScheduler

  Loop

     If  Signal_Queue is Empty Then
         Get transaction T from the Transaction_Queue;
     Else
         Get the transaction T corresponding to the signal
         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 TransationScheduler;
```

4.2 Local Transaction Manager

The GTS keeps the task in the LTM's task queue for processing. It is the LTM's responsibility to execute the task efficiently and consistently in addition to the maintenance of the triggered rule graphs. As shown in Figure 9.a, the LTM consists of a task processor (TP), a data processor (DP), a lock manager (LM), a recovery manager (RM), a task queue, a wait queue (WQ) and an acknowledgment queue (AQ). LTM keeps track of the control flow among the operations of a task which are initially in a sequence and evolve into a complex control structure because of dynamically triggered rule graphs.

The LTM processes the incoming tasks in an interleaved fashion. The tasks to be processed are stored in the task queue. When the TP has to suspend the processing of a

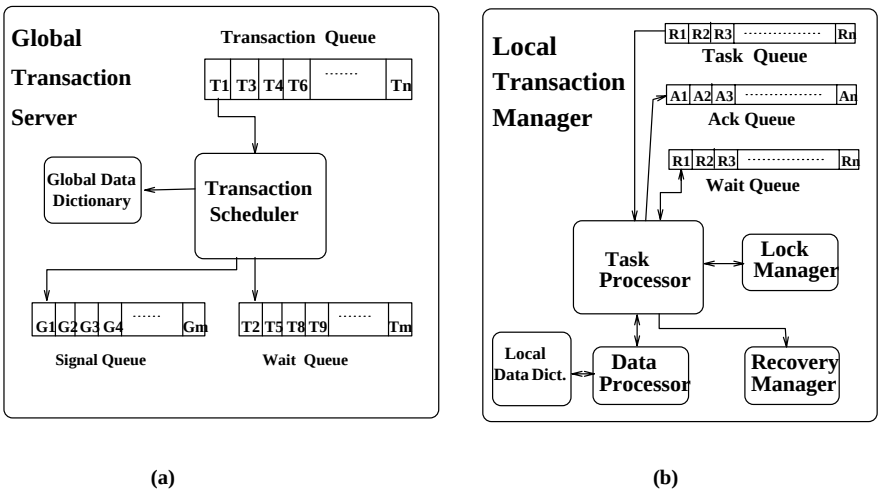

Figure 9: Components of the Server

task, it would keep that task in the WQ and mark the control point to resume the processing later. TP calls the underlying DP for performing data processing operations. TP processes rules also. Basically it processes database operations specified in rules. TP acquires the locks for the data being accessed before sending the operations to the data processor. LM receives the requests for locks from the local TP and sometime from the remote TPs. The local LM is responsible for the locks on the object class that is stored in that particular node and it follows the locking rules explained in Section 3.2.

When a task is being processed, an operation can trigger a rule graph. The TP handles the dynamically triggered rule graphs depending upon the trigger times as follows. i) In case of *before*, the rule graph is kept in the WQ of GTS and a signal which includes the rule graph id is kept in SQ. The task is suspended and kept in the local WQ. GTS schedules the rule graph and acknowledges to the LTM when the rule graph completes its execution. TP resumes the task as soon as it sees the acknowledgment in the AQ. ii) In case of *immediately-after*, the operation is performed and the task is kept in the local WQ. The TP does exactly the same as the case *before*. iii) In case of *after*, the rule graph is kept in the GTS's WQ and the TP resumes the task processing. The GTS appends the rule graph to the TG just before the commit node as shown in Figure 7.b. iv) In case of *parallel*, the triggered rule graph is kept in the transaction queue. The rule graph is scheduled like a TG. The detailed algorithm for the LTM is as follows.

`Algorithm LTM`

```
  Loop

   If  Ack_Queue is Empty Then
     Get Task R from the Task_Queue;
   Else
     Get the Task R corresponding to the signal
     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 a signal in GTS signal
             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 a signal in GTS signal
             queue; Exit;
        after: Keep the triggered rule graph in
             the wait queue and keep a signal
             in signal queue;
        parallel: Keep the triggered rule graph
             in the transaction queue;

       Else  Obtain locks for the operation and
             send the operation to data processor;

    Until end of the task;

    Send a signal to GTS signal queue that the
    task is completed;

  Forever;

End LTM;
```

5 Related Work

There have been several research efforts on execution models for the rules and transactions of active database systems [27, 37, 47, 8, 23, 48, 46, 41, 29, 50, 5, 18, 38, 3, 24, 6, 26, 16]. However, very few of them focus on the design and implementation issues in shared-nothing parallel environments. Although our system as an integrated parallel active database server is unique, we shall compare our parallel rule execution model and the transaction model with that of some related systems.

5.1 Parallel Rule Execution Model

The main feature of our rule execution model is user-defined control over parallel execution of rules. The rule execution models of POSTGRES [47], Ariel [23], Starburst [50], RDL1 [44], OPS5 [17] support explicit control over the rule execution[7], however, in these systems, only one rule is fired at a time which is not ideal in a parallel environment. To avoid this problem, several parallel rule execution models have been proposed [27, 41, 40, 28, 45, 30, 31]. In [30, 45], several control constructs are provided to control the parallel execution of rules but they do not guarantee serializable execution of rules. It is the responsibility of the rule programmer to maintain the serializability using the control constructs. On the other hand, in [40, 28], serializable execution of rules is guaranteed (by the static analysis of the rules) but user-defined control over rules is not supported. Also the above works are in the context of AI and do not discuss the integration of the parallel rule execution model with that of transactions which is essential for database rule systems. HiPAC [8] integrates the parallel rule execution model with that of transactions. The rules are modeled as subtransactions and the serializability among rules is guaranteed by the underlying nested transaction model [34, 27]. Other works that support parallel execution of rules in database environments include [6, 39]. In [39], Raschid defines a correctness criterion for concurrent execution of rules in database environments. The semantic relationships of the rules are analyzed for establishing an execution order among interfering rules and also it is made sure that the condition remains true when the action is executed. In our system, data used by condition is locked until the completion of the action, so there is no possibility of condition being not true

[7] In OPS5, it is not explicit, the rule programmer has to carefully design the rules such that desired order is imposed by the OPS5 conflict resolution criteria.

when the action is executed. The main disadvantage with the above systems is lack of user-defined control over rule execution. Ceri [6] proposes several locking protocols for parallel and distributed execution of rules maintaining the user-defined priorities for Starburst. Since in Starburst priority system, a total order is established among the rules using a combination of user-defined and default priorities [1], making the system deterministic at the same time reducing the scope for potential inter-rule parallelism. Our approach gives the user control over the rule execution order in one hand and exploits the parallelism among independent rules in the other hand. In addition, the rule serializability is maintained automatically using the DB locking scheme. While we discuss the strategies for executing multiple rules in parallel, in [26], it is discussed how a rule can be further decomposed into sub-rules and can be executed concurrently.

5.2 Transaction Model and Trigger Times

POSTGRES [47], Ariel [23] use flat transaction model in which the triggered actions are incorporated by linear expansion of the transaction [51]. The triggered actions they are executed immediately. In POSTGRES, triggered actions can occur on demand also (by lazy evaluation). HiPAC [27], Beeri [3], Raschid [38] and Starburst [6] are some of the works that use nested transactions to support the concurrent execution of rules. Starburst supports only trigger time *after* and all the triggered rules are executed at the end of the top-level transaction as subtransactions. However, in [6] it is mentioned that their approach is extensible to make any point of transaction life-time as trigger time. HiPAC and Beeri provide variety of trigger times for rules which include immediate, deferred and detached[8]. The immediate rule-tasks (the tasks generated by rules that need to be executed immediately) are modeled as subtransactions of the triggering task, deferred actions are modeled as subtransactions of the top-level transaction, and detached actions are modeled as separate transactions. NTM locking rules automatically maintains the correctness of the rule-tasks since they are modeled as an integrated part of the NTM control structure. In our system, all the above execution options are supported and all the triggered tasks are modeled in the same way. In addition, our transaction structure (which can be viewed as a nested model with control structure among siblings) can model the control structure among rules, and the triggering order of

[8]The semantics are same as our immediately-after, after and parallel.

deferred rules which is not possible with the transactions supported by the above systems. Also, our graph-based model is upward compatible with NTM in the sense that the applications developed for NTM are directly supported by graph-based transaction model. Several other transaction models including [10, 49, 42, 4, 25, 2] have advanced control structure for transactions. However our transaction model is different in that it maintains the traditional ACID properties [22], incorporates the control semantics of active rules and their trigger times, and it is designed for and implemented on a shared-nothing parallel architecture.

6 Conclusions

A transaction model with a more general control structure represented by a DAG has been introduced to capture the control semantics of active database transactions. After a detailed analysis of the run-time behavior of active database transactions, we have identified their requirements which include modular view of rules, uniform representation of rule control (which is very useful in parallel environments to execute the rules in parallel without violating the intended semantics.) and natural incorporation of the rules into the transaction structure according to their trigger time semantics. It is shown that the proposed graph-based control structure can model the control relationships among the rules and also enable the transaction framework to include the rules uniformly and naturally. Two different concurrency control schemes have been discussed for correct and concurrent execution of the tasks in the new control structure. The locking scheme and the scheduling algorithms are useful for both transactions as well as rules. In addition, all the proposed concepts have been implemented and tested on a parallel shared-nothing architecture with a real database and the schema designed in OSAM*/P.

The proposed transaction model can be used not only in active database environments, but also as a more expressive transaction model for any database environments, e.g., it can be used in multi-database environments to model the inter-data dependencies which can be captured using a dependency graph, and also it can be used to capture inter-task dependencies within a transaction. The main high-light of this model is, it allows the *decomposition* of a transaction into several subtransactions, and, at the same time, it allows an *arbitrary control structure* among the subtransactions. It is also appropriate, in object-oriented and

distributed environments because the natural decomposition of a task into several subtasks improves the modularity and parallelism, and the control structure among subtransactions allow to capture arbitrary semantic relationships and also explicit data dependencies. The proposed concurrency control techniques not only suit shared-nothing parallel environments, but also distributed and centralized database environments.

References

[1] Rakesh Agrawal, Roberta Cochrane, and Bruce Lindsay. On maintaining priorities in a production rule system. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, September 1991.

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

[3] C. Beeri and Tivo Milo. A model for active object oriented database. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, September 1991.

[4] A. Buchmann, M. Tamer Ozsu, et al. *Database Transaction Models for advanced applications*, chapter A transaction model for active distributed object systems, pages 123–158. Morgan Kaufmann Publishers, 1991.

[5] Michael J. Carey, Rajiv Jauhari, and Miron Livny. On transaction boundaries in active databases: A performance perspective. *IEEE Trans. Knowledge Data Eng.*, 3(3), September 1991.

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

[7] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[8] Sharma Chakravarthy et al. Hipac: A research project in active, time-constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[9] Prashant V. Cherukuri. A Task Manager for Parallel Rule Execution in Multi-processor Environments. Master's thesis, Department of Electrical Engineering, University of Florida, 1993.

[10] Panos K. Chrysanthis and Krithi Ramamritham. *Database Transaction Models for advanced applications*, chapter ACTA: The SAGA Continues, pages 349–398. Morgan Kaufmann Publishers, 1991.

[11] Thomas H. Cormen, Chales E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill Book Company, 1990.

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

[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] Oscar Diaz, Norman Paton, and Peter Gray. Rule management in object oriented databases: A uniform approach. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, September 1991.

[15] 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.

[16] Opher Etzion. Pardes-a data-driven oriented active database model. *SIGMOD RECORD*, 22(1):7–14, 1993.

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

[18] 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.

[19] 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.

[20] Arun Gopalan. Transaction Management and Recovery in a Disributed Object-Oriented Database System. Master's thesis, Department of Electrical Engineering, University of Florida, 1992.

[21] T. Haerder and A. Reuter. Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4):287–318, Dec 1983.

[22] T. Haerder and K. Rothermal. Concepts for transaction recovery in nested transactions. *Proceedings of SIGMOD*, 12(2), July 1987.

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

[24] Eric N. Hanson and Jennifer Widom. *Advances in Databases and Artificial Intelligence*, chapter Rule Processing in Active Database Systems. JAI Press, 1992.

[25] Sandra Heiler, Sara Haradhvala, Stanley Zdonik, Barbara Blaustein, and Arnon Rosenthal. *Database Transaction Models for advanced applications*, chapter A flexible framework for transaction management in engineering environments, pages 87–122. Morgan Kaufmann Publishers, 1991.

[26] Ing-Miin Hsu, Mukesh Singhal, and Ming T. Liu. Distributed rule processing in active databases. In *Data Engineering*, 1992.

[27] 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, 1988.

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

[29] 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.

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

[31] Steve Kuo, Dan Moldovan, and Seungho Cha. Control in Production Systems with Multiple Rule Firings. Technical Report PKPL 90-10, Parallel Knowledge Processing Lab., U. of Southern California, March 1992.

[32] H. Lam, S.Y.W. Su, et al. GTOOLS: An Active Graphical User Interface Toolset for an Object-oriented KBMS. *International Journal of Computer Science and Engineering*, 1991.

[33] Qiang 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.

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

[35] Elliot Moss. Log-based recovery for nested transactions. In *Proc. 13th Int'l Conf. on Very Large Data Bases*, 1987.

[36] J. Moss. Nested Transactions: An Approach To Reliable Distributed Computing. MIT Laboratory for Computer Science, MIT/LCS/TR-260, 1981.

[37] L. Raschid and S.Y.W. Su. A transaction oriented mechanism to control processing in a knowledge base management system. In *Proc. of the 2nd Intl Conf on Expert Database Systems*, 1988.

[38] Louiqa Raschid, Timos Sellis, and Alexis Delis. On the concurrent execution of production rules in a database implementation. Technical Report UMIACS-TR-91-125, University of Maryland, September 1991.

[39] Louiqa Raschid, Timos Sellis, and Alexis Delis. A simulation-based study on the concurrent execution of rules in a database environment. *Journal on Parallel and Distributed Computing*, 20(1), Jan 1994.

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

[41] James G. Schmolze and Suraj Goel. A parallel asynchronous distributed production system. In *Proceedings of the 8th National Conference on Artificial Intelligence*, July 1990.

[42] Amit P. Sheth, Marek Rsinkiewicz, and George Karabatis. *Database Transaction Models for advanced applications*, chapter Using Polytransactions to manage interdependent data, pages 555–582. Morgan Kaufmann Publishers, 1991.

[43] Yuh-Ming Shyy and Stanley Y.W. Su. K: A high-level knowledge base programming language for advanced database applications. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, May 1991.

[44] 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*, Vancouver, 1992.

[45] Salvatore J. Stolfo et al. Parulel: Parallel rule processing using meta-rules for redaction. *Journal on Parallel and Distributed Computing*, 13(4):366–382, December 1991.

[46] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Trans. Knowledge Data Eng.*, 2(1):125–142, March 1990.

[47] Michael Stonebraker, Eric Hanson, and Spiros Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[48] 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, 1989.

[49] Helmut Wachter and Andreas Reuter. *Database Transaction Models for advanced applications*, chapter THE ConTract MODEL, pages 219–264. Morgan Kaufmann Publishers, 1991.

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

[51] Daniel Rios Zertuche and Alejandro P. Buchmann. Execution models for active database systems: A comparison. Technical Report TM-0238-01-90-165, GTE Laboratories, January 1990.

Appendix

Theorem 1: The algorithm schedules the tasks of a given control graph without violating the inter-task dependencies.

Proof: We prove the theorem by contradiction. Assume that there is a scheduled task t that violates the dependency, which implies that it is scheduled before the completion of t's predecessor say t1. According to the predecessor definition there is a direct path from t1 to t in the DAG corresponding to the TG. This implies that there is at least one incoming edge to t when it is scheduled which means t's indegree is not zero. This contradicts with the hypothesis that the algorithm schedules only independent tasks. Hence the proof. Note that the algorithm waits until the completion of all tasks that are scheduled and then it removes the tasks and the outgoing edges.

Theorem 2: The proposed concurrency control method which comprises of the locking scheme together with the topological scheduling algorithm is correct.

Proof: The concurrency control method is correct if and only if the following criteria are satisfied: i) within a transaction, the concurrent execution of the tasks is equivalent to some topological order, and the ii) concurrent execution of transactions is equivalent to some serial order of execution.

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

Theorem 3: The proposed graph-based locking scheme maintains the transaction serializability and, within the transaction the topological serializability.

Proof: This proof assumes that if there are no conflicting operations in two tasks, even if there is a dependency, they need not follow any particular order. We prove the theorem by contradiction. Assume that the proposed locking scheme violates the control flow, i.e. a task t1 commits before its predecessor t2 and they have conflicting operations on the same data item. This contradicts rule 4 which ensures that the commit of a task is delayed until all the predecessors are committed.

Assume that a task t1 performs an exclusive operation on a data item before its predecessor t2 and both of them eventually commit. Since t1 performed the operation first, it would have acquired the exclusive lock on the data item first. Eventually the predecessor t2 requests for the lock. Because the lock is owned by its successor (t1), according to the locking rule 1, the successor is rolled back and the lock is granted to the predecessor which effectively makes the predecessor to execute first. This contradicts with the assumption that both of them commit with the conflicting order.

Whenever a task is rolled back, all the successors are also rolled back. This preserves the execution order among the tasks with a predecessor-successor relationship. Abort of a task does not cause the tasks other than successors to abort. This is logical because the predecessors are not dependent on the successor (it is the other way around). A task does not use the data locked by another independent task until it is committed to make sure that it would never be aborted by the lock manager. This is enforced in rule 4. Hence the topological order is maintained.