Pergamon

Information Systems Vol. 23, No. 1, pp. 39–64, 1998

Printed in Great Britain
0306-4379/98 $17.00 + 0.00

CONCURRENT RULE EXECUTION IN ACTIVE DATABASES†

YÜCEL SAYGIN[1], ÖZGÜR ULUSOY[1] and SHARMA CHAKRAVARTHY[2]

[1] Department of Computer Engineering and Information Science, Bilkent University, Turkey
[2]Department of Computer and Information Science and Engineering, University of Florida, US

(Received 22 April 1997; in final revised form 3 February 1998)

Abstract — An active DBMS is expected to support concurrent as well as sequential rule execution in an efficient manner. Nested transaction model is a suitable tool to implement rule execution as it can handle nested rule firing and concurrent rule execution well. In this paper, we describe a concurrent rule execution model based on parallel nested transactions. We discuss implementation details of how the flat transaction model of OpenOODB has been extended by using Solaris threads in order to support concurrent execution of rules. Copyright ©1998 Elsevier Science Ltd

Key words: Active Databases, Nested Transactions, Rule Execution.

1. INTRODUCTION

Conventional, passive databases execute queries or transactions only when explicitly requested to do so by a user or an application program. In contrast, an active database management system (ADBMS) allows users to specify actions to be executed when specific events are signaled [13]. The concept of active databases has been originated from the production rule paradigm of Artificial Intelligence (AI). The AI production rule concept has been modified for the active database context so that rules can respond to the state changes caused by the database operations [21]. An active database implements reactive behavior since it is able to detect situations, which may occur in and out of the database, and to perform necessary actions which were previously specified by the user. In the absence of such an active mechanism, either the database should be polled or situation monitoring should be embedded in the application code. Neither of these approaches is completely satisfactory. Frequent polling degrades performance of the system and infrequent polling may deteriorate the timeliness of system responses. Embedding situation monitoring in the application code is error prone and reduces the modularity of the application [13].

Applications supported by ADBMSs cover a wide range of areas like authorization, access logging, integrity constraint maintenance, alerting, network management, air traffic control, computer integrated manufacturing, engineering design, plant and reactor control, tracking, monitoring of toxic emissions, and any other application where large volumes of data must be analyzed to detect relevant situations [5, 13]. Active DBMSs are proposed for system level applications as well, like supporting different transaction models [8].

In a typical ADBMS, system responses are declaratively expressed using Event-Condition-Action (ECA) rules [13]. An ECA rule is composed of an *event* that triggers the rule, a *condition* describing a given situation, and an *action* to be performed if the condition is satisfied. Coupling modes between event and condition, and between condition and action determine when the condition should be executed relative to the occurrence of the event, and when the action should be executed relative to the satisfaction of the condition, respectively.

An ADBMS should support both concurrent and sequential rule execution. Sequential rule execution is necessary when a certain execution order is enforced by priorities or when the rules have a predefined sequence of execution. Sequential execution may also be supported in levels; i.e., a certain number of groups of rules can be executed sequentially while the rules in each group are executed concurrently. Concurrent rule execution, on the other hand, is very important from the performance perspective of the system. Nested transaction model [27] is considered as

†Recommended by Nicole Bidoit

a suitable tool to implement rule execution since it provides us with a good model for concurrent rule execution and it can handle nested rule firing together with multiple triggering of rules well. Nested rule firing occurs when the condition evaluation or action execution of a rule causes some other rules to be fired which may go on recursively and multiple firing of rules occurs when an event causes more then one rule to be fired.

In this paper, we describe a rule execution model for ADBMSs based on parallel nested transactions. Our execution model supports concurrent rule execution through nested transactions, and sequential rule execution through user defined priorities. Rules with the same priority are executed concurrently which allows us to have sequential execution among rules with different priority levels and concurrent execution for the rules with the same priority. We also discuss the implementation details of concurrent rule execution using parallel nested transactions. The locking protocol described in [23] has been implemented which allows us to control the concurrency among all the transactions in a transaction hierarchy running in parallel. Implementation has been performed by extending the flat transaction semantics of OpenOODB using Solaris threads. OpenOODB is an open (i.e., extendible) object oriented database management system developed by Texas Instruments [37]. In our implementation, all the transactions in a transaction hierarchy are allowed to run in parallel, therefore achieving the highest level of concurrency. Solaris threads are used for running the subtransactions in parallel which provides us with efficient handling of transactions executing concurrently [36]. Our implementation of concurrent rule execution is currently being integrated into Sentinel [9] which is an ADBMS developed at the University of Florida.

In summary, the primary contributions of our work are:

- to describe a rule execution model that supports sequential as well as concurrent rule execution (in terms of both inter and intra-rule concurrency),
- to use nested transactions as the tool for modeling concurrent and nested rule execution,
- to implement the proposed nested transaction model for concurrent rule execution using threads.

A detailed discussion of the issues introduced in this section is provided in the following sections. In Section 2 we provide a detailed description of rule execution in ADBMSs. Our execution model for ADBMSs is described in Section 3 together with its formal specification. Section 4 describes the implementation details of parallel nested transactions on OpenOODB using Solaris threads, and discusses the integration of our implementation into Sentinel. A brief discussion on the execution overhead of parallel nested transactions is provided in Section 5. We conclude the paper by outlining future directions of this work.

2. RULE EXECUTION IN ADBMSS

Rule execution has a key role in determining the performance of an ADBMS. Nested and multiple rule firing cause a large set of rules to be fired for execution. Multiple rule firing occurs when an event causes more then one rule to be fired as illustrated in Figure 1(a) and nested rule firing occurs when the execution of a rule causes other rules to be fired recursively as illustrated in Figure 1(b). Depending on the level of concurrency allowed, a hierarchy of rules fired by the same event (i.e., multiple firing of rules) or rules fired recursively (i.e., nested rule firing) execute in the system. Management of nested and multiple firing of rules is a major scheduling issue in ADBMSs.

Rules can be executed sequentially or concurrently. Sequential rule execution is necessary when a certain execution order is enforced by priorities or when the rules have a predefined sequence of execution. Concurrent rule execution, on the other hand, is very important from the performance perspective of the system. Concurrency in rule execution can be achieved through either:

- inter-rule concurrency, or
- intra-rule concurrency, or
- both inter and intra-rule concurrency.

In the first case, rules are executed concurrently as if they are atomic transactions. In the second case, rules are divided into subcomponents and those subcomponents are executed concurrently. As another alternative, we may have both types of concurrency together, which is the most flexible concurrent rule execution model.

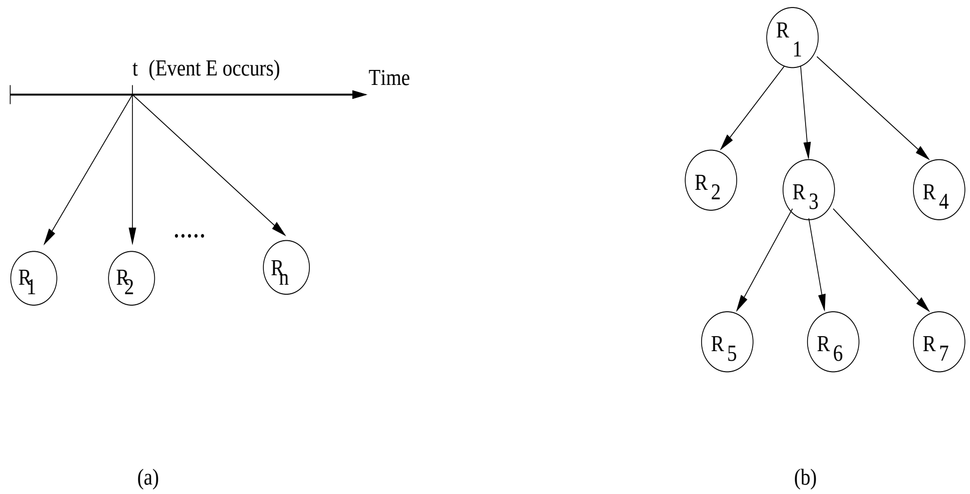

Fig. 1: (a) Multiple Rule Firing, (b) Nested Rule Firing.

Nested transactions are accepted as a suitable tool for implementing concurrent rule execution in ADBMSs which can handle multiple and nested rule firing well [6, 14, 24]. In the nested transaction model, some transactions can be started inside some other transactions forming a transaction hierarchy. The transaction at the top of the hierarchy is called a top-level transaction, and the other transactions are called subtransactions. Subtransactions can be executed concurrently which is a desirable situation if subtransactions are performing tasks that can be overlapped. Concurrency control of nested transactions is discussed in [23]. Multiple rule firing can be handled by executing the rules fired by the same event as subtransactions of the transaction causing the event.

A rule in an ADBMS consists of an *event*, a *condition* and an *action*. If the event is missing, then the rule is a *condition-action* (CA) rule or a *production rule*; if no condition is specified, then the resulting rule is an *event-action* (EA) rule or simply a *trigger* [30]. When an event is detected, the system searches for the corresponding rules. The condition part of the rule triggered by that event is evaluated and the action is taken if the condition is satisfied.

The action part of a rule may be executed in one transaction immediately as a linear extension of the triggering transaction. This is called *coupled execution* [24]. In coupled execution, we do not have intra-rule concurrency. We can give Starburst as an example of coupled execution of rules [38, 1]. In Starburst, rules are based on the notion of transitions. A transition is a change in database state resulting from the execution of a sequence of data manipulation operations. Rules are activated at assertion points. There is an assertion point at the end of each transaction and users may specify other assertion points within a transaction. The state change resulting from the database operations issued by the user since the last assertion point creates the first relevant transition which triggers a set of rules. A rule r is chosen from the set of triggered rules such that no other triggered rule has precedence over it. Condition of r (if it has any) is evaluated. Action part of r is executed provided that its condition evaluates to true; otherwise another rule is chosen. After the execution of r's action, rules that are not considered up until now are triggered only if their transition predicates hold with respect to the predicate created by the composition of the initial transition and the execution of r's action. Rule processing terminates after all triggered rules are executed. Ariel is another ADBMS which uses coupled rule execution [20]. Rule instantiations in Ariel are set oriented and execution of rules is performed by recognize-act cycle as in production rules.

Although coupled execution is useful in some cases, it degrades the performance of the system by increasing the response time of transactions. If we allow actions to be executed in separate transactions, then the triggering transaction can finish more quickly and release resources earlier, and this way transaction response times can be improved. We may also want the condition part of the rule to be executed as a separate transaction since conditions which are queries on the database can be long and time consuming. Allowing conditions and actions to be executed in separate transactions is called *decoupled execution* [15, 24]. Decoupled execution is useful especially when the fate of the condition or action is independent of the commit or abort of the originating transaction [15]. With decoupled execution we may have intra-rule concurrency.

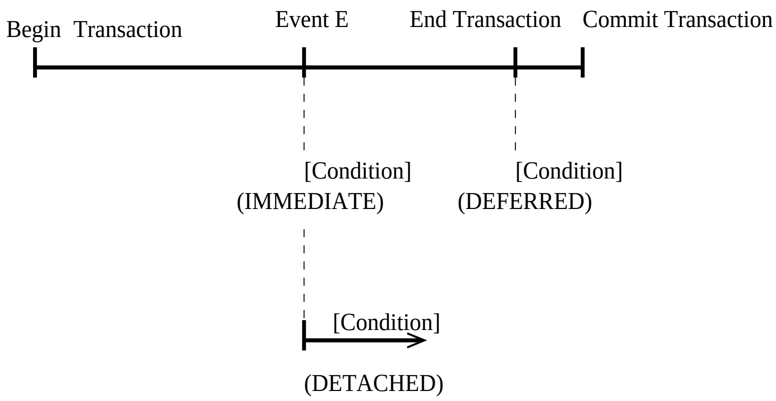

Fig. 2: Basic Coupling Modes between Event and Condition.

It is also important to specify when the condition will be evaluated relative to the triggering event and when the action will be executed relative to the condition evaluation. This is achieved by defining *coupling modes* for conditions and actions which are stated to be a functionality that an active database should provide [7]. There are three basic coupling modes: *immediate*, *deferred*, and *detached* (or *decoupled*) [13]. Basic coupling modes between event and condition are illustrated in Figure 2. If the condition is specified to be evaluated in *immediate* mode, then it is executed right after the triggering operation that caused the event to be raised. If the action part is specified to be executed in immediate mode then it is executed immediately after the evaluation of the condition. In case the condition is specified to be in *deferred* mode, its evaluation is delayed until the commit point of the transaction, and similarly if the action is in deferred mode relative to the condition, again it is executed right before the transaction commits. Finally, in *detached mode*, condition is evaluated or action is executed in a separate transaction. Detached mode can further be classified into four subcategories: *detached coupling*, *detached causally dependent coupling*, *sequential causally dependent coupling*, and *exclusive causally dependent coupling* [4]. In *detached coupling* there is no dependency between the triggering and triggered transactions. In *detached causally dependent coupling*, the triggered transaction can commit only if the triggering transaction commits. In *sequential causally dependent coupling*, the triggered transaction can start executing only if the triggering transaction commits. Finally, in *exclusive causally dependent coupling*, triggered transaction commits only if the triggering transaction fails.

A deferred execution method proposed in [24] and mentioned in [15] involves the execution of transactions in cycles. In cycle-0, deferred transactions that have been fired up to that point are executed. Transactions spawned during cycle-0 in immediate mode are executed as a linear extension of their parents as usual. Execution of the deferred transactions that are fired during cycle-0 by another deferred transaction is postponed to the next cycle, which is cycle-1. Again deferred transactions that are fired in cycle-1 are postponed to the next cycle, which is cycle-2, and so on. This process continues until there are no deferred transactions left.

Intra-rule concurrency is achieved by dividing a rule into a condition and an action, and executing them concurrently. In our execution model, intra-rule concurrency is in the condition-action level, i.e. there are no subcomponents of conditions or actions executing concurrently. This can be extended to finer granularity intra-rule parallelism by using nested transactions. Inter-rule concurrency can be achieved by executing the rules concurrently. In the following subsection, nested transactions are explained as a concurrent rule execution model for ADBMSs with a discussion of different kinds of parallelisms supported.

3. RULE EXECUTION MODEL

Our rule execution model is based on parallel nested transactions which can support concurrent rule execution while managing multiple and nested firing of rules. Nested transactions are proposed as a means to control the activities in various areas ranging from long-running activities [16] to resource management systems [31]. Parallel nested transactions are accepted to be a suitable tool for concurrent rule execution in ADBMSs as well [6, 14, 24]. In what follows, we will discuss nested transactions together with the concurrency control, deadlock detection, and recovery issues.

3.1. Basic Concurrency Control, Deadlock Detection and Recovery Issues in Nested Transactions

Traditional (i.e., flat) transactions have only one branch of execution. In the nested transaction model, transactions can have multiple branches of execution. A nested transaction forms a hierarchy which can be represented as a tree structure, and standard tree notions like parent, child, ancestor, descendant, superior, and inferior also apply to it. The root of the tree is called a root or a top-level transaction. The root may have one or more children, and similarly children of the root may also have their own children. By dividing transactions into smaller granules, we localize the failures into subtransactions. Subtransactions can abort independently without causing the abortion of the whole transaction hierarchy. When a transaction aborts, all of its descendants are also aborted, but other transactions are not affected. Nested transactions are also very useful in terms of system modularity. If we consider a transaction hierarchy as a big module, its subtransactions may be designed and implemented independently as submodules, also providing encapsulation and security [23].

Using nested transactions, we can exploit the parallelism among subtransactions since subtransactions can be executed in parallel. There can be four different kinds of parallelism:

1. only sibling,
2. only parent-child,
3. parent-child and sibling,
4. no parallelism (i.e., sequential execution).

In the first case, where only sibling parallelism is allowed, parent stops its execution while its children are running concurrently. In the second case, only parent-child parallelism is allowed where the parent and a child run concurrently while the other children wait. In the third case, all transactions in the hierarchy can run in parallel. In the fourth case, we have no parallelism at all (i.e., transactions in the hierarchy are executed sequentially) [23]. In our model, we will assume parent-child and sibling parallelism since it provides us with the most flexible model of parallelism.

When transactions are executed concurrently, serializability is used as the correctness criterion, and it is ensured by using a concurrency control protocol. A child transaction can potentially access any object in the database. When a subtransaction commits, the objects modified by it are delegated to its parent transaction. In our execution model, we used a locking protocol for concurrency control in nested transaction execution. The protocol is described in the next section.

In nested transactions, ACID properties (i.e., atomicity, consistency, isolation, and durability) are valid for top-level transactions, but only a subset of them holds for subtransactions [23]. A subtransaction may commit or abort independent of other transactions. Aborting a subtransaction does not affect other transactions outside of its hierarchy, hence they protect the outside world from internal failures. If we had packed all subtransactions into one big flat transaction then we would have to abort the whole transaction.

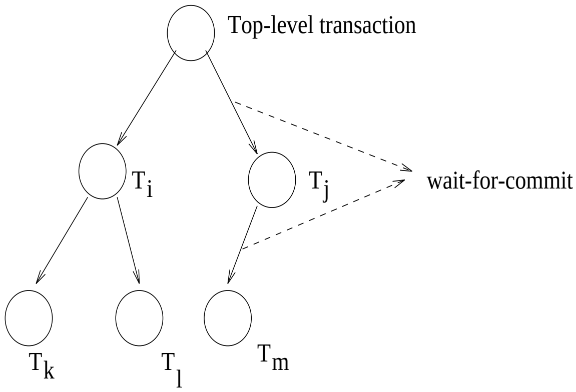

Fig. 3: Wait-For-Commit Relations.

Deadlock situation is a well known problem in the database field [3]. Well established deadlock detection methods for centralized systems are applied to the distributed databases as well [25]. One method of detecting the deadlock situation for flat transactions is to trace the *wait-for graph* maintained by the transaction manager of the system [3]. *Wait-for graph* for a group of transactions is constructed by identifying the transactions waiting for each other. For nested transactions, identifying the deadlocks is more complicated than flat transactions since deadlocks may arise among subtransactions in the same transaction hierarchy as well as subtransactions belonging to different transaction hierarchies. Parent-child relationships introduce new possibilities of deadlocks as well. Deadlock detection issues for nested transactions is briefly discussed in [23], and a deadlock detection method for distributed nested transactions is proposed in [33]. To the best of our knowledge, there is no other work proposing a different method for detecting deadlocks in nested transactions. In [33], Marta Rukoz considers deadlocks that may occur in a single transaction hierarchy. Since the communication overhead is high in a distributed system, the proposed method compresses the wait-for graph by only including the representative edges instead of including all kinds of wait-for relationships. Deadlock detection is simplified to the expense of maintaining a compact wait-for graph for a transaction hierarchy.

Deadlock detection for nested transactions is different from the one for flat transactions in that, there are some other *wait-for* relations besides the *wait-for-lock* relation. One possible *wait-for* relation associated with nested transactions is *wait-for-commit*; i.e., a parent transaction should wait for all its children to finish their execution. A *wait-for-commit* graph is illustrated in Figure 3 for a transaction hierarchy where the top-level transaction spawns subtransactions T_i and T_j, T_i spawns subtransactions T_k and T_l, and finally T_j spawns subtransaction T_m.

Another possible *wait-for* relation for nested transactions is *wait-for-lock* relation. There are two types of *wait-for-lock* relation; *wait-for-retained-lock* and *wait-for-held-lock*. In flat transaction model, when a transaction commits, all the locks it holds are released, and transactions waiting for one or more of those locks can be unblocked immediately provided that they are not waiting for any other transaction. In nested transaction model, locks held by a subtransaction are not released immediately after it commits, but they are inherited by the parent transaction and kept in *retain* mode which causes *wait-for-retained-lock* relations to occur. Examples of *wait-for-held-lock* and *wait-for-retained-lock* relations are illustrated in Figures 4(a) and 4(b).

In Figure 4(a) we see that transaction T_j is holding a write lock on object Obj_i at the time when transaction T_i requests a write lock on the same object. The arc labeled as *wait-for-held-lock* depicts this waiting situation. Following the commit of transaction T_j, all the locks that belong to T_j are inherited by the parent transaction, namely T_k, in retain mode. Transaction T_i still have to

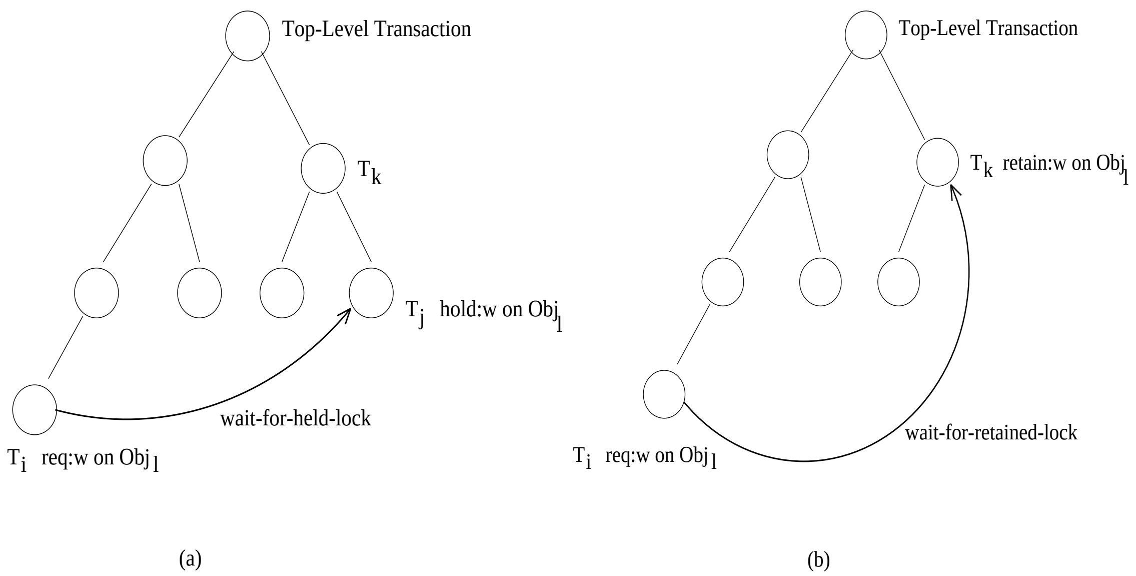

Fig. 4: Wait-For-Lock Relations; (a) Before the Commit of T_j, (b) After the Commit of T_j.

wait until the commit of T_k. T_i should wait until the first common ancestor of T_i and T_j inherits the lock on object Obj_l, which is the top-level transaction in this case.

Both *wait-for-commit* and *wait-for-retained-lock* relations should be taken into consideration in addition to the classical *wait-for* relation, for deadlock detection in nested transactions.

Deadlocks that may occur among top-level transactions can be resolved as in flat transactions which means aborting a top-level transaction, or finer granularity deadlock detection may be utilized if top-level transactions can exchange information. In finer granularity deadlock detection, subtransactions that cause the deadlock among top-level transactions may be identified and the deadlock can be resolved by aborting only some of those sub-transactions instead of aborting a whole top-level transaction. Finer granularity deadlock detection is difficult in heterogeneous systems but can be achieved in centralized systems.

Recovery of nested transactions is similar to the recovery of flat transactions. Standard recovery algorithms like versioning or log-based recovery can be used. Log-based recovery for nested transactions is discussed in [28, 32]. [32] introduces a model called ARIES/NT and this has several advantages over the recovery model provided in [28] . The biggest drawback of the recovery model of [28] is that it does not use Compensation Log Records (CLRs) which are necessary for performance reasons. A detailed description of CLRs is provided in [32]. OpenOODB, on which our execution model has been built, uses Exodus as storage manager whose recovery component is implemented based on ARIES [26], and ARIES/NT is provided for nested transactions as an extension to ARIES.

3.2. Nested Transactions for Concurrent Rule Execution

Since concurrency is a very important functionality that a database must provide, we provide an evaluation of the previously proposed execution models in terms of inter and intra-rule concurrency. The execution models proposed in [18, 20, 38] allow only sequential rule execution by user or system defined priorities and therefore they have neither inter nor intra-rule concurrency. In [18], authors discuss coupling modes to determine the transaction boundaries as well while supporting decoupled rule execution, but rules are still executed sequentially. In [2, 6, 14, 15, 16], concurrent rule execution and nested transaction paradigms are discussed. In [6], importance of nested transactions for concurrent rule execution is stressed but not discussed in detail. In [2], a rule execution model utilizing nested transactions is proposed but concurrency control is left as a future work and different kinds of parallelisms between parent-child and siblings are not mentioned. Authors in [15] propose nested transactions as a rule execution model for active databases. Nested transactions are extended with immediate, deferred, and decoupled coupling modes, where only sibling parallelism is allowed. Coupling modes facilitate intra-rule concurrency and nested transactions provide a control mechanism for inter-rule concurrency. Priorities are used to limit the concurrency among rules. Our execution model utilizes extended parallel nested transactions together with different coupling modes. Sequential causally dependent, and exclusive causally dependent coupling modes are supported in addition to the causally dependent and detached coupling modes. Full inter-rule and intra-rule concurrency is supported by including parent-child parallelism together with sibling parallelism. Our execution model supports sequential rule execution by utilizing priorities.

Nested transactions are proposed as a rule execution model for various active database systems (e.g., [2, 13, 15]). In [2], authors assert that nested transactions are compatible with object oriented systems in that, method invocations can be performed in a subtransaction in a natural way. In [13, 15, 16], authors explain nested transactions with only sibling parallelism as a rule execution model. In the execution model presented in our paper, nested transactions with both parent-child and sibling parallelism are utilized.

The nested transaction model implicitly assumes that subtransactions are spawned in immediate mode. In our execution model, transactions may spawn subtransactions in any coupling mode specified by the system. Each rule is encapsulated in a transaction. When a rule r_1 fires another rule r_2, then depending on the coupling mode, r_2 is encapsulated in another (sub)transaction and executed in the specified coupling mode. If the coupling mode is immediate or deferred, then r_2 is executed as a subtransaction of r_1. If the coupling mode is one of detached, sequential causally dependent, detached causally dependent, or exclusive causally dependent modes, then r_2 is executed as a top-level transaction. The overall structure of the currently executing rules in the system forms a forest consisting of trees whose roots are the rules fired in one of the detached coupling modes. As stated earlier, both parent-child and sibling parallelism are allowed which provides us with the maximum concurrency among subtransactions. Top-level transactions are executed in parallel. All nested transaction semantics applies among the individual rules in the nested transaction tree. Abort and commit dependencies among the top-level transactions are enforced by the transaction manager.

In addition to the coupling mode and the tuple- or set-oriented semantics, priority information also affects the order of rule execution. Typically, expert systems assumed a conflict resolution strategy which produced a sequential order for executing *eligible* rules one at a time. In contrast, DBMSs assume no priority among transactions/subtransactions and hence execute *eligible* transactions in an interleaved order to maximize some metric (such as throughput). The correctness criteria is the serialization of transactions. The serializability is provided through strict two-phase locking protocol. For the immediate and deferred coupling modes, as well as the detached causally dependent and detached sequentially dependent coupling modes, a subtransaction $S1$ which executes the condition of a rule R does not release its locks until the subtransaction $S2$ executing the action of the same rule commits. Therefore, a subtransaction $S2'$ executing the action of another rule R' is not allowed to modify the data locked by $S1$ until the commit of $S2$ (i.e, until rule R terminates). So, it is not possible that an executing rule can be untriggered by another rule. Also, we should note that, the rules that are allowed to run concurrently are the ones that have the same priority. If there is any dependency between any two rules that requires a strict execution order between the rules, such rules are assigned different priorities which determine the order of their execution. We assume that such dependencies are identified through a static analysis of the rule set prior to the execution of rules. Note that serial execution automatically satisfies the serializability criteria.

In fact, a combination of the above two alternatives is likely to meet the requirements of applications. This entails that the system be capable of supporting both serial execution using *a* conflict resolution strategy and concurrent execution based on the serialization criterion of correctness. Rules, in our execution model, can be executed both sequentially and concurrently. Rules are associated with priorities in order to be executed sequentially. Several rules can have the same priority value. Rule scheduler uses the priority information to either schedule the fired rules sequentially by assigning different priorities to different rules, or concurrently by assigning the same priority to multiple rules. Correctness of rules executing sequentially or concurrently is guaranteed by the semantics of nested transactions. When a rule is triggered, the time to start the execution of that rule is determined by the semantics of the coupling mode involved. When multiple rules are fired together, the order of execution among those rules can be determined by assigning priorities.

3.3. Concurrency Control in Rule Execution

The concurrency control algorithm used in our execution model is based on the notion of nested concurrency control. Harder and Rothermel [23] extended Moss's nested transaction model to contain downward as well as upward inheritance of locks. We have employed in our model the locking protocol provided in [23]. The protocol is composed of the following locking rules:

- Rule 1: Transaction T may acquire a lock in mode M or upgrade a lock it holds to mode M if:
  - no other transaction holds the lock in a mode that conflicts with M, and
  - all transactions that retain the lock in a mode conflicting with M are ancestors of T.

  A transaction *holds* a lock on an object if it has the right to access the locked object in the requested mode. In contrast, a transactions *retains* a lock on an object to control the access of the transactions outside the hierarchy of the retainer and the object cannot be accessed by the transaction retaining the lock.

- Rule 2: When subtransaction T commits, the parent of T inherits T's locks (held and retained). After that, the parent retains the locks in the same mode as T held or retained them before.

- Rule 3: When a top-level transaction commits, it releases all the locks it holds or retains.

- Rule 4: When a transaction aborts, it releases all the locks it holds or retains. If any of its superiors hold or retain any of these locks, they continue to do so.

- Rule 5: Transaction T, holding a lock in mode M, can downgrade the lock to a less restrictive mode, M'. After downgrading the lock, T retains it in mode M.

These locking rules can be used with different types of coupling modes. A transaction spawned in detached causally dependent mode should be able to use the spawning transaction's locks in the same way as a subtransaction spawned in immediate or deferred mode. Since the transaction spawned in detached causally dependent mode should abort if the spawning transaction aborts, it can use its parent's locks without causing any problem in the recovery. Concurrency among top-level transactions can be increased if this information can be exchanged among those transactions. This is not possible in our system since top-level transactions are independent and execution of those transactions is controlled by OpenOODB. Both shared and exclusive lock modes are available to transactions in our execution model as described in Section 4.2.1 .

3.4. A Formal Discussion of Our Rule Execution Model

ACTA is a transaction framework that can be used to formally describe extended transaction models [12]. Using ACTA, we can specify the interactions and dependencies between the transactions in a model. ACTA characterizes the semantics of interactions (1) in terms of different types of dependencies between transactions (e.g., commit dependency and abort dependency) and (2) in terms of transactions effects on objects (their state and concurrency status, i.e., synchronization state) [12]. We are going to utilize two basic blocks of the ACTA framework which are *history*, and *dependencies* between transactions. *History*, denoted by H, represents the concurrent execution of a set of transactions and contains all the events invoked by those transactions, also indicating the partial order in which these events occur. Invocation of an event ϵ by a transaction t is denoted by ϵ_t. There are three possibilities that can affect the occurrence of an event:

1. an event ϵ can occur only after the occurrence of another event $\epsilon^{'}$ (denoted as $\epsilon^{'} \rightarrow \epsilon$),

2. an event ϵ can occur only if a condition c is true (denoted as $c \Rightarrow \epsilon$),

3. a condition c can require the occurrence of an event ϵ (denoted as $\epsilon \Rightarrow c$).

Standard dependencies defined in ACTA [11] which we have used for specifying our execution model are:

- Commit Dependency (denoted as $t_j\ CD\ t_i$). If transactions t_i and t_j both commit then t_i should commit before t_j. This can be shown axiomatically as:
  $Commit_{t_j} \in H \Rightarrow (Commit_{t_i} \in H \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j}))$.

- Abort Dependency (denoted as $t_j\ AD\ t_i$). If t_i aborts then t_j should also abort:
  $Abort_{t_i} \in H \Rightarrow Abort_{t_j} \in H$

- Weak-Abort Dependency (denoted as $t_j\ WD\ t_i$). If t_i aborts and t_j has not yet committed, then t_j aborts:
  $Abort_{t_i} \in H \Rightarrow (\neg(Commit_{t_j} \rightarrow Abort_{t_i}) \Rightarrow (Abort_{t_j} \in H))$

- *Exclusion Dependency* is denoted by $t_j\ ED\ t_i$, and ensures that if t_i commits , then t_j must abort. We can state this formally as:
  $Commit_{t_i} \in H \Rightarrow Abort_{t_j} \in H$.

These dependencies may be the result of the structural properties of transactions. For example, in nested transactions child transactions are related to their parent by commit and weak-abort dependencies.

All coupling modes except the *deferred mode* and *sequential causally dependent mode* can be specified easily using the standard dependencies of ACTA (i.e., commit dependency, abort dependency, weak abort dependency, and exclusion dependency described above). For the deferred mode, we need to specify a *cycling execution method*, which can be stated as follows:

Deferred transactions are executed in cycles at the end, but just before the commit of the transaction that spawned them. Cycling execution can start only in a top-level transaction or a subtransaction spawned in immediate mode since deferred subtransactions spawned by another deferred transaction are executed in the next cycle after the commitment of their parent. Subtransactions spawned by a deferred transaction in immediate mode are executed immediately, which deviates from the standard specification of the "deferred execution in cycles".

The specification of all the coupling modes we considered, including the deferred mode and sequential causally dependent mode is provided in the following.

Coupling Modes The coupling modes we considered in our execution model are listed below:

- *immediate mode*: It has the same semantics as the creation of a subtransaction in standard nested transaction model. Spawning of an immediate subtransaction is denoted by the primitive *Spawn_Imm*.

- *detached mode*: It has the same semantics as the creation of top-level transactions in the standard nested transaction model. There are no dependencies between the spawning and spawned transaction. Spawning of a detached transaction is denoted by the primitive *Spawn_Detached*.

- *detached causally dependent mode*: In this mode, spawned transaction aborts if the parent aborts, therefore there is an abort dependency between the spawning transaction and spawned transaction. Spawning of a transaction in this mode is specified by the primitive *Spawn_Caus*.

- *sequential causally dependent mode*: It specifies that a child transaction cannot start its execution until its parent commits. This can be enforced by a $Sequential\_Dependency(SQD)$ which is provided as an extension to the standard ACTA dependency set and can be stated formally as:

  $$t_j SQD t_i \Leftrightarrow ((Begin_{t_j} \in H) \Rightarrow (Commit_{t_i} \rightarrow Begin_{t_j}))$$

  The primitive *Spawn_Seq* indicates that a subtransaction is spawned in this mode.

- *exclusive causally dependent mode*: It is denoted by the primitive *Spawn_Exc* and it ensures that the spawned transaction commits only if the spawning transaction aborts. This can be enforced by using a standard ACTA dependency, namely the *Exclusion Dependency* between the spawning and spawned transactions.

- *deferred mode*: It is denoted by the primitive *Spawn_Def*. Assume that t_0 is a top-level transaction, t_p is a top-level or subtransaction, and t_c is a child transaction spawned by t_p in deferred mode.

  Case 1: t_p is a top-level transaction or a subtransaction spawned in immediate mode, i.e., t_c is going to be executed in cycle-0. In this case, t_c is executed just before the commit of t_p after all other operations of t_p are completed; i.e., all operations of t_p precede all operations of t_c.
  Case 2: t_p is a transaction spawned in deferred mode. This means that t_c is spawned during a cycle. Then, every operation performed by t_c should succeed all the operations of t_p and the operations of siblings of t_p that are spawned in deferred mode (i.e., executed in the same cycle).

There is an ambiguity in the method described in [24] for the cycling execution of rules fired in deferred coupling mode. If a rule is fired in deferred mode by a transaction during the execution of a cycle, it is executed in the next cycle; but if a rule is fired in deferred mode by a transaction which has been fired in immediate mode then the fate of this transaction, i.e., whether it will be deferred to the next cycle or it will be executed in another execution cycle is left unspecified. We chose to execute these kinds of rules in another cycle before the commit point of the immediate rule.

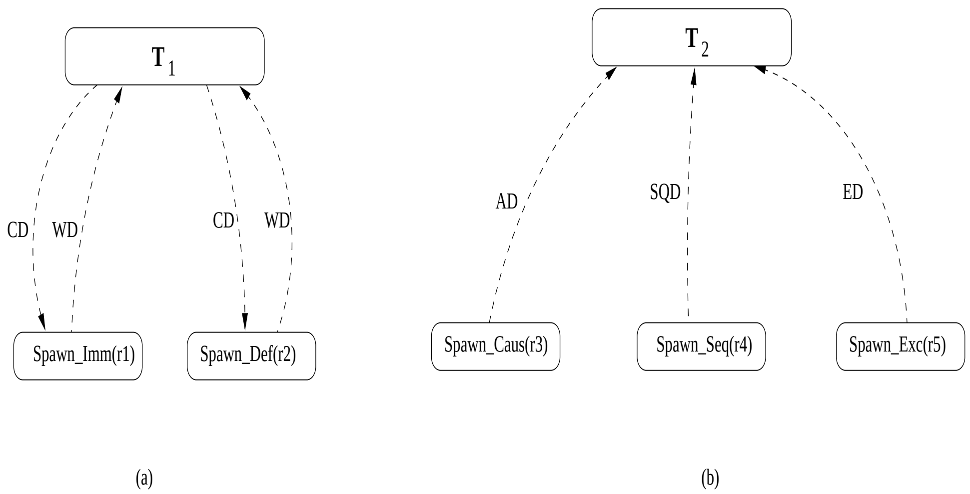

Fig. 5: Dependencies Induced by Different Coupling Modes

Dependencies induced by different coupling modes between event and condition are illustrated in Figure 5. In Figure 5(a), transaction T_1 causes an event to be raised that triggers r_1 in immediate mode and r_2 in deferred mode. These rules are executed as subtransactions of T_1. Subtransactions executing rules r_1 and r_2, if not committed yet, should abort if T_1 aborts (WD). Transaction T_1 should wait for the commit of r_1 and r_2 (CD). In Figure 5(b), transaction T_2 causes an event to be raised that triggers rules r_3, r_4, and r_5 in detached causally dependent, sequential causally dependent and exclusive causally dependent modes respectively. These rules are executed as top-level transactions. In this case T_2 does not have any dependencies on any of r_3, r_4, or r_5 while r_3 should abort if T_2 aborts (AD), r_4 cannot start its execution before T_2 commits (SQD), and r_5 commits only if T_2 aborts (ED).

Parallel nested transactions used in our rule execution model allows parent child concurrency. Therefore rules can be executed concurrently with the transaction that spawned them depending on the coupling mode between their event and condition. For example in Figure 5(a), rules r_1 and r_2 can be executed concurrently with T_1. Parallel nested transactions support sibling concurrency as well, allowing the rules triggered by the same transaction to execute concurrently among each other depending again on the coupling mode between their event and condition. In Figure 5(b), rule r_3 can be executed concurrently with rule r_5 while r_4 has to wait for the commit of T_2 since there is a sequential causally dependent coupling mode between the event and condition of r_4. Executing rules concurrently with the spawning transaction or executing rules concurrently among each other is called inter-rule concurrency. Intra-rule concurrency is achieved when components of a rule are executed concurrently. In our execution model we support intra-rule concurrency in the condition-action level, i.e., condition and action of a rule may execute concurrently when necessary. In Figure 6, condition, C_1 of a rule spawns the action A_1 of the rule in detached causally dependent mode, and C_1 executes concurrently with A_1. Figure 6 illustrates the nested and multiple firing of rules as well. Rules r_6 and r_7 are fired as a result of the execution of A_1 and executed as subtransactions of A_1, which is nested rule firing; and firing of rules r_6 and r_7 at the same time by the same event is an example of multiple firing of rules.

In our execution model, we consider the coupling modes between the event and condition, and also the condition and action. We can give the option of defining the coupling mode between the condition and action to the user, where the user can select immediate or sequential causally dependent coupling mode. Other coupling modes would not make sense due to the semantics of the relation between the condition and action. In immediate mode, condition evaluation is followed by execution of the action only if the condition evaluates to true. In sequential causally dependent mode, action execution starts before condition evaluation is completed. This improves the concurrency in a system in case there exist abundant resources in the system, which is a reasonable assumption due to the continuous decrease in the prices of system resources. The transaction in which the action is executed can commit only if the condition commits and returns true. The performance impact of the sequential causally dependent coupling mode between condition and action needs further investigation.

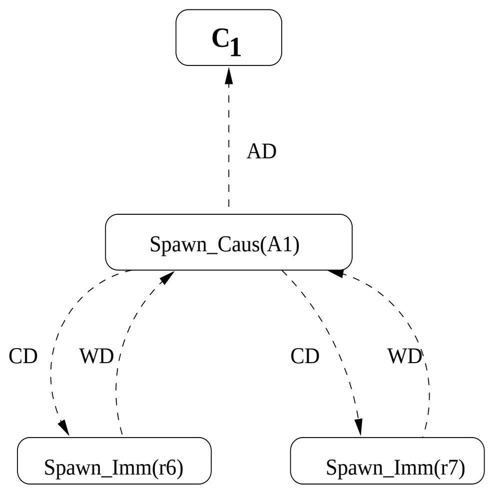

Fig. 6: Intra Rule Concurrency and Nested Firing of Rules

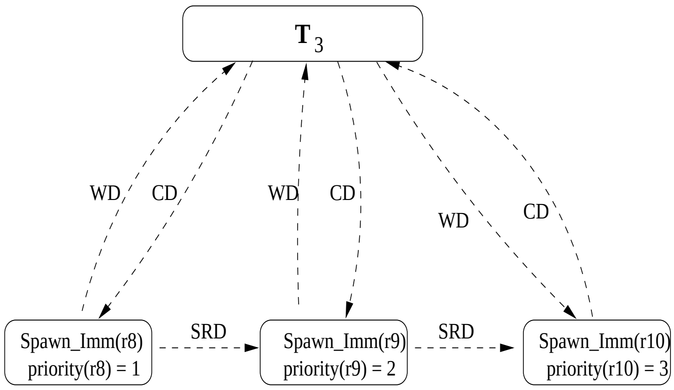

Fig. 7: Dependencies Induced by Priorities

Priorities

In order to provide priorities in ACTA primitives, we define $priority(t)$ to specify the priority of transaction t. Priorities are assumed to be assigned by the user. Our priority scheme assumes that priorities are taken into consideration only for the transactions which do not have any dependencies among each other since dependencies are considered as implicit priorities. As we stated earlier, deferred transactions spawned by the same transaction are independent of each other. So we can use priorities to schedule them by enforcing the operations of a deferred transaction with a higher priority to precede the operations of a lower priority transaction spawned in deferred mode by the same transaction. Similarly, we can use priorities in scheduling subtransactions spawned in immediate mode by the same transaction by enforcing that the operations of a high priority transaction are scheduled before the operations of a lower priority transaction. In order to enforce that in the specification, we introduce *Serial_Dependency(SRD)* which can be stated formally as:

$$t_j SRD t_i \Leftrightarrow ((Begin_{t_j} \in H) \Rightarrow (Commit_{t_i} \rightarrow Begin_{t_j}) \vee (Abort_{t_i} \rightarrow Begin_{t_j}))$$

Dependencies induced by assigned rule priorities are depicted in Figure 7 where multiple firing of rules r_8, r_9, and r_{10} occur due to an event raised by the execution of transaction T_3. Serial dependencies (SRD) are formed among the rules according to previously assigned priorities.

A full set of the axiomatic definitions of our rule execution model is provided in [34].

4. IMPLEMENTATION OF NESTED TRANSACTIONS FOR RULE EXECUTION

4.1. Previous Work

Although various implementations of the nested transaction model have been provided to date, to the best of our knowledge, implementation of concurrent rule execution through nested transactions has not been attempted for any ADBMS.

One implementation of nested transactions has been performed on the Eden Resource Management System (ERMS) [31]. In ERMS, transaction managers are composed hierarchically; i.e., for each subtransaction there is a corresponding transaction manager. For ensuring the serializability, two-phase locking is used, and a version-based recovery is used for the recovery of sub-transactions. In [17], the implementation described focuses on nested transactions for client workstations of an object oriented DBMS(OODBMS).

Nested transactions have also been implemented for supporting parallelism in engineering databases [22]. That particular implementation supports both parent-child and sibling parallelism as in our implementation. Nested transactions are proposed for transaction management in multidatabase systems and for managing method invocations in disconnected objects as well [10, 19].

4.2. Implementation

We implement nested transactions by extending the flat-transaction semantics of OpenOODB [37]. OpenOODB is an open object oriented DBMS that can be extended by special constructs called *sentries*. In our implementation, a component architecture method is used instead of sentries; i.e., a new component is added without significantly modifying the existing ones. Our first task was to construct the object relationships of OpenOODB by examining the class declarations. In Figure 8, the whole OpenOODB object relationships diagram is given.

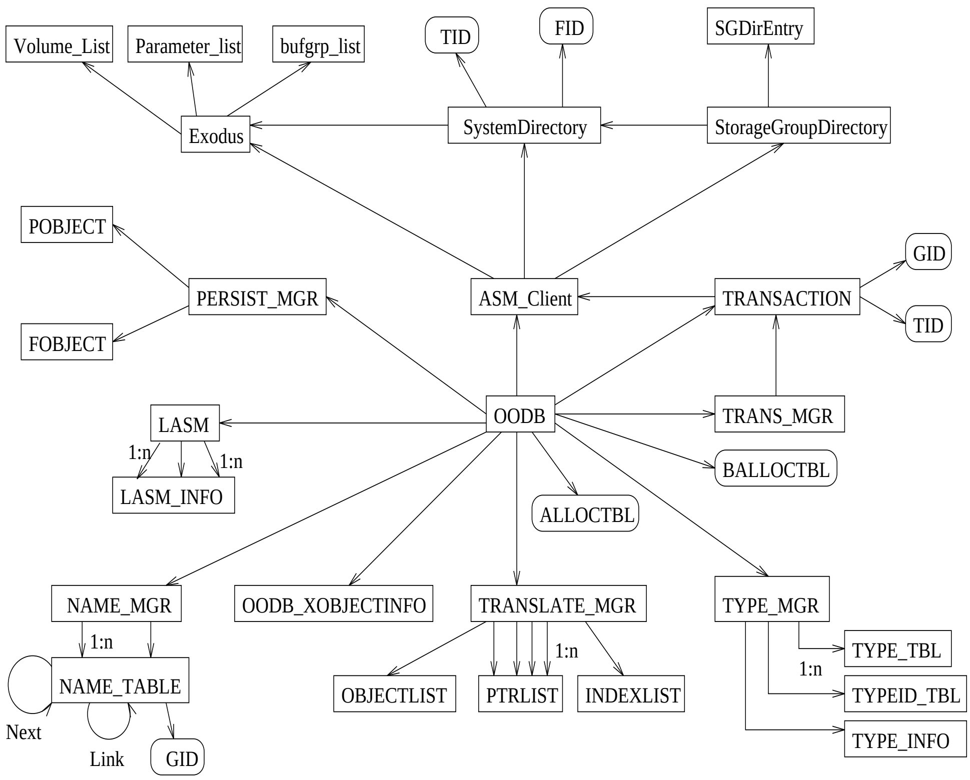

Fig. 8: OpenOODB Object Relationship Diagram.

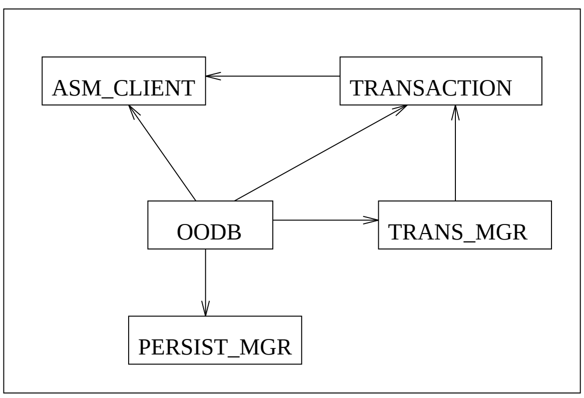

Fig. 9: Related Object Relationships

Among the components illustrated in this diagram, the ones that need to be considered for our implementation are isolated. These isolated components are shown in Figure 9. In this figure, we see that the main OpenOODB object *OODB* has a pointer to each of four objects namely

PERSIST_MGR, *TRANS_MGR*, *TRANSACTION*, and *ASM_CLIENT* which means that whenever an instance of an object of type *OODB* is created, its constructor creates instances of *PERSIST_MGR*, *TRANS_MGR*, *TRANSACTION*, and *ASM_CLIENT* objects. Furthermore, the constructor of *TRANS_MGR* object creates an instance of *TRANSACTION* object which is also used directly by *OODB*.

To give a flavor of how a transaction is started and objects are fetched from the database, we give a sample application of OpenOODB in Figure 10.

```
OODB *p_oodb;
My_Class *my_obj = new My_Class;
My_Class *tmp_obj;
char *obj_name = "obj1";
main()
{
        p_oodb →beginTransaction();
        /* make the object persistent and give a name to it */
        my_obj →persist(obj_name);
        p_oodb →commitTransaction();

        p_oodb →beginTransaction();
        /* fetch the object with the given name */
        p_oodb →fetch(obj_name);
        p_oodb →commitTransaction();
}
```

Fig. 10: A Simple OpenOODB Application.

As can be seen from the figure, an OpenOODB main object *p_oodb* is created which provides us with an interface to OpenOODB. A transaction is started by using *p_oodb* $\rightarrow$ *beginTransaction* and committed by *p_oodb* $\rightarrow$ *commitTransaction*. Abortion of a transaction is achieved by *p_oodb* $\rightarrow$ *abortTransaction*. Objects are made persistent by *my_obj* $\rightarrow$ *persist*() and are fetched from the database by the *p_oodb* $\rightarrow$ *fetch*(...). OpenOODB fetch operation does not give the flexibility of specifying the lock mode but acquires a default *READ* lock from EXODUS storage manager. To provide the application programmer with more flexibility, we decided to modify the fetch operator of OpenOODB so that it takes the locking mode as a parameter. As a second stage, nested transaction primitives:

- *spawn_sub_transaction*
- *commit_sub_transaction*
- *abort_sub_transaction*

are added to the transaction manager of OpenOODB (i.e., *TRANS_MGR* in Figure 9). Finally, a Lock Manager is implemented to support the nested transaction primitives that are added to the transaction manager. Among the nested transaction primitives, only the *spawn_sub_transaction* takes parameters. The first parameter of it is the name of the function where the subtransaction is written in, the second one is the spawn-mode. Spawn-mode specifies the coupling mode between the parent and child. For our nested transaction component, we implemented the immediate and deferred coupling modes. Detached coupling modes are handled by the rule manager of the ADBMS. Nested transaction primitives are used for the rule execution by starting the execution of rules inside subtransactions. When a rule is fired, then a top-level transaction or a subtransaction is created depending on the coupling mode between the event and condition of the rule.

If the coupling mode is immediate or deferred then a subtransaction is created. The primitive *spawn_sub_transaction* is used for creating a subtransaction. Parent of the spawned subtransaction is the transaction that caused the rule to be fired. If the coupling mode is one of the detached coupling modes, then a top-level OpenOODB transaction is created in a new process and the rule is executed in that transaction. The primitives *commit_sub_transaction* and *abort_sub_transaction* indicate the successful and unsuccessful termination of a rule executed in a subtransaction, respectively. OpenOODB transaction commands *commitTransaction* and *abortTransaction* indicate the successful and unsuccessful termination of a rule executed in an OpenOODB top-level transaction, respectively. An executing rule may cause other rules to be fired which is called nested rule firing. Nested rule firing is handled by the nested transactions in a natural way since an arbitrary number of transactions may be initiated inside a transaction. As a result, rules fired by the execution of another rule are started as subtransactions or top-level transactions depending on the coupling mode between the event and condition. Firing of multiple rules is handled by spawning multiple subtransactions at the same time and executing them concurrently, or sequentially by assigning priorities. Rules may access the objects residing in the database by the *fetch_object* method. Concurrent execution of rules running as subtransactions is handled by the nested transaction component, while rules executing concurrently as top-level transactions are managed by OpenOODB via Exodus.

Using Figure 9 we can describe where our lock manager fits in the object relationships diagram. In that figure, the main object of OpenOODB (i.e., $OODB$), points to a $TRANS\_MGR$ object which has a $TRANSACTION$ object. And the $TRANSACTION$ object has a $LOCK\_MANAGER$ object; i.e., the constructor of the transaction object creates the $LOCK\_MANAGER$ object which can be accessed by $OODB$. This way the constructs implemented in $LOCK\_MANAGER$ and $TRANS\_MGR$ can be used by the application via the OpenOODB interface object, $OODB$.

$LOCK\_MANAGER$ has two main data structures, namely the *Lock_Table* and the *Transaction_Table*. *Lock_Table* is a hash table that is used to keep the lock information of objects that have previously been fetched by a transaction in the transaction hierarchy. *Lock_Table* is hashed by the object name, and given an object, we can reach all the transactions that have a lock on this object with any mode. *Transaction_Table* keeps the transaction hierarchy, wait-for graph and the lock information of the subtransactions. The hash tables for the lock and transaction management are interconnected, that is, we can reach the objects that are held by a transaction given its transaction identifier. This provides us with efficient abort and commit of subtransactions. *Transaction_Table* is hashed by transaction identifiers(*tid*). Given the *tid* of a transaction†:

- We can reach all the objects held by that transaction in any hold and lock mode. The hold mode of a lock can be either *hold* or *retain*, and the lock mode can be either *READ* or *WRITE*.
- We can reach the transactions for which the transaction with *tid* is waiting.
- We can reach the transactions waiting for the transaction with *tid*.
- We can reach all the children and ancestors of the transaction with *tid*.

If we take a snapshot of the system, we see top-level transactions executing concurrently. Each top-level transaction represents a nested transaction hierarchy. In Figure 11 we see the tasks performed at the beginning and at the end of a top-level transaction and a subtransaction. Each OpenOODB object has a transaction manager object associated with it. When a top-level transaction is started, this transaction manager object creates an instance of the transaction object which further creates the lock manager. Lock manager handles the concurrency control issues and the maintenance of the transaction hierarchy via transaction and lock tables. Each nested transaction hierarchy has a different transaction manager.

†In the remaining part of this section, we use the term transaction to denote both top-level transactions and subtransactions.

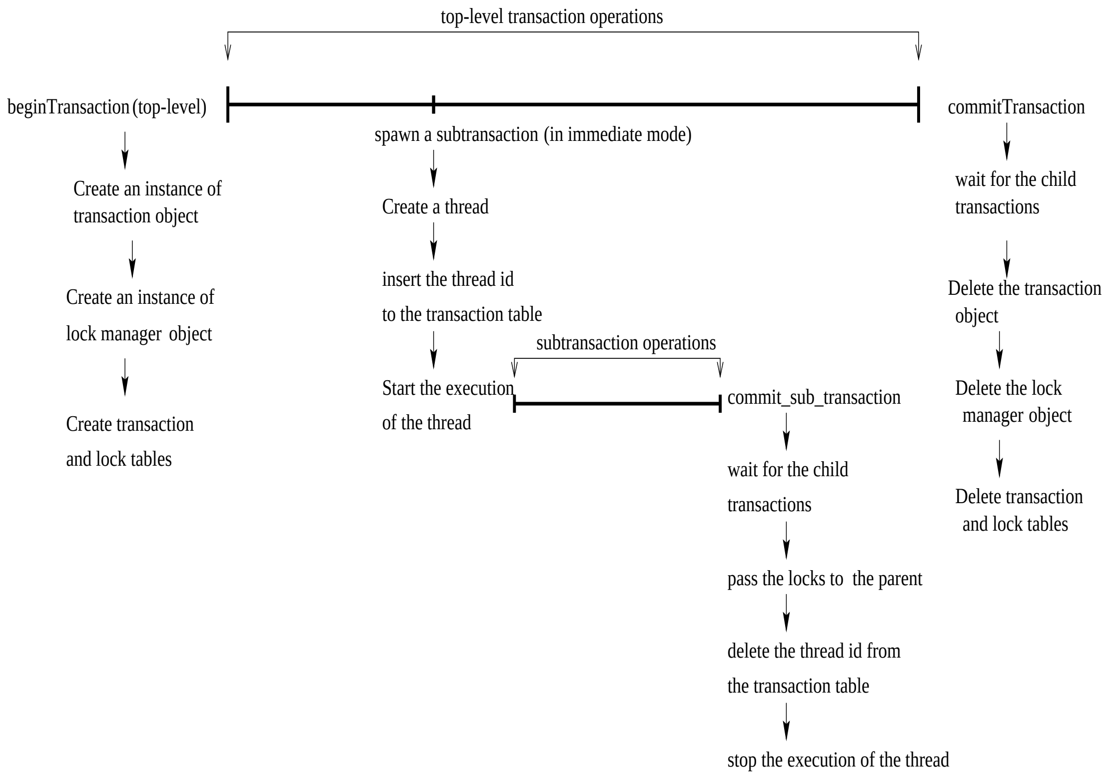

Fig. 11: Dynamics of Top-Level and Subtransaction Creation

For the parallel execution of subtransactions, Solaris threads are used [36]. Solaris is a fully functional distributed operating and windowing environment [35]. Thread is a sequence of instructions executed within the context of a process. Traditional Unix process contains a single thread of control. Solaris provides us with Multi-threaded Programming. Multi-threading separates a process into many execution threads each of which runs independently.

Advantages of Multi-threading can be listed as:

- overlap in time, logically separating tasks that use different resources,
- sharing the same address space,
- providing cheap switching among threads.

Primitive	**Explanation**
$thr\_create()$	create a thread
$thr\_self()$	return the thread identifier of the calling thread
$thr\_suspend()$	block the execution of a thread
$thr\_continue()$	unblock a thread
$thr\_kill()$	send a signal to a thread
$thr\_exit()$	terminate a thread
$thr\_join()$	wait for the termination of a thread

Table 1: Thread Primitives Used.

Mutual exclusive locks are used to control the concurrent access of different threads to the shared data structures.

Thread primitives used in our implementation are listed in Table 1. Using *thr_create*() we execute a given function in a thread. In our implementation, subtransactions are defined as functions in a specified format and are executed concurrently using the *spawn_sub_transaction* primitive provided by our implementation of nested transactions. In Figure 12, the *spawn_sub_transaction* primitive executes the transaction, embedded inside a function, in a thread using *thr_create*(). Threads are created in suspended mode so that the necessary information is inserted into the *Transaction_Table*. Since the *tid*s are unique within a Unix process, and top-level transaction boundaries do not exceed the process boundaries, it was very convenient for us to define the *tid*s as the transaction identifiers. This way we do not need to pass the transaction identifier to the subtransaction as a parameter. Subtransactions can access their *tid*s by calling the thread library function *thr_self*(). This function returns the thread identifier of the calling thread, i.e., the *tid*. When an OpenOODB top-level transaction is created, the *tid* is also inserted into the transaction table for the sake of completeness of the transaction hierarchy.

```
// create an OpenOODB main object
OODB *p_oodb;
void *sub2(void *res)
{
    // lock the object with name obj3 in WRITE mode
    int rc = p_oodb →fetch_object("obj3",WRITE);
    // in case of an error, abort the subtransaction
    // otherwise, commit the subtransactions
    if ( rc == ERROR )
      p_oodb → sub_abort();
    else
      p_oodb →sub_commit();
}


void *sub1(void *res)
{
    // lock the object with name obj1 in READ mode
    p_oodb →fetch_object("obj1",READ);
    // create a subtransaction in IMMEDIATE mode
    p_oodb →spawn_sub_tr(sub2,IMMEDIATE);
    // commit the subtransaction
    p_oodb →sub_commit();
}


main()
{
    // start an OpenOODB transaction
    p_oodb →beginTransaction();
    // spawn a subtransaction in IMMEDIATE mode
    p_oodb →spawn_sub_tr(sub1,IMMEDIATE);
    // commit the OpenOODB transaction
    p_oodb →commitTransaction();
}
```

Fig. 12: Sample OpenOODB Application Using Nested Transactions.

Since all the subtransactions in the transaction hierarchy can access the data structures in the $LOCK\_MANAGER$, we define a global mutex variable. This way, the critical sections of the methods modifying the *Lock_Table* and *Transaction_Table* are wrapped by *mutex_lock* and $mutex - unlock$.

When we look at the data structures, we observe that there are linked lists belonging to both *Transaction_Table* and *Lock_Table* which means that deletions and insertions of new blocks to those lists take a long time. At this point, we made an optimization by implementing our own memory management via keeping lists of deleted blocks so that they can be used efficiently whenever they are needed. Another optimization was to extend our component with a sort of garbage collection; i.e., when we want to delete a block from the *Lock_Table* or *Transaction_Table*, we do not delete it physically but mark it as deleted. This technique makes the usage of doubly linked lists unnecessary. Garbage collection is performed during the searches in the *Lock_Table*.

4.2.1. Implementation of the Locking Protocol and Deadlock Detection for Nested Transactions

When a transaction requests a lock on an object it specifies the locking mode as well by providing the *fetch_object* method with the *lock_type* parameter. Allowed lock-modes are *READ* and *WRITE*.

We implemented the locking protocol described in Section 3.3 considering both *READ* and *WRITE* locks (i.e., *shared* and *exclusive* locks respectively).

As an example, consider the nested transaction structure in Figure 13(a). If T_i retains a *WRITE* lock on an object O_i and no other transaction inside the sphere S_i (i.e., T_j, T_k, T_l and T_m) has any lock on O_i, then all the transactions inside S_i can acquire a *READ* or a *WRITE* lock on O_i. If T_j acquires a *WRITE* lock on O_i, then no other transaction (including the ones inside the sphere S_j in Figure 13(b)) can acquire a *READ* or a *WRITE* lock on O_i.

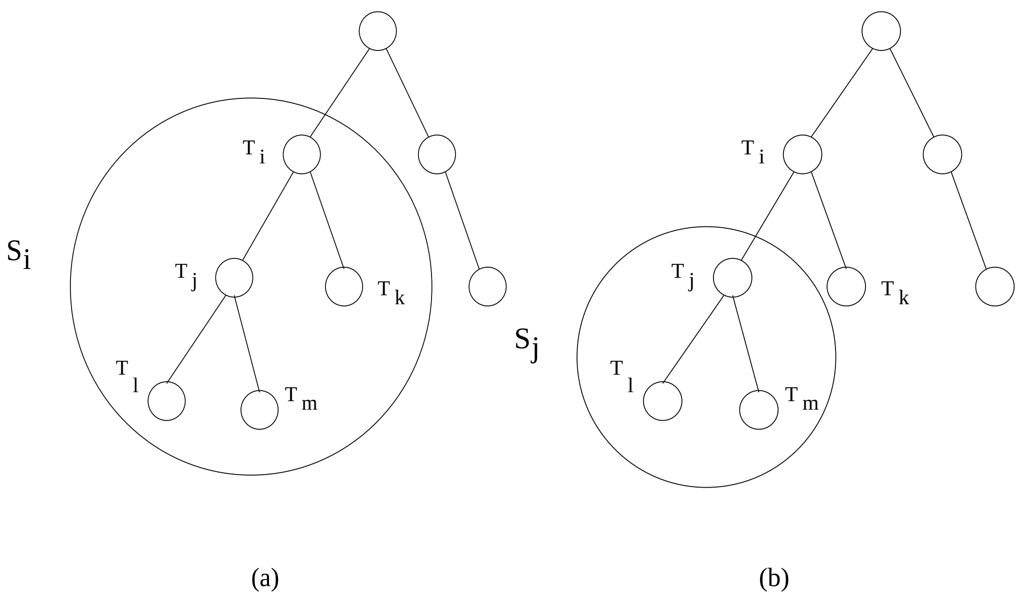

Fig. 13: (a) Control Sphere of T_i, (b) Control Sphere of T_j.

In handling an object access request, the *fetch_object* method first checks whether the requested object is in the *Lock_Table*. If not, it just requests the object from OpenOODB and returns a pointer to it. If the requested object is in the *Lock_Table* then the nested transaction concurrency control protocol is put into action. If the lock can be granted, then a pointer to the object is returned as in the previous case. If the lock cannot be granted with the requested *lock_type*, then for each transaction that has a lock on the object that conflicts with the requested *lock_type*, a node is inserted to the *wait_for_list* of the lock requesting transaction and a corresponding node is also appended to the *waited_by_list* of the conflicting transaction. Additionally, the *wait_for_count* (i.e., the number of transactions for which the transaction in concern is waiting for) is incremented for each node appended to the *wait_for_list*. Deadlock detection is performed each time a node is appended to *wait_for_list*. If no deadlock is detected after appending all the nodes, then the transaction that requested the lock is suspended using *thr_suspend()*, otherwise it is aborted. Transactions are unblocked using the *thr_continue()* function provided by the thread library. Unblocking of a transaction may occur due to the commit or abort of another transaction. When a transaction is aborted, all its locks are released, and all the nodes in the *waited_by_list* of that transaction are deleted; while doing the deletions, *wait_for_count*s of the corresponding (blocked) transactions are decremented by one. Blocked transactions whose *wait_for_count*s become zero, are unblocked, and their lock requests are reconsidered. When a subtransaction commits, all the locks held or retained by that transaction are inherited by the parent transaction which may cause some transaction(s) to be unblocked. Those transactions are identified by checking the *waited_by_list* of the committing transaction, and decrementing the *wait_for_count*s of the transactions which are descendants of the transaction inheriting the locks. The transactions whose *wait_for_count*s become zero are unblocked and their lock requests are reconsidered as in the previous case.

OpenOODB views a transaction hierarchy as one flat transaction; i.e., it is not aware of subtransactions. Lock requests made by a subtransaction is treated by OpenOODB as if the top-level transaction made the request. If there exists a deadlock cycle involving subtransactions belonging to different transaction hierarchies, then, from the OpenOODB viewpoint, it means that there is a deadlock cycle involving the top-level transactions as well. Therefore, deadlocks that can occur among transactions belonging to different transaction hierarchies are resolved by OpenOODB via EXODUS storage manager. Deadlocks among the transactions belonging to the same transaction hierarchy are detected and resolved by the new component we added to OpenOODB for managing nested transactions. To detect deadlocks, we use the *wait-for-graph* data structure. Whenever a sub-transaction is spawned, a node is appended to the *child-list* of the parent transaction. The union of those linked lists is also used as the *wait-for-commit* graph. When a transaction is blocked on a lock request, all the transactions that cause this transaction to be blocked are kept as a linked list (*wait-for* list) to be used for unblocking the transaction later on. Those lists also represent the *wait-for-retained-lock* or *wait-for-held-lock* relationship as explained in Section 3.1. During the commit of a transaction, *wait-for* list of the committing transaction is inherited by its parent.

Deadlock detection is performed before each insertion to a *wait-for-graph* and just before lock inheritance using the graph coloring technique. If a *wait-for* arc (T_i,T_j) is going to be inserted to the *wait-for-graph* or for any of the locks being inherited, if the inheriting transaction is not an ancestor of a transaction waiting for that lock, deadlock detection algorithm is started assuming that the new link or the inherited link is added. Initially all the nodes of the graph are colored $BLACK$. Deadlock detection algorithm colors T_i as $WHITE$, and marks all the nodes on its way recursively till there are no remaining nodes or a $WHITE$ node is reached. Deadlock detection algorithm uses the child and *wait-for* lists of the transactions during its traversal. Since all the nodes are initially $BLACK$, a $WHITE$ node reached means that there is a cycle in the graph, implying a deadlock situation. Deadlock algorithm recursively restores the colors of all the nodes it traversed back to $BLACK$.

If the insertion of the arc (T_i, T_j) to the *wait-for-graph* leads to a deadlock, we can abort either of T_i and T_j, or any other transaction in the deadlock cycle. Deciding which transaction to abort is not a trivial task. The simplest solution is to abort T_i (i.e., the transaction that requests the lock). Another possibility would be to abort the transaction that caused the lock-requesting transaction to block. Other possibilities for deciding which transaction to abort require the involvement of some information such as the arrival time or the number of subtransactions of the transactions in concern. We can choose the transaction that has the smallest transaction hierarchy, or we can choose the youngest transaction. As a future work, these alternative techniques could be implemented and their effects on the performance could be observed. We have chosen the simplest solution in our implementation, i.e., aborting the lock requesting transaction.

4.3. Integration of Our Implementation to Sentinel

Our implementation of concurrent rule execution is currently being integrated into Sentinel [9] which is an ADBMS built on OpenOODB. OpenOODB uses the Exodus storage manager. Concurrency control and recovery is provided by Exodus as well. Following capabilities were added to OpenOODB to transform it into an ADBMS [29]:

- specification of ECA rules either as a part of the class definition or as part of an application which is further pre-processed into a code for event detection and rule execution,
- detection of primitive events by using the sentry mechanism of OpenOODB. Sentry mechanism provides a wrapper method by which an event is notified to the composite event detector,
- composite event detection for each OpenOODB application.

Each application has a local event detector which receives the primitive event signals. When a primitive event occurs, it is sent to the local event detector and the application waits until all immediate rules are signaled. The global event detector communicates with the local event detectors for locally detected events [29].

In Sentinel, rules are treated as objects which means that they can be created, modified, and deleted in the same way as other objects. Subscription mechanism is used to reduce the checking overhead of rules; i.e., when an object generates an event, during rule execution, the rules that are checked are the ones which have previously been subscribed to the object that has generated the event.

```
class Rule:Notifiable // Rule class made notifiable
{
    char *name; // Rule name
    Event *event-id; //
    PMF *condition, *action; // PMF is a pointer
                             // to a member function
    Coupling mode; // Coupling Mode
    int enabled; // Rule Enabled or not

    public:
        virtual int Enable();
        virtual int Disable();
        virtual Update(Event* eventid);
        virtual int Condition();
        virtual int Action();
        Rule(Event* eventid, PMF condition, PMF action, Coupling mode);
        ~Rule();
```

Fig. 14: Rule Class of Sentinel.

In Sentinel, there is a *rule class* and all the rules in the system are instances of that class. Rules are specified as condition and action functions whose pointers are stored in a rule object along with the event. Rule object is an instance of the rule class. Events, conditions, and actions can be reused in multiple rule definitions. The rule class, shown in Figure 14, is a subclass of the *notifiable class* which means that it can receive and record primitive events generated by reactive (i.e., event generating) objects. A notifiable rule object subscribes to a set of reactive objects and all the primitive events generated by those reactive objects are propagated to the rule object via the notification mechanism. The notifiable rule object records only the primitive events of interest and discards the rest. Each rule has a *name*, *event-id*, *condition*, *action*, *mode* and *enabled*. The *event-id* denotes the identity of the event object associated with the rule. *Condition* and *action* are pointers to the condition and action member functions, respectively. The attribute *mode* denotes the coupling mode, and *enabled* indicates whether the rule is enabled or disabled.

Rule specification is done by the user at the beginning of the application. User specifies the event, condition, and action, priority and the coupling mode. The event detection is done after the application is written and rule definition is completed as part of the rule processing. The code written by the user is pre-processed so that the events are detected based on the method name given in the rule specification. After that, event specification is wrapped and the rule processor code is called upon the detection of the event. If the application reaches an event, the corresponding rule(s) is executed. Each rule is executed as a subtransaction in a separate thread.

When an event is detected, all the rules with the same priority for that event are wrapped into subtransactions and executed inside threads or top-level OpenOODB transactions and executed inside another process depending on the coupling mode.

In integrating our implementation to Sentinel, declaration of the condition and action functions are modified so that they can be executed in subtransactions. The modified function declarations look like the sample subtransactions provided in Figure 12. Furthermore, the rule execution component of Sentinel is modified so that conditions and actions of rules can be executed in parallel inside the subtransactions.

5. EXECUTION OVERHEAD OF PARALLEL NESTED TRANSACTIONS

To have an idea about how the performance of the system can be affected by the overhead of the implementation of parallel nested transactions, we simulated the fetch operation of OpenOODB using a dummy method in case the object is not in the *Lock_Table*. Once an object is inserted to the *Lock_Table*, fetch requests for that object is considered by the nested transaction component. Fanout (i.e., the maximum number of subtransactions that can be spawned by a transaction) and the total number of transactions in a transaction hierarchy are two important parameters that indicate the complexity of the transaction hierarchy.

We conducted two experiments to evaluate the impact of the parameters fanout and the total number of transactions in a transactions hierarchy on the performance in terms of the total CPU time used by all the transactions in a transaction hierarchy. In both experiments, total number of objects accessed by all the transactions in the hierarchy was kept constant. Approximately 20% of the fetch requests are done with a WRITE lock, and the rest are done with a READ lock. There are no hot spots in the object space. The number of objects that a transaction requests is calculated by the total number of objects divided by the total number of transactions in the hierarchy. Therefore as the total number of transactions increases, the number of objects accessed by a transaction becomes less but the total number of objects accessed by the hierarchy remains constant. The OpenOODB operations are not involved in the CPU time measurements as they are just simulated rather then being actually implemented. Therefore, the measured CPU times correspond to the execution overhead introduced by the implementation of parallel nested transactions. Transactions used in the experiments are actual transactions generated automatically by a transaction generator program. The total number of objects accessed by the whole transaction hierarchy is 100000 and the object size is 100 bytes.

In the first experiment, the number of transactions was kept constant and the performance impact of increasing the fanout value was evaluated. Total number of the transactions in the hierarchy is 70 and the fanout value ranges between 1 and 5. As illustrated in Figure 15, the CPU time used by the overall transaction hierarchy increases slightly with increasing fanout. In the second experiment, we kept the fanout constant at 3 and increased the number of transactions from 5 to 70. By keeping the fanout value constant, we eliminated the impact of fanout. The change in the CPU time as a result of increasing the number of transactions is negligible as in the previous case. The results of this experiment leading to that observation are illustrated in Figure 16. Both experiments that we have briefly discussed show that the overhead induced by the implementation of our parallel nested transaction model is not considerable.

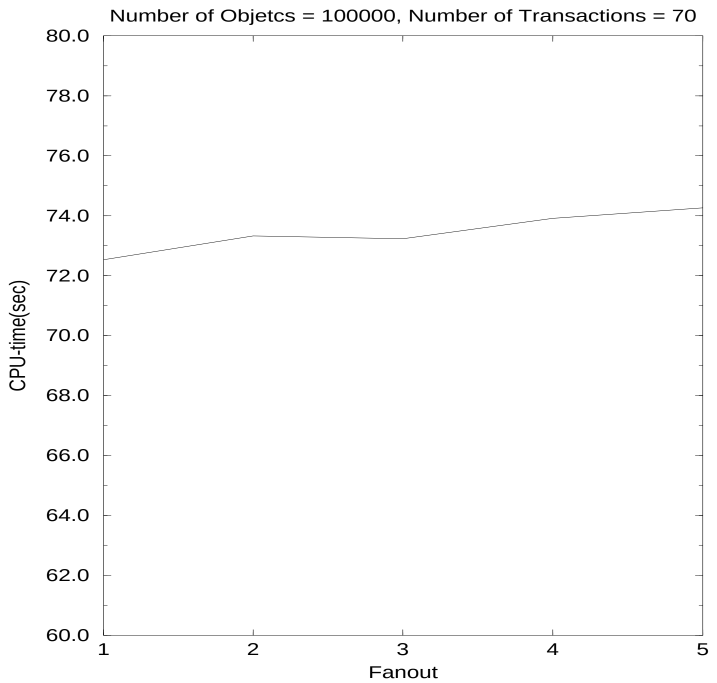

Fig. 15: Impact of Fanout on CPU-Time Used by a Transaction Hierarchy.

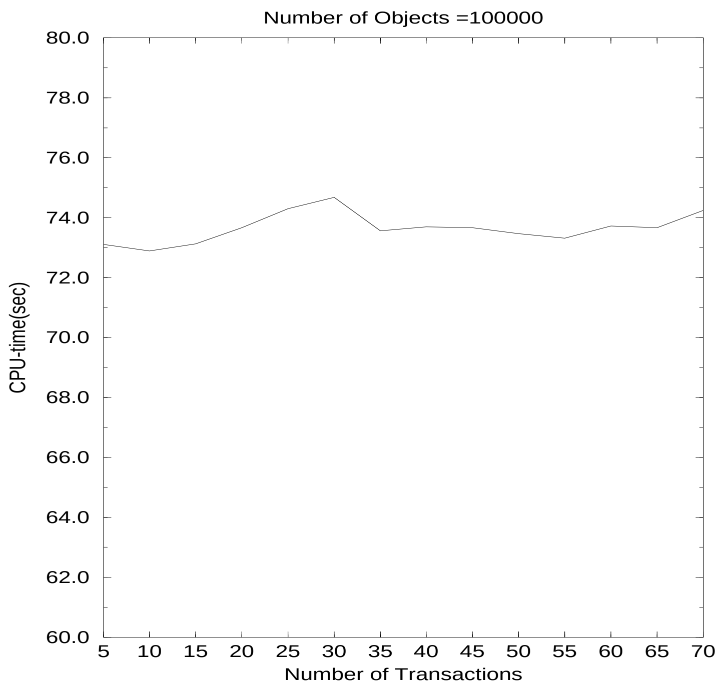

Fig. 16: Impact of the Number of Transactions on CPU-Time Used by a Transaction Hierarchy.

6. DISCUSSION AND FUTURE WORK

In this paper, we described a rule execution model for active database management systems (ADBMSs). We used nested transactions for concurrent rule execution and covered some aspects of nested transactions like concurrency control and deadlock detection. Although we implemented nested transactions on OpenOODB, our rule execution module was designed in such a way that it can easily be ported to other systems as well.

Some of the highlights of our implementation of rule execution can be listed as follows:

- Previous implementations for rule execution assume only sibling parallelism. We implement, on the other hand, both sibling and parent-child parallelism which is the most flexible kind of parallelism.
- We support deferred and detached modes of rule execution as well as immediate mode.
- In previous implementations, triggered rules are executed in distinct processes. We execute triggered rules in threads which is more efficient than forking processes.

It was shown through experiments that the execution overhead of nested transactions that were used in the implementation of concurrent rule execution is negligible and is not significantly affected by the number of transactions in the transaction hierarchy.

Implementation of the recovery of nested transactions is left as a future work. A log based recovery method, ARIES/NT [32], was proposed for nested transaction recovery which is an extension of the ARIES recovery method. Version based recovery techniques can also be applied which are easier to implement but have some major drawbacks against log-based recovery techniques [26]. Since our implementation was built on top of Exodus, implementation of recovery for nested transactions should involve some modifications to Exodus which is a difficult task. There is also no consensus on the abort semantics of rules. One way to think is that only the rule is aborted but not the triggering rule which is compatible with the nested transaction abort semantics. Another way is to abort the triggering rule as well in which case the nested transaction model should further be modified to support the abortion of the parent in case the child aborts. Same problems arise when a rule should be aborted as a result of a deadlock.

Our implementation of concurrent rule execution through parallel nested transactions is currently being integrated into Sentinel which is an ADBMS developed at the University of Florida. Following the completion of the integration, we are planning to investigate the impact of concurrent rule execution on the performance of Sentinel.

Acknowledgements — The authors wish to thank Prof. Alejandro Buchmann and the database research group of the Technical University of Darmstadt for their help during the initial phases of our work.

REFERENCES

[1] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 59–68, San Diego, California (1992).

[2] C. Beeri and T. Milo. A model for active object oriented database. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 337–349, Barcelona, Spain (1991).

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley (1987).

[4] A. Buchmann. Active object systems. In Asuman Dogac, M. Tamer Ozsu, Alex Biliris, and Timos Sellis, editors, *Advances in Object-Oriented Database Systems*, pp. 201–224, Springer-Verlag (1994).

[5] A.P. Buchmann, A. Deutsch, and J. Zimmermann. The REACH Active OODBMS. Technical report, Technical University Darmstadt (1995).

[6] A.P. Buchmann, J. Zimmermann, J.A. Blakeley, and D.L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pp. 117–128, Taipei, Taiwan (1995).

[7] A Joint Report by the ACT-NET Consortium. The active database management system manifesto: A rulebase of adbms features. *ACM SIGMOD Record*, **25**(3):40–49 (1996).

[8] A. Chakravarthy and E. Anwar. Exploiting Active Database Paradigm For Supporting Flexible Transaction Models. Technical Report UF-CIS-TR-95-026, University of Florida, Computer and Information Science and Engineering Department (1995).

[9] S. Chakravarthy, E. Anwar, and L. Maugis. Design and Implementation of Active Capability for an Object-Oriented Database. Technical Report UF-CIS-TR-93-001, University of Florida, Department of Computer and Information Sciences (1993).

[10] W.-S. Choe. Concurrency control for nested transactions in multidatabase systems. In *Proceedings of Seventh International Conference of Computing and Information*, pp. 649–668, Trent University, Peterborough, Ontario, Canda (1995).

[11] P.K. Chrysanthis and K. Ramamritham. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Databases*, pp. 103–112, Barcelona, Spain (1991).

[12] P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, **19**(3):450–491 (1994).

[13] U. Dayal. Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pp. 150–169, Jerusalem (1988).

[14] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 204–214, Atlantic City (1990).

[15] U. Dayal, M. Hsu, and R. Ladin. A generalized transaction model for long-running activities and active databases. *Data Engineering Bulletin*, **14**(1):4–8 (1991).

[16] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 113–122, Barcelona (1991).

[17] L. Daynes, O. Gruber, and P. Valduriez. Locking in OODBMS client supporting nested transactions. In *Proceedings of the 11th International Conference on Data Engineering*, pp. 316–322, Tai-pei, Taiwan (1995).

[18] S. Gatziu, A. Geppert, and K.R. Dittrich. The samos active dbms prototype. Technical report, Institut fur Informatik, Universität Zürich (1994).

[19] P. Graham, K. Barker, and A. Reza-Hadaegh. Disconnected objects: Reconciliation in a nested object transaction environment. In *Proceedings of ECOOP'96, Workshop on Mobility and Replication* (1996).

[20] E.N. Hanson. The design and implementation of the ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, **8**(1):157–173 (1996).

[21] E.N. Hanson and J. Widom. An Overview of Production Rules in Database Systems. Technical report, University of Florida, Department of Computer and Information Sciences (1992).

[22] T. Harder, M. Profit, and H. Schonning. Supporting Parallelism in Engineering Databases by Nested Transactions. Technical Report 34/92, University of Kaiserslautern, Computer Science Department (1992).

[23] T. Harder and K. Rothermel. Concurrency control issues in nested transactions. *VLDB Journal*, **2**(1):39–74 (1993).

[24] M. Hsu, R. Ladin, and D.R. McCarthy. An execution model for active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pp. 171–179, Jerusalem (1988).

[25] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, **19**(4):303–328 (1987).

[26] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwartz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, **17**(1):94–162 (1992).

[27] E. Moss. *Nested Transactions.* M.I.T. Press, Cambridge, Mass. (1985).

[28] J. Eliot and B. Moss. Log-based recovery for nested transactions. In *Proceedings of the 13th VLDB Conference*, pp. 427–432, Brighton (1987).

[29] S. Neelakantan. Scheduling rules in an active DBMS using nested transactions. Master's Thesis in progress, Department of Computer and Information Science and Engineering, University of Florida, 1997.

[30] N.W. Paton and O. Diaz. Active Database Systems. Technical report, University of Manchester, Department of Computer Science (1995).

[31] C. Pu and J.D. Noe. Design and implementation of nested transactions in eden. In *Proceedings of Sixth International Symposium on Reliability in Distributed Software and Database Systems*, pp. 126–136, Kingsmill-Williamsburg, VA (1987).

[32] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pp. 337–346, Amsterdam (1989).

[33] M. Rukoz. Hierarchical deadlock detection for nested transactions. *Distributed Computing*, **4**:123–129 (1991).

[34] Y. Saygin, Ö. Ulusoy, and S. Chakravarthy. Concurrent rule execution in active databases. Technical Report BU-CEIS-9707, Department of Computer Engineering and Information Science, Bilkent University (1997).

[35] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. *Getting Started with Solaris* (1992).

[36] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. *Multithreaded Programming Guide* (1994).

[37] D.L. Wells, J.A. Blakeley, and C.W. Thompson. Architecture of and open object-oriented database management system. *IEEE COMPUTER*, pp. 74–81 (1992).

[38] J. Widom. The starburst active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, **8**(4):583–595 (1996).