A Unifying Framework for Transactions in Competitive and Cooperative Environments

Panayiotis K. Chrysanthis *Krithi Ramamritham*

Department of Computer and Information Science
University of Massachusetts
Amherst MA. 01003

Abstract

Recently, a number of extensions to the traditional transaction model have been proposed to support new information-intensive applications such as CAD/CAM and software development. However, these extended transaction models capture only a subset of interactions that can be found in such applications, and represent only some of the points within the spectrum of interactions possible in competitive and cooperative environments.

ACTA is a formalizable framework intended for characterizing the whole spectrum of interactions. The ACTA framework is not *yet* another transaction model, but it consolidates the different transaction models into a unified and versatile framework. ACTA allows for specifying the *structure* and the *behavior* of transactions as well as for reasoning about the concurrency and recovery properties of the transactions. In ACTA, the semantics of interactions are expressed in terms of transactions' effects on each other and on the objects that they access. Its ability to capture the semantics of previously proposed transaction models is indicative of its generality. The reasoning capabilities of this framework have also been tested by using the framework to compare the properties of existing transaction models and to derive new transaction models by manipulating the characterization of existing models.

Published in *IEEE Office Knowledge Engineering*,
4(1):2–21, February 1991.

This material is based upon work supported by the N.S.F. under grant DCR-8500332.
E-mail: panos@ccs3.cs.umass.edu, krithi@nirvan.cs.umass.edu

1 Introduction

Broadly speaking, whether a system is characterized as competitive or cooperative depends on how *interactions* among activities in the system are viewed: In competitive environments, interactions are curtailed whereas they are promoted in cooperative environments. Traditional transaction mechanisms [4, 11] were designed for competitive environments where transactions were assumed to be short-lived and to execute without interference. Interactions among transactions due to the interleaving of their execution are constrainted to occur according to the serializability requirement [7, 17].

Transaction models such as Nested Transactions [16] and Recoverable Communicating Actions [24], were initially proposed as alternative transaction models for dealing with reliability in cooperative distributed systems. These models are able to handle partial failures while exploiting the spatial and functional distribution of the activities in the system.

The need for cooperative transactions models emerges from the demands of new information-intensive applications such as office information systems, software development, stock trading databases and CAD/CAM. Although powerful, the traditional transaction model is found lacking in both *efficiency* and *functionality* when used in complex information systems which support these new applications. Efficiency is of particular importance considering the throughput demands placed on these complex information systems. Beyond being distributed, these systems are typically object based [26]. Activities in complex information systems tend to access many objects, involve lengthy computations, and are interactive, i.e., pause for input from the user. Even in those cases where activities with such characteristics can be modeled as traditional transactions, they degrade the system performance due to increased data contention, thus failing to meet the high throughput demands. In terms of functionality, reactive (endless), open-ended (long-lived) and collaborative (interactive) activities which are often found in these systems, cannot be captured by traditional transactions due to serializability as the correctness requirement.

The characteristics of different applications call for different styles of cooperation. An application may call for several styles of cooperation among its activities simultaneously. Various extensions to the traditional transaction model have been proposed [16, 3, 18, 24, 10, 20, 8, 23, 22]. These, referred to herein as *complex transactions*, can provide the basis for realizing different styles of cooperation. Irrespective of how successful these extended transactions models are in supporting the systems that they were intended for, they were designed with particular situations in mind and thus, they can capture only a subset of the interactions to be found in any complex information system.

We have developed a comprehensive transaction framework, called *ACTA*, that consolidates these different transaction models into a unified and versatile framework for characterizing the whole spectrum of interactions. In ACTA, the semantics of interactions between transactions are expressed in terms of transactions' effects on each other and on the objects that they access. For example, a transaction has two possible outcomes, namely, commit or abort. Consequently, a transaction can

affect *the abortion* or *the commitment* of other transactions. Also, a transaction can affect the *state* of objects as well as the *concurrency status*, i.e., synchronization state, of objects. (Henceforth, we refer to concurrency status as just *status*).

ACTA provides (1) for specifying the effects due to the *structure* and the *behavior* of transactions and (2) for reasoning about the concurrency and recovery properties of the transactions. *Structure* refers, for example, to the nesting structure of a transaction, and *behavior* refers to the operations invoked by a transaction. In this paper we focus on the specification part of our framework.

In Section 2, we examine the characteristics of complex transactions. In Section 3, we present ACTA, our comprehensive transaction framework and discuss the intuition underlying the framework. Section 4 demonstrates the expressiveness of the framework through the study of the properties of the Joint Transaction model [20] and its variations. Section 5 concludes with a summary and discusses future steps.

2 Characteristics of Complex Transactions

Traditional transactions [7, 11] are based on the notion of *atomicity* and thus are often referred to as *atomic transactions*. Atomicity is characterized by two properties: failure atomicity and serializability. *Failure atomicity* means that either all or none of the transaction's operations are performed. *Serializability* means that concurrent transactions execute without any interference as though they were executed in some serial order. However, these properties combine several important notions such as:

1. *Visibility*, referring to the ability of one transaction to see the results of another transaction *while* it is executing.
2. *Permanence*, referring to the ability of a transaction to record its results in the database.
3. *Recovery*, referring to the ability, in the event of failure, to take the database to some state that is considered correct.
4. *Consistency*, referring to the correctness of the state of the database that a committed transaction produces.

The flexibility of a given transaction model depends on the way these four notions are combined. Thus, these notions have to be revisited in order to understand the properties of complex transactions and to decide on the mechanisms for supporting them. For example, visibility does not always have to be curtailed, permanence need not require all the results to be recorded in the database, recovery does not imply the complete restoration of the state, and consistency does not necessarily require serializability. The ACTA framework described in the next section allows us to capture the properties of transactions as related to these four notions.

Generally, *complex transactions* can be said to consist of either a set of operations on objects or a set of components, each of which is a complex transaction. Thus, the component transactions are not necessarily *atomic*. This recursive formulation implies that a complex transaction may exhibit a rich and complex internal structure. In contrast, traditional transactions have a flat single level structure. In this sense, the base case in this recursive definition of complex transactions is similar to a traditional transaction. A simple example of complex transactions is nested transactions. We will be using nested transactions to clarify and illustrate various concepts in the rest of the paper, since this is a well known transaction model and does not need any further introduction.

The internal structure of complex transactions is *explicit* and is provided as a user facility. The way that component transactions are combined to form complex transactions reflects the semantics of an application. Such semantics can be exploited in designing *transaction specific concurrency control* and *transaction specific recovery*. The idea is similar to the use of semantic information about the objects and their operations in designing *type specific concurrency control* to enhance concurrency within objects [1, 21, 13, 25].

Transaction specific concurrency control allows the definition of new weaker notions of conflicts among operations not possible with the information available only about objects and their types. For instance, operations invoked by two transactions can be interleaved as if they commuted, if the semantics of the application allow the dependencies between the transactions to be ignored. Clearly, transaction specific concurrency control might *not* achieve serializability but still preserve consistency [9, 15]. This seems to be an attractive means for increasing the performance in a complex information system.

Transaction specific recovery can be designed along the same lines to exploit the semantics of the application in order to minimize the effects of transaction failures. Transaction specific recovery reduces the cost of recovery by tolerating partial failures and by supporting both forward and backward recovery. In the event of failure of transaction components, the failed portions can be isolated, allowing the rest of the transaction to proceed. Failed portions of a transaction can be retried, compensated (annulling their effects), replaced by another contingent alternative, or even be ignored [16, 10, 14]. Furthermore, complex transactions naturally support user-controlled checkpointing since the boundaries of component transactions act as checkpoints.

In order to cooperate, two transactions must be aware of each other and must be able to share partial results and coordination information. *Visibility* represents this ability of one transaction to see the effects of another transaction while the latter is executing. It can be argued that this is true even in competitive environments. For example, two-phase locking is based on the visibility of locks. If locks were not visible, transactions would not be able to coordinate by requesting locks, acquiring locks and releasing locks. However, in a competitive environment, in general, a user is not allowed to utilize information about the state of the lock or any other information that might be visible due to the implementation. In contrast, in a cooperative environment, visibility is an issue that the users must be

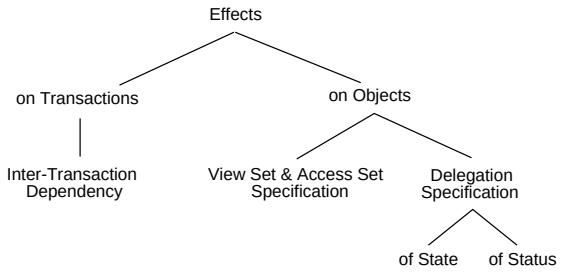

Figure 1: Dimensions of the ACTA framework

aware of and they should be able to make use of it. In fact, we believe the degree of visibility is what controls cooperation.

3 The ACTA Framework

The behavior of a transaction system is determined by the behavior of its active components and the interactions among these components. The active components in our framework are *transactions,* inherently parallel activities, and the passive components are *objects*, abstract entities manipulated by transactions.

Transactions may produce unexpected results if they interact indiscriminately. A correctness criterion for transactions constrains these interactions to those that produce a result contained in a set of acceptable results. In order to specify a correctness criterion that prevents some interactions from occurring while allowing others, we must be able to express these interactions. Interactions among transactions are reflected in the effects they cause and thus, we can express them in terms of these effects: We distinguish between transactions' effects on each other and transaction effects on the objects that they access. These two types of effects as well as a precise way to specify them are described in Section 3.1 and 3.2 respectively.

In general, transactions affect each other and their changes to objects are made effective or visible upon the occurrence of certain *significant* events. For instance, in the traditional transaction model, significant events are *begin-transaction, commit* and *abort*. In addition to the above, the Nested Transaction model has the *spawn* event which spawns a child transaction. In the Split Transaction model, *split* and *join* are significant events. In ACTA, specification of transaction semantics with respect to the events of significance to a model can be specified using the taxonomy of effects captured in figure 1.

3.1 Effects of Transactions on other Transactions

Dependencies provide a convenient way for specifying and reasoning about the behavior of concurrent transactions [12, 21]. By examining the possible effects of interacting transactions on each other, it is possible to determine the dependencies that may develop between the transactions. Two important

dependencies that a transaction may develop on any other transaction are *commit-dependency* and *abort-dependency* which are related to the significant events of commit and abort. These dependencies, collectively known as *completion dependencies*, are defined as follows:

Commit-Dependency: If a transaction A develops a *commit-dependency* on another transaction B (denoted by $A \rightsquigarrow B$), then transaction A cannot commit until transaction B either commits or aborts. This does not imply that if transaction B aborts, then transaction A should abort. *Transitive-commit-dependency* (denoted by $\overset{*}{\rightsquigarrow}$) is defined by the transitive closure of commit-dependencies. A transaction A has a transitive-commit-dependency on every member of the set of transactions formed by the transitive closure of commit-dependencies starting from A.

Considering a very simple example, if T_1 reads an object x and then T_2 writes x, serializability can be preserved by T_2 forming a commit-dependency on T_1.

Abort-Dependency: If a transaction A develops an *abort-dependency* on another transaction B (denoted[1] by $A \rightarrow B$), and if transaction B aborts, then transaction A should also abort. This neither implies that if transaction B commits, then transaction A should commit, nor that if transaction A aborts, then transaction B should abort. *Transitive-abort-dependency* (denoted by $\overset{*}{\rightarrow}$) is defined by the transitive closure of abort-dependencies. A transaction A has a transitive-abort-dependency on every member of the set of transactions formed by the transitive closure of abort-dependencies starting from A.

Considering once again a simple example, if T_3 writes x and then T_4 reads x, serializability can be preserved by T_4 forming an abort-dependency on T_3.

In general, in a complex transaction system, the completion of a transaction may not depend on a simple condition, such as the completion of another transaction, but may depend on a complex condition required to capture the interactions among the transactions in the system. Thus, in general, the commit-dependency of a transaction A can be expressed as: $A\ commits \Rightarrow Condition$, which states that if A commits, then $Condition$ is satisfied. Similarly, the abort-dependency of A can be expressed as: $Condition \Rightarrow A\ aborts$, which states that if $Condition$ is satisfied, then A is aborted. In general, the dependencies of any significant event that relates to transaction execution can be expressed in a manner similar to that of commit and abort dependencies.

Commit-dependencies and abort-dependencies impose a commit order which prevents transactions from prematurely committing, thereby preventing object inconsistencies, given that transactions preserve the consistency of the database when run in isolation. Depending on a transaction model and its correctness notion, some dependency cycles may lead to inconsistencies and hence, they are prohibited, whereas other dependency cycles are accommodated. In the latter case, if two transactions

[1] The specific direction of the arrows for commit and abort dependencies is chosen for readability reasons. To reflect the required order of transactions' commitment, the arrows should be drawn in the opposite direction.

form a circular dependency involving abort-dependencies, then both have to commit or neither. In the case that two transactions develop a circular dependency involving only commit-dependencies, then if both transactions commit, their commitment must be synchronized. That is, if one of the transaction aborts the other transaction can still commit. Similarly, in the case that two transactions develop a circular dependency involving dependencies of different types, i.e., one transaction has a commit-dependency on another transaction which has an abort-dependency on the first transaction, then the commitment of both transactions must be synchronized.

3.1.1 Source of dependencies

Dependencies between transactions may be a direct result of the structural properties of the complex transaction formed by the interacting transactions, or may indirectly develop as a result of interactions of transactions over shared objects.

Dependencies due to Structrure

The structure of a complex transaction defines its component transactions and the dependencies between them. The dependencies are the links in the structure. For example, in hierarchically-structured nested transactions, a child transaction C has an abort-dependency on its parent P that guarantees the abortion of the child in case its parent aborts ($C \rightarrow P$), while a commit-dependency of the parent on its children ensures that the parent does not commit before all its children have terminated ($P \rightsquigarrow C$).

In order to completely specify the structure of a complex transaction, it is often necessary to specify *prohibited* dependencies, i.e., dependencies that should be prevented from developing between two transactions. For example, the hierarchical structure of nested transactions prohibits a child from developing an abort-dependency on any transaction other than its parent. ACTA uses *dependency production rules* for specifying both completion dependencies and prohibited dependencies. A dependency production rule (production rule or production for short) consists of a *labeling specification* and a *dependency specification*:

$$\underbrace{\begin{array}{c} A \vdash B \\ B \vdash C \end{array}}_{labeling\ specification} \qquad \underbrace{\left\{ \begin{array}{c} C \rightsquigarrow B \\ B \rightarrow C \end{array} \right\}}_{dependency\ specification}$$

The labeling specification of this production denotes that a transaction currently labeled A can be relabeled to B and another transaction currently labeled B can be relabeled to C ($\vdash$ is the relabel symbol). The dependency specification denotes that relabeled transaction C should develop a commit-dependency on relabeled transaction B and relabeled transaction B should develop an abort dependency on relabeled transaction C. A production rule is applicable in the context where transactions exist that satisfy the left hand side labels for all the labeling specifications. Once

a production rule is applied, all the transactions involved are relabeled according to the labeling specification and dependencies between the transactions are established according to the dependency specification. It is possible for the dependency specification of a production to be empty. In such a case, the labeling specification is used to specify the context in which the interaction between the transactions can take place.

The dependencies that are allowed by the structure of a complex transaction are those specified in the production rules that characterize the structure of the complex transaction. All other dependencies not specified in the production rules are prohibited dependencies. For example, the complete specification of the structure of nested transactions involves three production rules that constitute the specification of the dependencies formed due to the *spawn* event used to generate a child transaction. As already mentioned, spawn is a significant event of the Nested Transaction model. Upon an invocation of the spawn event one of these production rules must be applied.

$$\begin{array}{l} T \vdash P \\ T \vdash C \end{array} \left\{ \begin{array}{l} P \rightsquigarrow C \\ C \rightarrow P \end{array} \right\}, \quad \begin{array}{l} C \vdash P \\ T \vdash C \end{array} \left\{ \begin{array}{l} P \rightsquigarrow C \\ C \rightarrow P \end{array} \right\}, \quad \begin{array}{l} P \vdash P \\ T \vdash C \end{array} \left\{ \begin{array}{l} P \rightsquigarrow C \\ C \rightarrow P \end{array} \right\}$$

T is a generic (traditional) transaction. Initially, all transactions are labeled T but once they are relabeled, they maintain that label until another relabeling, if any, occurs. For example, after the first production is applied to two transactions, one of the transactions that was initially labeled T will have the label P whereas the other one will have the label C.

The first production generates the root of the nested transaction and the first child of the root. The second production is for expanding the nested transaction in depth by turning a child to a parent and adding its child, whereas the third production is for expanding the nested transaction in breadth by adding a new child to an existing parent. Note that none of the productions allow the simultaneous relabeling of a parent as a child transaction ($P \vdash C$) and a child as a parent transaction ($C \vdash P$). Thus, the labeling rules prohibit a parent, including the root, from developing an abort-dependency on any of its descendants. Similarly, a child is prevented from developing an abort-dependency on more than one transaction since none of the specifications allows an abort-dependency between an existing child and an existing parent in a production.

Beyond capturing the effects due to the *static* structure of complex transactions, the dependency production rules can also specify the *dynamics* of the evolution of the structure of complex transactions. The above specification of nested transactions defines that a nested transaction expands in a top-down manner from the root to the leaves and hence conforms to the original Nested Transaction model. Suppose a nested transaction can be allowed to expand both top-down and bottom-up. This can be specified by adding new production rules that expand the nested transactions by turning the current root to a child transaction whose parent becomes the new root. The production rules in the above specification of nested transactions are modified to explicitly specify the root transaction by labeling it as R. Here is the complete specification of this variation of nested transactions which is similar to the notion of *supertransactions* [19].

$$\begin{matrix} T \vdash R \\ T \vdash C \end{matrix} \left\{ \begin{matrix} R \rightsquigarrow C \\ C \rightarrow R \end{matrix} \right\}$$

This first production creates a nested transaction by generating the root and its first child.

$$\begin{matrix} R \vdash P \\ T \vdash R \end{matrix} \left\{ \begin{matrix} R \rightsquigarrow P \\ P \rightarrow R \end{matrix} \right\}$$

This production expands a nested transaction upwards by making the current root the child of the new root.

$$\begin{matrix} R \vdash R \\ T \vdash C \end{matrix} \left\{ \begin{matrix} R \rightsquigarrow C \\ C \rightarrow R \end{matrix} \right\}$$

This production expands a nested transaction in breadth by adding a new child to the root.

$$\begin{matrix} R \vdash R \\ R \vdash P \end{matrix} \left\{ \begin{matrix} R \rightsquigarrow P \\ P \rightarrow R \end{matrix} \right\}$$

This production joins two nested transaction into one by making the root of one nested transaction a child of the root of the other.

$$\begin{matrix} C \vdash P \\ T \vdash C \end{matrix} \left\{ \begin{matrix} P \rightsquigarrow C \\ C \rightarrow P \end{matrix} \right\}$$

This production expands a nested transaction in depth by turning a child transaction to a parent and adding its child.

$$\begin{matrix} P \vdash P \\ T \vdash C \end{matrix} \left\{ \begin{matrix} P \rightsquigarrow C \\ C \rightarrow P \end{matrix} \right\}$$

This production expands a nested transaction in breadth by adding a new child to en existing parent.

Dependencies due to Behavior

Dependencies formed by the interactions over a shared object are specified by the *compatibility table* associated with the object which encodes the object's synchronization properties. In the traditional framework, a compatibility table is a simple binary relation with a *yes* entry for (O_i, O_j) indicating that the operations O_i and O_j are compatible, or a *no* entry indicating that the two operations are incompatible. Compatible operations do *not conflict* and can execute concurrently. In our case, an entry (O_i, O_j) could be a *condition* involving completion dependencies, operation arguments and results. This means that if transaction T_j invokes an operation O_j and later a transaction T_i invokes the operation O_i on the same object, then the *condition* must be satisfied in order for O_i to execute. For example, an entry could contain *No-Dependency* which corresponds to the standard *yes* entry in the commutativity-based tables, or it could be *Form-Commit-Dependency* which can be found in recoverability-based tables [2]. Other entries could be *Wait*, corresponding to the standard *no* entry in commutativity-based tables of locking-based schemes, *Abort, Form-Abort-Dependency, Allow-if-Abort-Dependency-already-exists, Allow-if-Commit-Dependency-already-exists*, etc.

Two other entries are *Notify* and *Confirm*. These entries are of particular importance because they are related to the notion of *notification* useful in a cooperative environment [8]. A *Notify* entry corresponding to (O_i, O_j) implies that transaction T_j invoking O_j should be notified of O_i's presence. *Notify* can be one of many mutually-consistent conditions specified as a compatibility-table entry. This allows a full set of notification facilities such as those in [8]. For instance, the two conditions *Notify*

and *Form-Commit-Dependency* in (O_i, O_j) will cause a commit-dependency to be established from transaction T_i to T_j as well as $notify\ T_j$ about the development of the commit-dependency. Such a pair of conditions can be used to define a recoverability-based table in a cooperative environment. Transaction T_j can use the information about the existence of the commit-dependency to postpone the invocation of another operation that causes a commit-dependency of T_j on T_i, and hence a circular commit dependency.

Confirm can be one of many conditions that can be specified as a compatibility-table entry. In the presence of *Confirm*, a condition cannot be satisfied unless it is accepted by the affected transactions. For example, if entry (O_i, O_j) has *Confirm* and *Form-Commit-Dependency* both T_i and T_j should confirm that they are agreeable to the formation of the dependency before a commit-dependency is established from T_i to T_j. Otherwise O_j is rejected. Note that confirmation is a stronger notion than notification.

The generality of the entries of the compatibility table allows ACTA to capture different types of type-specific concurrency control discussed in the literature [21, 13, 1], and even to tailor them for cooperative environments.

3.2 Effects of Transactions on Objects

Transactions' effects on objects are captured by the introduction of two sets, the *View Set* and the *Access Set*, and by the concept of *delegation*.

3.2.1 View Set and Access Set

Each object is characterized by its state and its status. The *state* of an object is represented by its contents. The state of an object changes when an operation invoked by a transaction modifies the contents of the object. The *status* of an object is represented by the synchronization information associated with the object. The status of an object changes when a transaction performs an operation on the object.

Transactions' effects on objects can be restricted by limiting the number of objects accessible to them. For this reason, every transaction is associated with a set of objects, called *View Set*, which contains all the objects potentially accessible to the transaction. Rules for composing the View Set are determined by the specific transaction model.

The effects of a transaction on objects are conditional upon the outcome of the transaction. Objects already accessed by the transaction are contained in another set, called *Access Set*. When an object in the View Set of a transaction is accessed by the transaction or a new object is created by the transaction, the object becomes a member of the transaction's Access Set. As long as an object remains in the Access Set of a transaction, it continues to be accessible to the transaction. An object ob in the View Set of a transaction T_1 can be accessed by T_1 only if the concurrency control

status of ob permits it. As mentioned earlier, part of the synchronization information is contained in the compatibility table. Let T_1 access ob using Op_1. Assume T_2 has already accessed ob using Op_2. If the (Op_1, Op_2) entry is *Wait*, T_1 will not be allowed to access ob and hence, ob cannot be added to the Access Set of T_1. In other words, status of an object with respect to a transaction depends on whether the object is in the View Set or Access Set of the transaction and on the state of the compatibility table of the object.

When a transaction *aborts*, the state and the status of all objects in the transaction's Access Set are restored in its View Set. When a transaction *commits*, the state of all objects in its Access Set is made persistent, i.e., the changes are effected, in the View Set, while the status is restored in the View Set. $AccessSet_T$ refers to the Access Set of a transaction T, and $ViewSet_T$ refers to the View Set of T.

For example, in nested transactions, the ability of a subtransaction to access any object accessed by one of its ancestor transactions is expressed by defining the View Set of the subtransaction in terms of the Access Sets of its ancestor transactions: $ViewSet_C = \{\cup AccessSet_A | C \xrightarrow{*} A\} \cup DB$, where DB stands for the database, the entity that has all the objects in the system. The state of the objects in DB reflects the most recently committed state of the objects. The transitive-abort-dependency uniquely specifies the ancestors of a subtransaction.

In our notation, $\cup$ is an *ordered union*. Informally, if $C = A \cup B$, then C contains all the elements of A and B as in a set union. However, if there is an element in A duplicated in B, C contains the element from A. We need this for the following reason. Suppose an object ob in DB is modified by P and is then accessed by Q. Then only the modified version of ob should be accessible to Q. Note that this notion of versions is different from object versions maintained explicitly for application-dependent reasons. We propose to capture the latter by viewing such versions as different objects. Versions in the current situation exist only until the root transaction terminates.

3.2.2 Delegation

A transaction may *delegate* the responsibility for finalizing its effects on some of the objects in its Access Set to another transaction. This is achieved by removing the delegated objects from the Access Set of the first transaction (*delegator*) and adding them to the Access Set of the second transaction (*delegatee*). That is, *delegation* represents the ability of a transaction to give up some of its objects which are then taken over by another transaction. Delegation effectively broadens the visibility of the delegatee and is useful in selectively making tentative or partial results as well as hints, such as, coordination information, accessible to other transactions.

The notion of delegation defined thus far is related to one of the two dimensions of objects, namely, the state, and thus is called *delegation of state*. There is another type of delegation related to the status of objects. *Delegation of status* as opposed to delegation of state, implies that the changes

done by the delegating transaction to the delegated objects are undone, before these objects are added to the Access Set of the delegatee. Effectively, the delegation of status represents the ability of one transaction to annul the changes and relinquish control of the visibility of some of its objects to another transaction. $DelegateSet_{state}(T_1, T_2)$ and $DelegateSet_{status}(T_1, T_2)$ refers to the set of objects delegated by T_1 to T_2. Since delegation of state is the common form when we drop the subscript, we are referring to delegation of state.

The notion of inheritance used in nested transactions is an instance of delegation. Specifically, inheritance as proposed in [16] corresponds to the delegation of state when the delegator commits ($DelegateSet_{state}(C, P) = AccessSet_C$, where $C \rightarrow P$), whereas inheritance in [18] corresponds to the delegation of status when the delegator aborts ($DelegateSet_{status}(C, P) = AccessSet_C$, where $C \rightarrow P$) and delegation of state when the delegator commits ($DelegateSet_{state}(C, P) = AccessSet_C$, where $C \rightarrow P$). Delegation does not occur only upon commit or abort but a transaction can delegate any of the objects in its Access Set to another transaction at any point during its execution (For examples see Co-Transactions and Reporting Transactions in Section 4.3 and 4.4 respectively).

Another form of delegation is *limited delegation* which does not remove the delegated objects from the Access Set of the delegator but makes them inaccessible to the delegator before adding them to the Access Set of the delegatee. That is, unlike the delegation of state, with limited delegation the effects of the delegator on the delegated objects are not discarded if the delegatee aborts.

Delegation is not only used in controlling the visibility of objects, but delegation in conjunction with commit and abort dependencies specifies the recovery properties of a transaction model.

In cooperative environments, transactions (components) cooperate by having intersecting Access Sets and View Sets, by delegating objects to each other, or by *notify*ing each other of their behavior. By being able to capture these aspects of transactions, the ACTA framework is designed to be applicable to cooperative environments.

4 Reasoning about Transactions in ACTA

In the previous section, the semantics of nested transactions were specified using the ACTA framework. In this section, in order to demonstrate the usefulness of our framework in reasoning about the properties of existing and future transaction models, a number of new models are generated by perturbing the characterization of an existing transaction model, namely, the Joint Transaction model. The characterization of other previously proposed transactions models such as Split Transactions, Recoverable Communicating Actions, Cooperative Transactions, Transaction Groups and Multi-Coloured Actions can be found in [6]. The ability of the ACTA framework to capture the semantics of these schemes is indicative of its generality. The properties of a new transaction model resulting from the combination of nested transactions and Split Transactions were studied in [6].

4.1 Joint Transactions

In the Split Transactions model [20], *join* is a significant completion event in addition to the standard *commit* and *abort* events. That is, it is possible for a transaction instead of committing or aborting, to join another transaction. The joining transaction releases its objects to the *joint* transaction.

When a transaction invokes a join, it completes. However, the effects of the joining transaction are made persistent in the database only when the joint transaction commits. Otherwise they are discarded. Thus, if the joint transaction aborts, the joining transaction is effectively aborted. Transactions behave like traditional transactions while they execute.

In the ACTA framework, the characterization of joint transactions is as follows. The *begin-transaction* event has the semantics of the begin-transaction event in the traditional transactions in which the View Set of the transaction is set to be the DB.

- Begin-Transaction Specification
  Sets the View Set to be the DB when the transaction begins:
  $ViewSet_T = DB$

Here is the complete specification of the *join* event. There are four types of join: (1) a traditional transaction joins another traditional transaction, (2) a traditional transaction joins another joint transaction, (3) a joint transaction joins another traditional transaction, and (4) a joint transaction joins another joint transaction.

- Join Specification
  - Dependency Specification

$T \vdash B$ $T \vdash A$	$\{\}$	This production joins two traditional transactions.
$B \vdash A$ $T \vdash B$	$\{\}$	This production joins a joint transaction B to a traditional transaction T.
$B \vdash B$ $T \vdash A$	$\{\}$	This production joins a traditional transaction T to a joint transaction B.
$B \vdash B$ $B \vdash A$	$\{\}$	This production joins two joint transactions into a single joint transaction.

  - Delegation Specification
    $DelegateSet_{state}(A, B) = AccessSet_A$

The delegation specification states that, when join occurs, the joining transaction's objects are delegated to the joint transaction. This means the joining transaction's effects are made permanent

to the DB only if the joint transaction commits. Join is effectively a conditional successful completion event. In this regard, a joining transaction behaves similar to child transaction of a nested transaction when child transaction commits. Note that the relabeling specification of joint transactions defines a structure similar to that of nested transactions (notice the similarity between the production rules for joint transactions and the first four productions of the variation of nested transactions in Section 3). Because of this, joint transactions can be said to be compatible with nested transactions in the sense that joint transactions can be used to join nested transactions into a single nested transaction, known as a supertransaction [19] by making their root transactions children of the supertransaction.

The *commit* event for joint transactions has the same semantics as the commit event in traditional transactions in which commit makes the transaction's effects on the objects in its Access Set permanent in its View Set, i.e., permanent in the DB, and visible to all other transactions in the system:

- Commit Specification
  Changes are committed to DB when the transaction commits:
  $ViewSet_T$ = $AccessSet_T \cup ViewSet_T$ (i.e., DB = $AccessSet_T \cup DB$)

Finally, the *abort* event also has the same semantics as the abort in traditional transactions.

- Abort Specification
  Changes are discarded when the transaction aborts:
  $AccessSet_T$ = ϕ
  $ViewSet_T$ = ϕ

  It can be shown that joint transactions are both serializable and failure atomic.

4.2 Chain Transactions

A special case of joint transactions is one that restricts the structure of joint transactions to a linear chain of transactions. We can call these transactions *Chain Transactions*. A chain transaction is formed initially by a traditional transaction joining another traditional transaction and subsequently by the joint transaction joining another traditional transaction. The dependency specification of the join event in chain transactions is given by the first two production rules of the join event in joint transactions:

$$\begin{matrix} T \vdash B \\ T \vdash A \end{matrix} \{\}, \quad \begin{matrix} T \vdash B \\ B \vdash A \end{matrix} \{\}$$

Chain transactions provide the control structure for realizing pipeline-like computations.

4.3 Reporting Transactions

A variation of the Joint Transaction model is the transaction model in which join does not signify the completion of the joining transaction. A joining transaction continues its execution and periodically *reports* its results to the joint transaction by delegating more objects to the joint transaction. We can call these transactions as *Reporting Transactions*. Reporting transactions must invoke either commit or abort to complete their computation.

The semantics of the joint and report events in the Reporting Transaction model follow. The other significant events are the same as in Joint Transaction model.

- Join Specification
  - Dependency Specification
    $\begin{matrix} T \vdash B \\ T \vdash A \end{matrix} \{A \rightarrow B\}, \begin{matrix} B \vdash A \\ T \vdash B \end{matrix} \{A \rightarrow B\}, \begin{matrix} B \vdash B \\ T \vdash A \end{matrix} \{A \rightarrow B\}, \begin{matrix} B \vdash B \\ B \vdash A \end{matrix} \{A \rightarrow B\}$
  - Delegation Specification
    $DelegateSet_{state}(A, B) = AccessSet_A$

The abort-dependency effectively maintains the completion semantics of joining transactions in the Joint Transaction model by guaranteeing the abortion of the joining transaction A when the joint transaction B aborts. If no abort-dependency were established from joining transaction to joint transaction, then reporting transactions would be neither serializable nor failure atomic. If the abort-dependency were replaced by a commit-dependency of the joint transaction on the joining transaction, it can be shown that reporting transactions would be commit-serializable [20]. Because the production rules prevent A from joining a third transaction, A cannot report to any transaction other than B.

Reporting transactions support an additional significant event, called *Report*, which is used by joining transactions in reporting.

- Report Specification
  - Delegation Specification
    $DelegateSet_{state}(A, B) = ReportSet$ where $A \rightarrow B$ and $ReportSet \subseteq AccessSet_A$

The abort-dependency constraining the delegation ensures that A is a transaction allowed to report to B, since an abort-dependency should already exist from a joining (reporting) transaction A to a joint transaction B.

Reporting transactions can be useful in structuring data-driven computations. Reporting transactions can be restricted to a linear form in a manner similar to chain transactions, or allowed to form more complex control structures.

4.4 Co-Transactions

The characterization of reporting transactions allows A to continue its execution but prevents B from joining A. Suppose A is suspended when it joins B and also B is allowed to join A. A can be suspended, if, at the join, its View Set becomes empty. We call this *ViewSet Curtailment*. A is effectively suspended since after A delegates all the objects in its Access Set to B, due to ViewSet curtailment, A can no longer access any object in the system. A will be able to resume execution when B joins A. This is because after the join A's Access Set will no longer be empty while B will be suspended. We can call these transactions *Co-Transactions* because they behave like *co-routines* in which control is passed from one transaction to the other transaction at the time of the delegation and they resume execution where they were previously suspended. In the Co-Transaction model specified below, View Set of the co-transaction that resumes execution is restored.

Clearly, in the Co-Transaction model, the join event is not a completion significant event and co-transactions must invoke either commit or abort in order to complete their execution.

Here is the characterization of the join event of co-transactions in ACTA:

- Join Specification
  - Dependency Specification
    $\begin{matrix} T \vdash B \\ T \vdash A \end{matrix} \{A \rightarrow B\}, \begin{matrix} A \vdash B \\ B \vdash A \end{matrix} \{A \rightarrow B\}$
  - ViewSet Specification
    $ViewSet_A = \phi$
    $ViewSet_B = DB$
  - Delegation Specification
    $DelegateSet_{state}(A, B) = AccessSet_A$

Co-Transactions are useful in realizing applications that can be decomposed into interactive, and potentially distributed, subtasks which cannot execute in parallel. For instance, co-transactions can be used in setting a meeting between two persons by having one co-transaction executing per person against the individual's calendar database. Co-transactions can be easily modified to form more complex control structures in order to produce more interesting styles of cooperation.

5 Conclusion

ACTA, the comprehensive transaction framework presented in this paper, captures the spectrum of interactions among transactions in competitive and cooperative environments. Each point within the space of interactions is expressed in terms of transactions' effects on the commit and abort of other transactions and on objects' state and concurrency status (i.e., synchronization state).

ACTA allows for specifying the *structure* and the *behavior* of transactions as well as for reasoning about the concurrency and recovery properties of the transactions. The ACTA framework is not *yet* another transaction model, but is intended to unify the existing models. Its ability to capture the semantics of previously proposed transaction models is indicative of its generality. The reasoning capabilities of this framework have also been demonstrated by using the framework to study the properties of new transaction models that are derived by manipulating the Joint Transaction model.

We are currently investigating an ACTA-based formalism that will allow us to precisely characterize the correctness properties of a set of transactions or a transaction model. Such a model will, for example, allow us to determine whether or not the given model produces only serializable computations, and if not, whether the computations are *consistency preserving*, i.e., whether the interactions in the computations do not conflict in such a manner as to produce object inconsistencies. The dependency production rules used in specifying the structure of complex transactions is a step towards this effort.

In order to explore the practical impact of being able to develop new transaction models using this framework, we are also examining the development of a canonical model for implementing object managers and transaction managers to design type specific and transaction specific concurrency control and recovery mechanisms. In this regard, the compatibility table can help an object manager determine whether a transaction can be allowed to perform an operation on an object and if so, what type of dependencies are formed and whether notification is to be performed. The production rules can help the transaction manager determine whether a particular interaction between two transactions is permitted. The Access and View Set related specifications can help the transaction manager determine the visibility of objects to transactions and the changes to be effected when a transaction commits or aborts. Thus, the specifications allowed by ACTA are useful not only in precisely stating the semantics of a transaction model but also to construct the necessary transaction and object management support.

Overall, we believe that our framework will lead to a better understanding of the nature of interactions between transactions and the effect of transactions in environments that require transaction models that are not supported well by the traditional transaction model. Further, with the proposed framework, it is possible to precisely specify the type of interactions and effects allowable in a particular application, and explore ways for achieving cooperation. The concurrency and recovery properties of transactions in the given application can then be studied using the reasoning capabilities built into the framework. Finally, by including an examination of the implementation mechanisms required to support complex transactions within its purview, our work also intends to provide answers concerning the increased complexity entailed by the improved flexibility in constructing complex transaction models.

6 Acknowledgment

The authors would like to thank S. Mazumdar and S. Raghuram for many helpful discussions as well as the reviewers for their useful suggestions.

References

[1] Badrinath B. R. and K. Ramamritham. Semantics-based concurrency control: Beyond Commutativity. In *Proceedings of the 4th IEEE Conference on Data Engineering*, pages 132–140, February 1987.

[2] Badrinath B. R. and K. Ramamritham. Performance Evaluation of Semantics-based Multilevel Concurrency Control Protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 163–172, Atlantic City, NJ, May 1990.

[3] Bancilhon F., W. Kim, and H. Korth. A model of CAD Transactions. In *Proceedings of the 11th International Conference on VLDB*, pages 25–33, Stockholm, August 1985.

[4] Bernstein P. A. and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[5] Chrysanthis P. K. and K. Ramamritham. Capturing the Structure and the Behavior of Complex Transactions. In *Proceedings of the 3rd Workshop on Large Grain Parallelism*, SEI, Carnegie Mellon University, Pittsburgh, October 1989.

[6] Chrysanthis P. K. and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, Atlantic City, NJ, May 1990.

[7] Eswaran K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[8] ernandez M. and S. Zdonik. Transaction Groups: A Model for Controlling Cooperative Transactions. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128–138, January 1989.

[9] Garcia-Molina H.. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2), June 1983.

[10] Garcia-Molina H. and K. Salem. SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, May 1987.

[11] Gray J. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th VLDB Conference*, pages 144–154, September 1981.

[12] Gray J. N., R. A. Lorie, G. R. Putzulo, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared database. In *Proceedings of the 1st International Conference on VLDB*, pages 25–33, Framingham, MA, September 1975.

[13] Herlihy M. P. and W. Weihl. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM symposium on Principles of Database Systems*, pages 201–210, March 1988.

[14] Korth H. F., E. Levy, and A. Silberschatz. Compensating Transactions: A New Recovery Paradigm. In *Proceedings of the the 16th VLDB Conference*, pages 95–106, Brisbane, Australia, August 1990.

[15] Lynch N. A.. Multilevel atomicity — A new correctness for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[16] Moss J. E. B. *Nested Transactions: An approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[17] Papadimitriou C. H.. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[18] Pu C.. *Replication and Nested Transanctions in the Eden Distributed System*. PhD thesis, University of Washington, 1986.

[19] Pu C. From Nested Transactions to Supertransactions. *Bulletin of the IEEE Technical Committee on Data Engineering*, 10(3):19–25, September 1987.

[20] Pu C., G. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended activities. In *Proceedings of the 14th International Conference on VLDB*, pages 26–37, Los Angeles, California, September 1988.

[21] Schwarz P. M. and A. Z. Spector. Synchronizing Shared Abstract Data Types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[22] Shrivastava S.K. and S. M. Wheater. Implementing fault-tolerant distributed applications using objects and multi-coloured actions. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[23] Skarra A. and S. Zdonik. Concurrency Control and Object-Oriented Databases. In *Object-Oriented Concepts, Databases, and Applications*, pages 395–421. ACM Press, 1989.

[24] Vinter S., K. Ramamritham, and D. Stemple. Recoverable Actions in Gutenberg. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 242–249, May 1986.

[25] Weihl W. Commutativity-Based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[26] Zdonik S. B. and D. Maier. *(Ed.). Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.