ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior

Panayotis K Chrysanthis  Krithi Ramamrutham
Department of Computer and Information Science
University of Massachusetts
Amherst MA 01003

e-mail panos@cs3 umass edu, krithi@nirvan cs umass edu

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 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 developed for characterizing the whole spectrum of interactions. The ACTA framework is not yet another transaction model, but is intended to unify the existing models. 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 the commit and abort of other transactions and on objects' state and concurrency status. 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 study the properties of a new model that is derived by combining two existing transaction models.

1 Introduction

The need to support complex information systems emerges from the demands of new and complex applications, such as CAD/CAM, software development environments, object-oriented databases, stock trading databases, and distributed operating systems. These systems are typically distributed and object-based, i.e., designed in terms of an object-oriented paradigm. The ability of transactions to mask the effects of concurrency and failures makes them appropriate building blocks for these complex systems. Although powerful, the transaction model found in traditional database systems [5, 7] is found lacking in functionality and efficiency when used for these new applications. Efficiency is of particular importance considering the throughput demands placed on these complex information systems. In terms of functionality, traditional transactions were assumed to be short-lived and were targeted for competitive environments. 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. Furthermore, endless and collaborating activities which are often found in these systems, cannot be captured by traditional transactions due to serializability as the correctness requirement. Therefore, the need to capture reactive (endless), open-ended (long-lived) and collaborative (interactive) activities found in the new applications suggests the need for more cooperative models. 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.

In order to fill the need for more flexible transaction models, various extensions to the traditional model have been proposed, referred to herein as complex transactions, which can support the implementation of efficient systems. For example, Nested Transactions [11] have been proposed in the context of distributed languages to handle the problem of partial failures. Nested Transactions support only hierarchical computations similar to the ones that result from procedure calls. Recoverable Communicating Actions [18] which support arbitrary computation topologies, have been proposed in the context of distributed operating systems where interactions are more complex. Cooperative Transactions [3], Split Transactions [14] and Transaction Groups [6, 17] have also been suggested for capturing the interactions found in the new applications. Irrespective of how successful these extended transaction models are in supporting the systems that they were intended for, they merely represent points within the spectrum of interactions possible within competitive and cooperative environments. Therefore, they can capture only a subset of the interactions to be found in any complex information system.

While it is tempting to develop new transaction models that cover some of the remaining points in the spectrum, any such work will by necessity be ad hoc and not general. What will be better is to study the nature of transactions...
as such and develop a conceptual framework in which it will be possible to specify the effects of complex transactions and then reason about their properties.

We have developed such a comprehensive transaction framework, called ACTA\(^1\), 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 objects that they access. A transaction has two possible outcomes, namely, commit or abort. Consequently, the effects of one transaction on other transactions are classified as those on the abort of other transactions and those on the commitment of other transactions. The effects of a transaction on the objects that it accesses are also categorized into two classes: the effects of a transaction on the state of objects and the effects of a transaction on the concurrency status, i.e., synchronization state, of objects (Henceforth, we refer to concurrency status as just status).

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. Structure refers, for example, to the nesting structure of a transaction, and behavior refers to the operations invoked by a transaction.

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 tested by using the framework to study the properties of a new transaction model, called Nested-Split Transactions, that is derived by combining the Nested and Split Transaction models.

In Section 2, we examine the characteristics of complex transactions. In Section 3, we present ACTA, our proposed comprehensive transaction framework and discuss the intuition underlying the framework. Section 4 illustrates the use of the framework by applying it to model four existing transaction models. In the same section, the reasoning capabilities of the framework are demonstrated by studying the properties of the Nested-Split transaction model. Section 5 concludes with a summary and discusses future steps.

2 Complex Transactions: Definition and Issues

Traditional transactions 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.

Complex transactions have properties which relate to the above notions. Generally, complex transactions can be said to consist of either a set of operations on objects or a set of complex transactions. 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. The simplest example of complex transactions is Nested Transactions [11].

Complex transactions are distinguishable from the multilevel transactions [12, 10, 1] first in that their internal structure is explicit and provided as a user facility, and second in that their component transactions are not necessarily atomic. Multilevel transactions have an implicit hierarchical internal structure which is a result of transactions invoking operations on complex objects. Thus, the operations are decomposable into sub-operations. Both operations and sub-operations are considered atomic. That is, for the user, a multilevel transaction is nothing but a set of atomic operations similar to a traditional transaction, and nesting is provided as a system facility.

The way that component transactions are combined to form complex transactions reflects the semantics of the applications. 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 [2, 15, 9, 19].

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 committed, 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 preserves consistency. 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 by attempting another alternative, or even ignored. Furthermore, complex transactions naturally support user-controlled checkpointing since the boundaries of component transactions act as checkpoints.

The above observations motivate us to address the following questions in our research:

- **How do we capture the semantics of complex transactions?**
- **How can we reason about the concurrency and recovery properties of complex transactions?**

The ACTA framework described in the next section is our initial response to these questions. As we shall see, this framework allows us to capture transaction properties as related to the dimensions of (i) visibility, (ii) failure atomicity (recovery), (iii) permanence, and (iv) consistency.

### 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. This taxonomy of effects is captured in Figure 1.

These two types of effects as well as the formal means to specify them are described in Section 3.1 and 3.2 respectively.

The application of ACTA to various transaction models, in Section 4, should serve to clarify and illustrate the concepts underlying ACTA.

### 3.1 Effects of Transactions on other Transactions

Dependencies provide a convenient way of specifying and reasoning about the behavior of concurrent transactions [8, 15]. By examining the possible effects of interacting transactions on each other, it is possible to determine the dependencies that may develop between the transactions. There are two possible dependencies that a transaction may develop on any other transaction: **Commit-dependency** and **abort-dependency**.

- **Commit-dependency**: If a transaction \( A \) develops a commit-dependency on another transaction \( B \) (denoted by \( A \rightarrow B \)), 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.

- **Abort-dependency**: If a transaction \( A \) develops an abort-dependency on another transaction \( B \) (denoted by \( A \leftarrow B \)), 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.

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 form a circular dependency involving the same type of completion dependency, then both have to commit or neither. 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. This does not imply that both transactions have to commit or neither as in the case above.

Completion 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. It is often necessary for dependencies induced by the structure of transactions to be qualified either to further strengthen them by attaching to them more restrictions, or to restrict the scope of their applicability by attaching conditions. As an example of the former, abort-dependency can be restricted so that a transaction is not...
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 the Access Set, which contains all the objects potentially accessible to the transaction. Rules for composing the Access Set are determined by the specific transaction model. Examples are given in Section 4.

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 the View Set. When an object in the View Set of a transaction is committed, the object becomes a member of the transaction's Access Set. Objects in Access Set continue to be accessible to the transaction. An object ob in the View Set of a transaction T1 can be accessed by T1 only if the concurrency control status of ob permits it. For instance, if ob is in the Access Set of another transaction T2 and the concurrency control status of ob indicates that the operation that T2 uses to access ob is incompatible with the operation that T2 used to access ob then T1 will not be allowed to access ob and hence, ob does not become a member of the Access Set of T1. 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.

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 effectual, in the View Set, while the status is restored in the View Set. When a transaction T1 makes access to a transaction T2, AccessSetT1 refers to the Access Set of a transaction T2, and ViewSetT1 refers to the View Set of T2.

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 it 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. This type of delegation is referred to as delegation of status. 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. The notion of inheritance used in Nested Transactions is an instance of delegation. Specifically, inheritance as proposed in [11] corresponds to the delegation of state when the delegator commits, whereas in [13] corresponds to the delegation of status when the delegator aborts and delegation of
state when the delegator commits $\text{DelegateSet}_\text{state}(T_1, T_2)$ and $\text{DelegateSet}_\text{state}(T_1, T_2)$ 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.

Another form of delegation is limited delegation which makes the changes to the delegated objects persistent in the View Set before adding them to the Access Set of the target transaction.

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 notifying 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 Modeling Different Transaction Schemes

In this section, the semantics of four transaction models are specified using the ACTA framework. These are Nested Transactions, Split Transactions, Recoverable Communicating Actions and Cooperative Transactions. Because of space limitations, the characterization of Transaction Groups [6, 17] and Multi-Coloured Actions [16] are not included in this paper. Also, the properties of a new transaction model resulting from the combination of Nested Transactions and Split Transactions are studied in order to demonstrate the usefulness of our framework in reasoning about the properties of existing and future transaction models.

Throughout this section, the set $DB$ 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.

4.1 Nested Transactions

In the Nested Transaction model [11], transactions are composed of subtransactions or child transactions designed to localize failures within a transaction and to exploit parallelism within transactions. A subtransaction can be further decomposed into other subtransactions, and thus, the transaction may expand in a hierarchical manner. Subtransactions execute atomically with respect to their parent and their siblings, and can abort independently without causing the abortion of the whole transaction. However, if the parent transaction aborts, all its subtransactions have to abort. The parent transaction cannot commit until all its subtransactions have terminated.

A subtransaction can potentially access any object that is currently accessed by one of its ancestor transactions. In addition, any object in $DB$ is also accessible to the subtransaction. When a subtransaction commits, its objects are made accessible to its parent transaction. However, the effects on the objects are made persistent in $DB$ only when the root transaction commits.

Here is the characterization of Nested Transactions in the ACTA framework. We use $C$ to denote a child transaction of a parent transaction $P$.

- Dependency Specification
  \[ \forall C \text{ } C \not\subseteq P \]
  \[ \forall C \text{ } P \not\rightarrow C \]

  The abort-dependency of a child on its parent guarantees the abortion of the child transaction in case its parent aborts. Furthermore, the exclusive-abort-dependency prohibits a child transaction from having more than one parent, this ensures the hierarchical structure of the nested transactions.

  The commit-dependency of the parent on its children guarantees that the parent does not commit before all its children have terminated.

- View Set Specification
  \[ \forall C \text{ } \text{ViewSet}_C = \{ \cup \text{AccessSet}_A | C \not\subseteq A \} \cup DB \]

  In our notation, $\cup$ is an ordered union. More precisely, 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.

  The ability of a subtransaction to access any object currently 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. The transitive-abort-dependency uniquely specifies the ancestors of a subtransaction.

- Delegation Specification
  Delegation occurs when $C$ commits
  \[ \forall C \text{ } \text{DelegateSet}_\text{state}(C, P) = \text{AccessSet}_C \]

  The delegation specification states that, at commit, the child transaction's objects are delegated to its parent transaction. This effectively makes the effects of the committing child transaction selectively visible to its parent and to the parent's descendants (by the View Set specification above).

4.2 Split Transactions

In the Split Transaction model [14], it is possible for a transaction $A$ to split into two transactions, $B$ and $C$, where $B$ is the original transaction. $B$ and $C$ transactions may be independent, in which case they can commit or abort independently, or they may be serial, in which case $B$ must commit in order for the $C$ to commit. Whether $B$ and $C$ transactions are independent or serial depends on the objects accessible to them.

\footnote{In the case of [13] the delegation specification should state in addition Delegation occurs when $C$ aborts, \[ \forall C \text{ } \text{DelegateSet}_\text{state}(C, P) = \text{AccessSet}_C \]}

In the Spht Transaction model [14], it is possible for a transaction $A$ to split into two transactions, $B$ and $C$, where $B$ is the original transaction. $B$ and $C$ transactions may be independent, in which case they can commit or abort independently, or they may be serial, in which case $B$ must commit in order for the $C$ to commit. Whether $B$ and $C$ transactions are independent or serial depends on the objects accessible to them.
4.2.1 Independent Split Transactions

Here is the characterization of independent Split Transactions in the ACTA framework.

- Delegation Specification
  Delegation occurs when A splits
  \[ \text{AccessSet}_B = \text{AccessSet}_A - \text{DelegateSet}(A, C) \]
  \[ \text{AccessSet}_C = \text{DelegateSet}(A, C) \]

  The independence of the two transactions is guaranteed by having B and C operate on disjoint sets of objects. Delegation leaves C the responsibility of making persistent all the changes made by A to delegated objects up to the split.

4.2.2 Serial Split Transactions

Now we characterize Serial Split Transactions which have more complicated semantics than independent Split Transactions.

- Dependency Specification
  \[ C \xrightarrow{B} B \]

  The abort-dependency guarantees that transaction C aborts if B aborts and that C's commitment is delayed until B commits. The exclusive-abort-dependency prevents C from joining (see below) a third transaction. Note that this does not prevent transactions B and C from joining.

- View Set Specification
  Split Transactions were proposed in the context of the Read-Write database model. Hence, the View and Access sets of B and C can be specified in terms of the set of objects that they can read or write (e.g., \( \text{ViewSet}_B = \text{ViewWrite}_B \cup \text{ViewRead}_B \) and \( \text{AccessSet}_B = \text{ReadSet}_B \cup \text{WriteSet}_B \)).

\[ \text{ViewWrite}_B = \text{ViewWrite}_A \]
\[ \text{ViewRead}_B = \text{ViewRead}_A \]
\[ \text{ViewWrite}_C = \{ x \in \text{WriteSet}_B \wedge C.\text{WriteLast}(x) \} \cup DB \]
\[ \text{ViewRead}_C = \{ x \in \text{WriteSet}_B \wedge C.\text{CanRead}(x) \} \cup DB \]

WriteSet\(_B\) contains the objects that A has changed up to the split and may change after the split when executing as B. That is, WriteSet\(_B\) is a subset of the Access Set of A. The C.WriteLast specifies the objects that can be updated last by C. Similarly, C.CanRead specifies the objects that C can read but they are not delegated to C.

The ViewWrite\(_C\) (ViewRead\(_C\)) contains all the objects that C can potentially write (read) after the split. In this way, some of the changes to the objects up to the time of the split become visible to C. Note that this is not supported by the original notion of Split Transactions [14], although it might be appropriate for some applications.

- Delegation Specification
  This delegation occurs when A splits
  \[ \text{AccessSet}_B = \text{AccessSet}_A - \text{DelegateSet}(A, C) \]
  \[ \text{AccessSet}_C = \text{DelegateSet}(A, C) \]

  Any changes to objects in DelegateSet up to the split are left to C to be made persistent to the database.

- Delegation Specification
  This limited delegation occurs when B commits
  \[ \text{DelegateSet}(B, C) = \text{AccessSet}_B \cap \text{ViewSet}_C \]

  B delegates limited responsibility of the objects to C that C could potentially access but C did not. In this way, all the changes to these objects are made persistent to the database while C still has access to these objects.

  If B aborts then C is also aborted, given that C has an abort-dependency on B. All the objects acquired by both transactions are restored to the system.

  By comparing the characterizations of the independent and serial split transaction in ACTA, one can infer that the source of the abort-dependency in the case of serial split transactions is due to the View Set Specification and in particular, to the information flow allowed by the View Set Specification. A closer study of the View Set Specification reveals that the case that C is not allowed to read any object that is not delegated to it or is not in the database (e.g., ViewRead\(_C\) = DB), the abort-dependency of C on B can be substituted by a commit-dependency which avoids cascading aborts while still ensuring serial commit of B and C.

4.2.3 Joint Transactions

In the Split Transactions model, it is also possible for two transactions to join into one. This is called the joint transaction. The joint transaction is either of the original ones when the transactions join, they release their objects to the joint transaction.

The characterization of Joint Transactions in the ACTA framework is straightforward.

- Dependency Specification
  \[ A \xrightarrow{B} B \]

- Delegation Specification
  Delegation occurs when A commits
  \[ \text{DelegateSet}(A, B) = \text{AccessSet}_A \]

  The abort-dependency effectively joins transactions A and B, and indicates that B is the Joint transaction which continues executing. The exclusive-abort-dependency prevents A from joining a transaction other than B.

  The above characterization points to a variation of the Joint Transactions in which the delegation does not occur when A commits. That is, A can continue its execution and can periodically report its results to B by delegating more objects to B. We can call these transactions as Reporting Transactions.
4.3 Nested-Split Transactions

In order to test the reasoning capabilities of the framework, we created a new model by combining the Nested and Split Transaction models presented in the previous two sections. The framework was then used to check whether this new model retains the properties of the two original models.

Note that, given a nested transaction, it is possible to split a leaf node, an internal node, or a root node. The split nodes could execute independently or serially. Figure 2 captures the effects for all possible combinations. The dependencies shown follow from the specifications of dependencies for nested and split transactions. In these figures, a dotted arrow denotes a commit-dependency and a solid arrow denotes an abort-dependency.

When a node, say C (Figure 2b), splits into two subtransactions, say C1 and C2, where C1 is the original subtransaction of C, the dependencies between subtransaction C and transaction A are assumed to hold between C2 and A. Since both nested and split transactions involve exclusive-abort-dependencies (recall that exclusive-abort-dependency prevents a transaction from having an abort dependency on more than one other transaction), a node splitting may result in a subtransaction that has exclusive-abort-dependencies on two other subtransactions (Figure 2b, After the Serial Split).

Such inconsistencies may be resolved by means of consistency preserving rewrite rules. In general, consistency preserving rewrite rules are used to simplify the structure of a complex transaction by eliminating redundant dependencies. Figure 2a shows four such rewrite rules of which re-write 2 resolves the inconsistency mentioned above.

After applying the rewrite rules (in these cases only re-write 2 is applicable), we examine the remaining dependencies for each type of nested-split transaction to see if the resulting structure preserves the semantics of the Nested and Split transactions models. We conclude that in only one case the properties of the two models are preserved. This case involves the splitting of the leaf node into two independent subtransactions. In all other cases the model either establishes dependencies which destroy the structure of the nested transactions or eliminates some of the dependencies required by the nested transactions. For example, in figure 2b (After Applying Re-write Rule), the exclusive-abort-dependency of subtransaction C2 on subtransaction A is eliminated.

Even if splitting of nodes is restricted only to the independent splitting of leaf nodes, nested-split transactions is a useful new transaction model in a cooperative environment. Observe that an internal node becomes a leaf node any time that it has no active child subtransactions. That is, in nested-split transactions, a node may split at any point after all its child subtransactions have terminated and before activating any new subtransactions. For example, in figure 2c (Initial Nested Structure), when subtransaction D terminates, node C can be split into two independent subtransactions C1 and C2 as in Figure 2b (After the Independent Split of C). C1 may continue the execution of C spawning new subtransactions, while C2 may commit delegating its objects to A. Since all the objects accessible to A are potentially accessible to all of its descendants (by View Set Specification of nested transactions), the objects delegated to A by C2 are potentially accessible to B. This effectively achieves cooperation between the original siblings C and B while they are still executing.

In nested transactions, two siblings cannot cooperate while both siblings are active, since subtransactions delegate their objects to their parent only at commit time. Thus, nested-split transactions support higher level of visibility between subtransactions than nested transactions do.

This exercise showed us the efficacy of the ACTA framework in determining the properties of new transaction models. In this case, one derived by combining existing models.

4.4 Recoverable Communicating Actions

In the context of long and cooperative transactions, the Recoverable Communicating Actions (RCA) model has been proposed to deal with the problem of non-hierarchical computations [18]. In this model, an action, the sender, is allowed to communicate with another action, the receiver, by exchanging objects, resulting in an abort-dependency of the receiver on the sender. If the sender aborts then the receiver must abort as a result of the dependency.

By developing abort-dependencies, RCAs form a recoverable computation, a self-contained task or activity which has the semantics of an atomic update. For this reason, actions belonging to the same recoverable computation require synchronized commitment. That is, in the case of a sender which has no dependencies on any other action, the sender cannot commit independently. However, partial failures are tolerated since an action may abort without aborting the action with which it has developed an abort-dependency. In short, a recoverable computation can dynamically expand through the development of dependencies and shrink due to abortion of actions.

Here is the characterization of RCAs in ACTA.

- **Dependency Specification**
  
  \[
  \text{Receiver} \rightarrow \text{Sender} \\
  \text{Sender} \leftarrow \text{Receiver}
  \]

  The circular dependency involving different completion dependencies between sender and receiver guarantees the required synchronized commitment of the sender and receiver actions.

  The abort-dependency guarantees that the effects of aborted actions are not reflected in the database. Neither the abort nor the commit dependencies prevent an action from developing any new dependencies. It is even possible for an action to be both a sender and a receiver at the same time. In this manner, RCAs can produce non-hierarchical structures.

- **View Set Specification**
  
  \[
  \text{View Set} \text{Receiver} = \{z | \text{Received}(z)\} \cup \text{DB}
  \]

  **Received(z)** specifies that a sender transferred object z to the receiver, where, \( z \in \text{Access Set} \text{Sender} \).

  Given the complete characterization of Split and RCA models in ACTA, one can immediately observe that the two models involve different completion dependencies. This difference is sufficient to demonstrate that one model does not subsume the other. Another difference is that the notion of delegation does not exist in RCAs. Just as in the case.
After the Independent Split of C

Before Applying Re-Write Rule

After Applying Re-Write Rule

After the Serial Split of C

After the Serial Split of A

Splitting a Nested Transaction

Figure 2 Splitting a Nested Transaction

201
of nested-split transactions, using ACTA is easy to demonstrate that in spite of these differences the two models are compatible, in the sense that it is possible to combine and use them.

4.5 Cooperative Transactions

Cooperative Transactions [3] were proposed in the context of CAD/CAM and design applications supported by the checkin/checkout access date model. In the Cooperative Transaction model, transactions are decomposed into subtransactions, each with its own semantics and types. The model supports three distinct types of subtransactions: project transactions are decomposed into cooperative transactions, subcontractor transactions are composed of a set of subcontractor transactions, and subcontractor transactions may either have a structure similar to cooperative transactions in which case the client cooperative transaction acts as a local project transaction, or have the structure of an atomic transaction called short transactions.

Cooperative transactions have a hierarchical structure similar to nested transactions, but they do not support object inheritance in the same manner as in nested transactions. In cooperative transactions object flow is supported only between adjacent levels through intermediate semi-public or subcontractor databases. This does not imply that the transactions are prevented from accessing objects in the database. A semi-public database is similar to a subset of an Access Set in the ACTA framework.

The hierarchical structure of cooperative transactions is expressed using the universally quantified completion dependences.

Cooperative transactions also support situations in which there is a partial ordering that constrains the acceptable orderings of subcontractor and short transactions execution. These situations can be easily expressed via commitment dependencies in ACTA. For example, if the 3rd subcontractor of the 4th cooperative transaction should happen before the 4th one, these semantics can be specified as

\[ \text{Contractor}_{14} \rightarrow \text{Contractor}_{13} \]

\* View Set Specification

\[ \text{ViewSet}_{\text{Project}} = \text{DB} \]
\[ \text{ViewSet}_{\text{Coop}} = \{ \text{AccessSet}_{\text{Project}}(\text{Coop} \rightarrow \text{Project}) \} \cup \text{DB} \]
\[ \text{ViewSet}_{\text{Short}} = \{ \text{DesignersSet}_{\text{Coop}}(\text{Short} \rightarrow \text{Coop}) \} \cup \text{DB} \]
\[ \text{ViewSet}_{\text{Contractor}} = \{ \text{ContractorsSet}_{\text{Coop}}(\text{Contractor} \rightarrow \text{Coop}) \} \cup \text{DB} \]
\[ \text{AccessSet}_{\text{Coop}} = \text{DesignersSet}_{\text{Coop}} \cup \text{ContractorsSet}_{\text{Coop}} \]
\[ \text{DesignersSet}_{\text{Coop}} \cap \text{ContractorsSet}_{\text{Coop}} = \phi \]

Objects in the DesignersSet can only be moved to ContractorsSet and vice versa by the cooperative transaction whose Access Set is formed by these sets.

The View Set definitions of the short and subcontractor transactions specify that these transactions can only access objects currently accessed by their parent transaction. The View Sets of the short and subcontractor transactions are further constrained to be the DesignersSet and ContractorsSet respectively. That is, the View set of a short transaction is the DesignersSet, a subset of the Access Set of its parent cooperative transaction. The View Set of a subcontractor transaction is the ContractorsSet, a subset of the Access Set of its parent cooperative transaction.

\* Delegation Specification

Delegation occurs when a subtransaction commits

\[ \forall i, j \text{ DelegateSet}_{\text{state}}(\text{Coop}_i, \text{Project}) = \text{AccessSet}_{\text{Coop}_i} \]
\[ \forall i, j \text{ DelegateSet}_{\text{state}}(\text{Short}_j, \text{Coop}_i) = \text{AccessSet}_{\text{Short}_j} \]
\[ \forall i, j \text{ DelegateSet}_{\text{state}}(\text{Contractor}_j, \text{Coop}_i) = \text{AccessSet}_{\text{Contractor}_j} \]

Delegation occurs when a subtransaction aborts

\[ \forall i, j \text{ DelegateSet}_{\text{state}}(\text{Coop}_i, \text{Project}) = \text{AccessSet}_{\text{Coop}_i} \]
\[ \forall i, j \text{ DelegateSet}_{\text{state}}(\text{Short}_j, \text{Coop}_i) = \text{AccessSet}_{\text{Short}_j} \]
\[ \forall i, j \text{ DelegateSet}_{\text{state}}(\text{Contractor}_j, \text{Coop}_i) = \text{AccessSet}_{\text{Contractor}_j} \]

As in the case of nested transactions, the delegation specification states that at commit, the child (cooperative, short, or subcontractor) transaction's objects are delegated to its parent (project, or cooperative) transaction. However, in the case of short and subcontractor transactions the delegated objects are added to the respective DesignersSet and ContractorsSet (by the View Set specification above). This effectively makes the effects of the committing child transaction selectively visible to its parent, and forces the parent's short transactions to cooperate through the objects contained in the DesignersSet and the parent's subcontractors to cooperate through objects in the ContractorsSets.

In the case of abort, the child transaction's objects are delegated to its parent after the state changes done by the child on the objects are nullified.

5 Conclusion

ACTA, the comprehensive transaction framework proposed 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 a new model that is derived by combining the Nested and Split transaction models.

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 semantically-computable 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.

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.

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 should be 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.

References


