ACTA: A Comprehensive TransAction Model

Panayiotis K. Chrysanthis
Krithi Ramamritham
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 89-27

ACTA: A Comprehensive TransAction Model [*]

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 current and new applications such as CAD/CAM, software development environment, distributed operating systems and semi-automated factories. However, these extended models capture only a small set of interactions that can be found in any system, and represent points within the spectrum of interactions possible within competitive and cooperative environments.

ACTA is a comprehensive transaction model that characterizes the spectrum of interactions along four dimensions: *completion dependencies* which express the actions' effects on the disposition (commit or abort) of other actions, *communication obligations* which express the interactions among actions due to actions' effects on the objects, *in-progress effects* which express the effects of actions on the state and status (synchronization state) of objects during their execution, and *termination effects* which express the terminating actions' effects on the state and status of objects. The ACTA model is *not* yet another transaction model, but is intended to unify the existing models. Its ability to model previously proposed transaction-based schemes is indicative of its generality. By capturing a broad spectrum of interactions, the ACTA model provides the basis for the integration of existing schemes and for modeling or developing new ones.

This paper serves as an introduction to the ACTA model and discusses the intuition underlying the model.

[*]This material is based upon work supported by the National Science Foundation under grant CCR-8500332

1 Introduction

Systems, such as CAD/CAM, software development environments, distributed operating systems and semi-automated factories are built around an information system or closely interact with one. Majority of these are distributed. Cooperation among specific subsystems is an important concern in these systems. However, the model of coordination found in traditional database systems [Eswaran76, Gray81], although powerful, is not suitable for them. The reasons are related to both functionality and efficiency. This is because transaction models used till recently were targeted for competitive environments. However, the need to capture reactive (endless activities), open-ended (long-lived) and collaborating (interactive) activities found in complex information systems which support the new applications suggests the need for more cooperative models. Broadly speaking, whether a system is competitive or cooperative depends on how *communication* among activities in the system is viewed: In competitive environments, communication is curtailed whereas it is promoted in cooperative ones.

Various extensions to the traditional model have been proposed to fill this need for a more flexible transaction model which can support the implementation of efficient systems. For example, Nested Transactions [Moss81] 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. On the other hand, Recoverable Communicating Actions [Vinter86] which support arbitrary computation topologies, have been proposed in the context of distributed operating systems where interactions are more complex. Cooperative Transactions [Bachilon85], Split-transactions [Pu88] and Transaction Groups [Fernandez89] have been also 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 nevertheless represent points within the spectrum of interactions defined within competitive and cooperative environments. Thus they can capture only a small set of interactions that can be found in any system.

ACTA[1] is a comprehensive model which captures the semantics of interactions within the whole spectrum of competitive and cooperative environments along four dimensions: completion dependencies, communication obligations, in-progress effects and termination effects.

Actions' effects on other actions are expressed by *completion dependencies* which portray the effect of the commit or abort of one action on another. *Communication obligations* specify the actions' effects on other actions due to changes to the *state* (i.e. contents) of the objects, and/or on the *status* (i.e. synchronization state) of the objects which they access. *In-progress effects* express the changes to the state and/or status of the objects caused by actions during their execution. The effects of actions at the time of commit or abort on the state and/or the status of the objects are described by *termination effects.*

It should be noted that the ACTA model is *not* yet another transaction model. It is intended to be a comprehensive model which captures all types of cooperative and competitive interactions. In this sense, it is motivated by a need to unify the existing models and to provide a framework in which to express new ones. The completeness of the ACTA model is indicated by its ability to model all major previously proposed schemes. By capturing a broad spectrum of interactions, the ACTA model provides the basis for the integration of existing schemes and for modeling or developing new ones.

After introducing the ACTA model in section 2, section 3 is devoted to the modeling of four existing transaction models. Section 4 concludes with a summary and discusses future steps.

This paper aims to present just the intuition underlying the ACTA model in an informal manner. A formal model of ACTA is currently under investigation. Such a model will allow us to characterize the correctness properties of a given transaction model.

[1]ACTA means *action* in Latin.

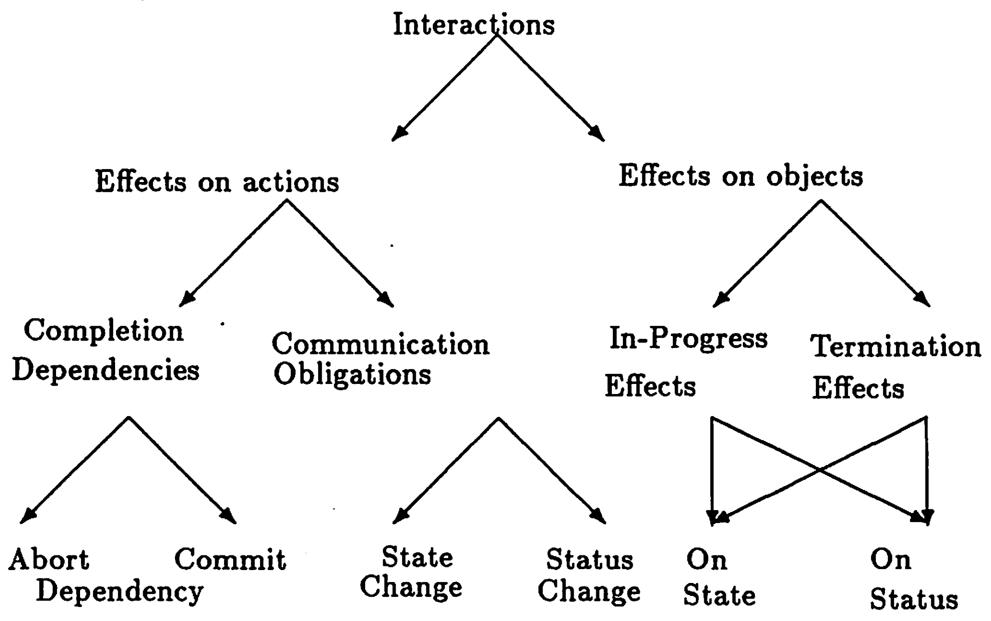

Figure 1: Taxonomy of Actions' Effects

2 The ACTA Model

In the ACTA model, *atomic actions,* or just *actions* are the units of recovery and consistency. An action either *commits*, making its effects on the objects permanent; or *aborts*, causing its effects to be discarded.

A self-contained task or activity is called a *computation.* Thus, a computation consists of a set of interacting atomic actions executing concurrently. A computation *terminates* when all of its component actions *terminate*, i.e. commit or abort. When a computation aborts, all the effects of its component actions are nullified.

A computation may dynamically expand and shrink through the addition or deletion of actions that contribute to the computation. The simplest computation is one with a single action. The interactions between actions in the same or different computations can be expressed in terms of actions' effects on each other and on the objects that they access. Each of them can be further refined as shown in Figure 1.

2.1 Effects of Actions on other Actions

Interacting actions are coordinated by developing *completion dependencies* and *communication obligations* among them. Completion dependencies constrain the disposition of interacting actions. Communication obligations capture the interactions among actions because of the changes made to the state or the status of objects accessed by the actions.

2.1.1 Completion Dependencies

The disposition of an action may depend upon the disposition of another action with which it has interacted. Since an action has two possible dispositions, namely, commit or abort, the action may be affected in only one of two ways by another action. *Commit-dependency* and *abort-dependency* express these two possible ways.

Commit-dependency and abort-dependency are collectively known as *completion dependencies* and are defined as:

Commit-Dependency: If an action A develops a *commit-dependency* on another action B (denoted by $A \overset{c}{\Rightarrow} B$), then action A cannot commit until action B either commits or aborts. This does not imply that if action B aborts, then action A should abort.

Abort-Dependency: If an action A develops an *abort-dependency* on another action B (denoted by $A \overset{a}{\Rightarrow} B$), and if action B aborts, then action A should also abort. This neither implies that if action B commits, then action A should commit, nor that if action A aborts, then action B should abort.

The purpose of commit-dependency and abort-dependency is to prevent an action from prematurely committing, thereby preventing object inconsistencies. However, they do not prohibit an action from *attempting to commit.*

Clearly, if two actions form a circular dependency involving the same type of action dependency, then both have to commit or neither. In the case that two actions develop a circular dependency involving dependencies of different types, i.e. one action has a commit-dependency on another action which has an abort-dependency on the first action, then the commitment of both actions must be synchronized. This does *not* imply that both actions have to commit or neither as in first case above.

Transitive-abort-dependency (denoted by $\overset{a*}{\Rightarrow}$) is defined by the transitive closure of abort-dependencies. An action A has a transitive-abort-dependency on every member of the set of actions formed by the transitive closure of abort-dependencies starting from A. *Transitive-commit-dependency* (denoted by $\overset{c*}{\Rightarrow}$) is similarly defined.

Both abort-dependency and commit-dependency can be qualified either to further strengthen them by attaching to them more restrictions, or to restrict the scope of their applicability by attaching to them conditions (*conditional-dependency*). As an example of the former, abort-dependency can be restricted so that an action is not allowed to develop an abort-dependency on more than one action. This stronger version of abort-dependency is called *exclusive-abort-dependency* (denoted by $\overset{a_1}{\Rightarrow}$) and is useful in controlling the expansion of a computation.

An example of conditional-dependency is the case of a *conditional-abort-dependency* where the abort-dependency of an action A on an action B holds as long as there is a commit-dependency of the action A on a third action C ($A \overset{a}{\Rightarrow} B$ *while* $A \overset{c}{\Rightarrow} C$). As soon as the condition (in this case $A \overset{c}{\Rightarrow} C$) becomes false, the other clause of the conditional-dependency also becomes inactive. Generally, a clause in a conditional-dependency may be an expression consisting of a number of action dependencies and/or other conditions connected by boolean operators. As will become apparent in section 2.2.3, the importance of conditional-dependency is that it allows a computation to expand while preventing interactions in a computation that may produce object inconsistencies.

2.1.2 Communication Obligations

For actions to coordinate themselves while interacting via shared objects, they should be cognizant of any changes to the *state* and/or the *status* of these objects. The state of an object is represented by its contents. The status of an object is represented by the synchronization information associated with the object. The state of an object changes when its contents changes. The status of an object changes when an action either tries to access an object or releases the object.

Communication obligations are introduced to allow actions to make each other aware of changes made to the state or status of objects.

Obligation with respect to State Changes: If an action A has such an obligation relative to action B over an object O_i (denoted by $A \overset{s\ O_i}{\Longrightarrow} B$), A is obligated to inform B about any change to the state of O_i caused by A.

Obligation with respect to Status Changes: If an action A has such an obligation relative to action B over an object O_i (denoted by $A \overset{l\ O_i}{\Longrightarrow} B$), A is obligated to inform B about any change to the state of O_i caused by A.

Communication obligations, in contrast to completion dependencies, have no *direct* effect on the disposition of either the object or the actions involved.

2.2 Effects on Objects

The second dimension in the ACTA model expresses the effects of actions on the objects. The effects on objects are further distinguished based on whether the effect is caused by an *in-progress* action or a *terminating* action. These effects may reflect on the state and/or the status of the object.

To simplify the specification of these effects on the state and the status of the objects, we first introduce the notion of *virtual databases.* The *system database* is the repository of all objects. For changes to an object to be permanent, they should be committed to the system database. An object that is currently accessed by an action may also belong to a specific *virtual database.* It is possible for an object to belong to a different virtual database for different actions. That is, an object in different virtual databases may have different states. Furthermore, only a subset of operations supported by an object may be invokable in different virtual databases. In general, it is possible to have a many-to-many relationship between actions and virtual databases since an action may access objects in multiple virtual databases and objects in a particular virtual database may be accessed by multiple actions. Actions execute against the system database as well as a set of virtual databases.

To simplify the discussion of the model in this paper, we present the model assuming that only one version is maintained for each object in the *system database*. That is, a computation can produce only a single final version for each object which it has accessed. However, since an object may belong to multiple virtual databases while a computation is active, it may produce a number of versions of the objects each of which can be accessed by different actions belonging to the computation. The model does not become overly complicated when the system database maintains multiple versions for the objects and supports actions' request to access previous versions of an object.

2.2.1 In-progress Effects

Every object is associated with two operations that change its status. These operations are in addition to the ones defined by the object to manipulate its state. These operations, namely, *acquire* and *release*, control the accessibility to the object.

Acquire (`acquire(object, operation[s], condition[s])`) allows an action to gain access to an object for an operation or a set of operations. In order to control the interaction over an object, an action may attach conditions on the object when acquiring it. These conditions may specify, among others, which operations are prohibited from concurrently acquiring the object and whether completion dependencies or communication obligations should be developed between actions invoking two specific operations on the object. For example, the shared read type interaction is specified as "acquire an object for *read* with the condition *no-write*," i.e. no other action can acquire the object for the write operation. The same action may subsequently acquire the object for write as long as no other action has acquired the object and set the no-write restriction on it. That is, the conditions set by one action are applicable only to other actions. The conditions imposed by one action do not invalidate the conditions imposed by another action on the same object. However, the stronger condition at any given time is the one applicable. The conditions set by an action on an object are valid until the object is *released.*

Release (`release(object, where, operation[s], condition[s])`) allows an action to make some of its objects accessible to some virtual database, and/or to specific operations. After an action releases an object, the action cannot invoke an operation on the object unless it has re-acquired the object for that operation. An action makes an object accessible to a specific action or to a set of actions by releasing the object to the appropriate virtual database. The database from which an action acquires the object is called the *origin* and the virtual database to which the object is released is called the *target.* An action may attach conditions on a released object similar to those attached at the time of acquiring an object in order to control the future disposition of the object. For example, the semantics of basic two-phase locking protocol [Bernstein87] are expressed as follows: (i) every action A that performs a read operation on an object should release it with the condition that any action B which subsequently acquires the object for write should develop a commit-dependency on A; and, (ii) every A performing a write operation on an object should release it with the condition that any action B which subsequently acquires the object for read should develop an abort-dependency on A, and any B which subsequently acquires the object for write should develop a commit-dependency on A.

Further conditions may be set by some action at either the time of acquisition or release of the object. Such conditions may specify (i) whether the object should be released back to the origin virtual database, (ii) whether the object should be released to some specific virtual database and (iii) how the objects *propagate.* Object propagation deals with the disposition of the objects in a virtual database when the last action that has access to the virtual database, terminates. In particular, object propagation conditions specify the virtual database to which such an object should be moved.

Depending on the conditions on the acquisition of an object, the release of the object to a different virtual database has the semantics of moving or copying the (changed) object to that virtual database. Specifically, if an action acquires that object with conditions prohibiting any other action from accessing the object, the release behaves like a move. On the other hand, if the object is still accessible by other actions in the origin database, the release behaves like a copy.

The state of an object changes when an operation that alters the contents of the object is invoked on the object. An action can make its changes to an object visible to other actions by releasing the object. This may or may not lead to a non-serial behavior depending on the specifications of the other dimensions in the ACTA model, and in particular of completion dependencies and termination effects.

The set of operations that manipulate the state of an object is expanded to include the operation *restore* which allows an action to restore an object to some previous state. Specifically, an object can be restored to one of three possible states: last committed state, last checked state or previous state. *Committed state* is the state of an object in the system database produced by all committed action. Restoring an object to the last committed state is referred to as *fully restoring* the object. The state of an object in a virtual database produced by a committed action is known as *checked state*. *Previous state* is the most recent state of an object generated by an action different from the invoking action.

Restore, like release, can be invoked on an object by an action any time during the execution of the action. Restoring an object to its previous state is useful in discarding the most recent effects of an action to the object. On the other hand, restoring an object to its committed or checked state is useful in discarding the changes made by one or more in-progress actions since the time the object was last committed.

2.2.2 Termination Effects

At commit, the state of an object produced by an action is made permanent only if the object has already been released in the system database. However, an action may choose to release an object in some virtual database rather than in the system database. Clearly, these changes are not guaranteed to be permanent. The same rule is also applied to newly created objects in a virtual database: unless a newly created object is released in the system database, it is not made permanent. Thus, the ACTA model allows for a larger variety of effects of a committed action on the state of the objects. This is also true for abort.

In the case that an action attempts to commit while it has an abort-dependency on another action in-progress, the action is *conditionally committed.* A conditionally committed action may release its objects to some virtual database with the appropriate propagation conditions but *not* to the system database.

All objects acquired by an action but not yet released are collectively referred to as *the objects of the action.* When an action aborts, all its objects are restored to their last checked state and released to the databases from which they have been acquired. For objects in the system database, the last checked state is the same as the last committed object state and hence restoring an object to its last committed or checked state in the system database has the same effects.

When an action releases an object to some virtual database, it can specify whether the release is *permanent*, in which case the disposition of the action has no effect on the state of the object, or *temporary*, in which case the object is restored when the action aborts. When the last action which has access to a virtual database terminates, every object in the virtual database which has not been acquired by the action and is associated with propagation conditions, is treated according to these conditions. All other objects are treated as if they were objects of the terminating action.

Current Release to Φ	Subsequent Release to Ω	Condition
OA	OA *or* OT	$B \overset{a}{\Rightarrow} A$ *or* $B \overset{a}{\Rightarrow} A$ *until* $release(O_i, \Omega, B)$ *or* $B \overset{a}{\Rightarrow} A$ *until* $release(O_i, \Omega, any)$
OC	OT	$A \overset{c}{\Rightarrow} B$ *or* $A \overset{c}{\Rightarrow} B$ *until* $release(O_i, \Omega, B)$ *or* $A \overset{c}{\Rightarrow} B$ *until* $release(O_i, \Omega, any)$
OT	OT	$B \overset{a}{\Rightarrow} A$ *or* $B \overset{a}{\Rightarrow} A$ *until* $release(O_i, \Omega, B)$ *or* $B \overset{a}{\Rightarrow} A$ *until* $release(O_i, \Omega, any)$ *and* $A \overset{c}{\Rightarrow} B$ *or* $A \overset{c}{\Rightarrow} B$ *until* $release(O_i, \Omega, B)$ *or* $A \overset{c}{\Rightarrow} B$ *until* $release(O_i, \Omega, any)$

Table 1: Subsequent Object Release Modes and their Conditions

2.2.3 The Importance of Conditions

An action may attach conditions to an object while acquiring and releasing objects. Conditions are important because they allow finer control of the interactions among actions over objects thus forcing interacting actions to behave correctly. As an example, consider the three conditions that specify when an object should be released to the origin virtual database from which it was acquired, if the acquiring action terminates:

1. *On-Abort or OA:* The released object is returned to the origin virtual database when the acquiring action aborts.

2. *On-Commit or OC:* The released object is returned to the origin virtual database when the acquiring action commits.

3. *On-Terminate or OT:* The released object is returned to the origin virtual database when the acquiring action terminates, i.e. aborts or commits.

Now, in order to guarantee proper return of the object from subsequent acquisitions, further conditions need to be attached when the object is released. There are a number of alternative conditions. The choice of the condition depends on the semantics of interactions allowed in a given computation.

Consider an object O_i acquired by A from a virtual database Φ to which the object was released under the condition *on-abort*. A can subsequently release O_i to a virtual database Ω under either the condition *on-abort* or *on-terminate* and one of the following provisions with respect to any action B which acquires the object (from Ω):

- B already has or develops an abort-dependency on A ($B \overset{a}{\Rightarrow} A$), or
- B develops a conditional-abort-dependency on A until the object is released to Ω by B ($B \overset{a}{\Rightarrow} A$ *until release*(O_i, Ω, B))[2], or
- B develops a conditional-abort-dependency on A until the object is released to Ω (by any action) ($B \overset{a}{\Rightarrow} A$ *until release*$(O_i, \Omega, \text{any})$).

In all possible releases, the abort-dependency of action B on action A guarantees that if action A aborts, the object will have the proper status (being in Ω) to be returned to Φ. That is, if A aborts, B also aborts and the object is returned to Ω, given that the object was conditionally released to Ω subject to *on-abort* or *on-terminate.*

Now, consider an object O_i acquired by A from a virtual database Φ to which it was released under the condition *on-commit.* A can subsequently release O_i to virtual database Ω under the condition *on-terminate* and the provision that action A already has or develops a commit-dependency on the action B which acquires the object (from Ω) ($A \overset{c}{\Rightarrow} B$), or it develops a conditional-commit-dependency on B until the object is released to Ω either by B or some other action ($A \overset{c}{\Rightarrow} B$ *until release*(O_i, Ω, B) or ($A \overset{c}{\Rightarrow} B$ *until release*(O_i, Ω, any)). The commit-dependency of action A on action B prevents action A from committing before B is in a position to properly return the object to the virtual database Ω.

Finally, the conditions that need to be imposed while releasing an object that was acquired under the condition *on-terminate* is similar to the above two cases. Table 1 summarizes the subsequent object release conditions and the provisions under which these releases are allowed.

[2] *while* $\neg A \equiv$ *until* A.

3 Modeling Different Transaction Schemes

In this section, five major transaction models which appeared in the literature are modeled using the ACTA model. These are traditional transactions, Nested Transactions, Recoverable Communicating Actions, Split-Transactions, and Cooperative Transactions. Although Transaction Groups [Fernandez89] are not included, their characterization is no harder than that of Cooperative Transactions, since they have a similar structure. The communication modes in Transaction Groups can be specified by means of communication obligations in ACTA.

3.1 Traditional Transactions

Since the traditional transaction model was developed for reads and writes, we use reads and writes in the description below. This holds for our description of Nested Transactions and Split-Transactions as well. The characterization of shared read access, as well as of the *basic* two-phase locking behavior, in the ACTA model was presented in the previous section 2.2.1. However, the most common semantics associated with traditional transactions are those defined by the *strict* two-phase locking protocol [Bernstein87] and *commutative* (e.g. shared read) access to objects. As might be expected, the characterization of the strict two-phase behavior is simpler that the basic one.

No completion dependencies or communication obligations are ever developed between transactions, since it is not possible for one transaction to access in a conflicting way any object which is accessed by another transactions, while the transactions are in-progress. During execution time, a transaction does not release any of its objects and the conditions when acquiring an object do not allow conflicting operations to be acquired by more than transaction. All transactions execute only against the system database.

In-progress Effects On Objects

Rule 1: Before a transaction reads an object, it must acquire the object for read and with the condition no-write.

Rule 2: Before a transaction writes an object, it must <u>acquire</u> the object for write and with the condition <u>no-read</u> and <u>no-write</u>.

Termination effects on Objects

Rule 3: If a transaction commits: all its objects are released to the system database.

Rule 4: If a transaction aborts: all its objects are restored to their last committed state and released to the system database.

The last two rules express the strictness property which requires that objects acquired by a transaction, should be released only when the transaction terminates.

3.2 Nested Transactions

In the Nested Transaction model [Moss81], transactions are composed of subtransactions designed to localize failures within a transaction. A subtransaction can be further decomposed into other subtransactions, and thus, the transaction may expand in a hierarchical manner. Subtransactions 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.

Object inheritance, i.e. the ability of one transaction to access the objects of another transaction, is supported between subtransactions and their descendants. The effects on the objects are made permanent only when the top-level transaction commits.

Here is the characterization of Nested Transactions in the ACTA model:

Completion Dependencies

Rule 1: $\forall i,\ Child_i \overset{a_x}{\Rightarrow} Parent$

The abort-dependency of a child on its parent guarantees the abortion of the child subtransaction in the case that its parent aborts. Furthermore, the exclusive-abort-dependency prohibits a child subtransaction from having more than one parent; this ensures the hierarchical structure of the Nested Transactions.

Rule 2: $\forall i$, *Parent* $\overset{c}{\Rightarrow}$ *Child*$_i$

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

In-progress Effects On Objects

For each subtransaction S_{ij} (the *ith* child of the subtransaction $S_{j,h}$), there is a virtual database Φ_{ij} which is shared by that subtransaction and its descendants, i.e. all the subtransactions which have a transitive-abort-dependency on S_{ij}. That is, any subtransaction executes against the system database, the virtual database associated with it (Φ_{ij}) and the virtual databases of its ancestors (for instance from S_{jh}'s virtual database Φ_{jh}).

Rule 3: Before a transaction reads an object, it must acquire the object for read and with the condition no-write.

Rule 4: Before a transaction writes an object, it must acquire the object for write and with the condition no-read and no-write.

Rule 3 and 4 specify that only reads can execute in parallel.

Rule 5: A subtransaction S_{ij} should release all its objects to the virtual database Φ_{ij}. The release should be temporary with the condition that the objects be restored to their last checked state in the virtual database Φ_{ij} on-abort.

The temporary release guarantees that the changes to the object by the releasing action continue to depend on the disposition of the action. The constraint *restore on-abort* makes sure that all objects are properly restored and released to the origin virtual database. This is assured even if the action which subsequently acquires the object, releases the object to some other virtual database and then aborts. Thus, rule 5 ensures the flow of both data and objects down the hierarchy as defined by object inheritance.

Termination Effects On Objects

Rule 6: If the top-level transaction commits*:*

all its objects should be published and released to the system database.

Rule 7: If the top-level transaction aborts*:*

all its objects should be restored to their last committed state and released to the system database.

The two above rules specify that only the top-level transaction can release a modified object to the system database.

Rule 8: If a subtransaction S_{ij} commits*:*

all its objects should be released to the virtual database Φ_{jh} shared by its parent subtransaction S_{jh}, its siblings S_{kj} and their children. The release should be permanent with the propagation condition that each object be restored to its last checked state <u>*on-abort*</u>*.*

Rule 8 guarantees the proper flow of both data and objects up the hierarchy. The constraint *restore on-abort* and the default handling of objects in the case that an action aborts ensures that all the changes so far on the objects by the conditionally committed subtransaction are not lost if the objects are subsequently inherited down the hierarchy. A couple of other things should be noted with respect to this rule: first, a subtransaction does not actually commit when it invokes the commit operation, but it *conditionally commits* and it can still be aborted; this is due to its abort-dependency on its parent subtransaction (rule 1). Recall that the conditional commit allows for objects to be released to a virtual database but not to the system database. Secondly, when a subtransaction S_{ij} conditionally commits, it is the last subtransaction to terminate and has access to the virtual database Φ_{ij} shared with its descendants. Since the subtransaction never permanently releases any object when it is active, all the objects in this virtual database are accessible to the subtransaction. Furthermore, none of the objects have any constraint with respect to commit. Thus, the subtransaction is responsible for appropriately releasing all the objects, and it releases them to the virtual database shared by its parent and siblings and their children according to rule 8.

Rule 9: If a subtransaction aborts*:*

(i.) all its objects previously acquired from virtual databases accessible to its ancestors are restored and released to the appropriate virtual databases.

(ii.) the rest of its objects are restored to their last committed state and released to the system database.

This last rule restates the handling of objects in a virtual database when the last action that has access to the virtual database, aborts. Recall that all objects released to this virtual database according to rule 5 are constrainted to be restored to their last checked state and released to their origin virtual database. The rest of the objects are not associated with any propagation condition since they are acquired from the system database and then released in the virtual database either by the aborting subtransaction or by one of its children which has already conditionally committed.

3.3 Split-Transactions

In the Split-Transaction model [Pu88], it is possible for a transaction to split into two transactions, the *parent* and *child* transactions where the parent transaction is the origin transaction. Parent and child transactions may be *independent* in which case they can commit or abort independently, or they may be *serial* in which case the parent must commit in order for the child to commit. Whether the parent and child transactions are independent or serial depends on the objects accessible to them. Here is the characterization of Split-Transactions in the ACTA model:

3.3.1 Independent Transactions

Completion Dependencies

Rule 1: No action-dependencies between parent and child transactions.

Independent transactions behave like traditional transactions and as such their interactions do not lead to the development of any kind of action-dependencies.

In-progress Effects On Objects

The child transaction is associated with a virtual database.

Rule 2: Before a transaction reads an object, it must acquire the object for read and with the condition no-write.

Rule 3: Before a transaction writes an object, it must acquire the object for write and with the condition no-read and no-write.

The commutative type access to objects guarantees that the parent and child transactions operate on disjoint sets of objects after the split. Otherwise their independence would be destroyed.

Rule 4: The parent transaction should permanently release the objects intended for the child to the child's virtual database.

The permanent release leaves to the child the responsibility to make all the changes to the released object up to the split permanent to the system database. This is also restated by the following two rules.

Termination Effects On Objects

Rule 5: If either transaction aborts*:*

all its objects are restored to their last committed state and released to the system database.

Rule 6: If either transaction commits*:*

all its objects are released to the system database.

3.3.2 Serial Transactions

Completion Dependencies

Rule 1: $\mathit{Child} \overset{a_x}{\Rightarrow} \mathit{Parent}$:

The abort-dependency guarantees that the child transaction aborts if the parent aborts and that the child commitment is delayed until its parent commits. The exclusive-abort-dependency prevents a child from joining (see below) a third transaction[3]. Note that this does not prevent child and parent transactions from joining.

In-progress Effects On Objects

Rule 2: The parent transaction should temporarily release all objects in the sets `ParentWriteSet ∩ ChildWriteSet (= ChildWriteLast)` *and* `ChildReadSet ∩ ParentWriteSet (= ShareSet)` *to the virtual database accessible to both parent and child.*

Rule 3: The parent transaction should permanently release all objects in the set `ChildWriteSet - ChildWriteLast` *to the child's virtual database.*

All the changes to the objects up to the release time become visible to the child transaction. The temporary release ensures that the changes to the objects which are accessed by both transactions, are not lost if the child aborts[4]. On the other hand, permanent release (in rule 3) leaves to the child the decision to make the changes on the rest of the objects permanent to the system database.

Termination Effects On Objects

Rule 4: if the parent transaction aborts*:*

all its objects are fully restored and released to the system database.

Rule 5: If the parent transaction commits*:*

all its objects are released to the system database.

[3]This constraint can be removed if the *join* operation requires that the joint transaction develops the same dependencies as the joining transaction.

[4]Furthermore, these two conditions allow the parent transaction to regain access to these objects after the abortion of the child. Note that this is not supported by the original notion of Split-Transactions [Pu88], although it might be appropriate for some applications.

Rule 6: If the child transaction aborts*:*

(i) all the objects acquired from the virtual database are restored to their last checked state and released to the virtual database.

(ii.) all the other objects are fully restored and released to the system database.

Rule 7: If the child transaction commits*:*

all its objects are released to the system database.

If the parent aborts then the child transaction is also aborted, given that the child has an abort-dependency on the parent. Thus, by rules 4 and 5, all the objects acquired by both transactions are fully restored and released to the system. When the parent commits, all its changes are made permanent including those on the object which were temporary released to the child. Rule 5 also guarantees that changes to the objects up to the split survive even when the child aborts since it calls for these objects to be restored to their last checked state discarding only the child's changes. Rule 4 becomes applicable after the parent transaction commits, given the abort-dependency of the child on the parent.

3.3.3 Joint Transactions

In the Split-Transactions model, it is also possible for two transaction to join into one, the *joint* transaction. The joint transaction is either of the origin ones. When the transactions join, they release their objects to the joint transaction.

The characterization of Joint-Transactions in the ACTA model is straight forward:

Completion Dependencies

Rule 1: Joining transaction $\overset{a}{\Rightarrow}$ *Joint transaction*

In-progress Effects On Objects

Rule 1: The joining transaction should permanently release all its objects to the virtual database accessible to the joint transaction.

Termination Effects On Objects

Rule 1: If the joint transaction aborts*:*

all its objects are fully restored and released to the system database.

Rule 2: If the joint transaction commits*:*

all its objects are released to the system database.

3.4 Recoverable Communicating Actions

In the Recoverable Communicating Actions (RCA) model [Vinter86], an action, the *sender*, is allowed to communicate with another action, the *receiver*, by exchanging objects resulting in an *action-dependency* of the receiver on the sender. Action-dependency in the RCA model has similar semantics to the abort-dependency in the ACTA model. The main difference is that in the RCA model, action-dependency requires synchronized commitment of the sender and the receiver even in the case that the sender commits first and has no dependencies on the receiver; in the ACTA model, abort-dependency does not. However, partial failures are tolerated since an action may abort without aborting the action on which it has developed an action-dependency.

Because of the similarities between RCA and ACTA models, the characterization of RCA in the ACTA model is quite simple:

Completion Dependencies

Rule 1: $Sender \overset{c}{\Rightarrow} Receiver$

Rule 2: $Receiver \overset{a}{\Rightarrow} Sender$

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

In-progress Effects On Objects Each receiver is associated with a virtual database which is accessible to both the sender and the receiver.

Rule 3: The sender should release the communicated objects to receiver's virtual database. The release should be temporary with the condition the objects to be restored in the virtual database on-abort.

Termination Effects On Objects

Rule 4: if an action aborts*:*

(i.) all its objects acquired from the system are fully restored and released to the system database.

(ii.) all its objects acquired from virtual databases are restored and released back to the origin virtual database.

Rule 5: if an action commits*: all its objects are released to the system database.*

3.5 Cooperative Transactions

In the Cooperative Transaction model [Bachilon85], transactions are multi-level, 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; *cooperative* 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 update 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 [Moss81]. In cooperative transactions object flow is only supported between adjacent levels through intermediate *semi-public* databases. Thus a semi-public database is similar to the virtual database in the ACTA model.

The characterization of Cooperative Transactions in the ACTA model is very close to the one in Nested Transactions due to their similarities in their structures.

Completion Dependencies

Rule 1: $\forall i,\ Cooperative_i \overset{as}{\Rightarrow} Project$

Rule 2: $\forall i,\ Project \overset{c}{\Rightarrow} Cooperative_i$

Rule 3: $\forall j,\ Subcontractor_j \overset{as}{\Rightarrow} (client) Cooperative_i$

Rule 4: $\forall j,\ (client)\ Cooperative_i \overset{c}{\Rightarrow} Subcontractor_j$

In-progress Effects On Objects

Rule 1: The project transaction should release all its objects to the virtual database Ω which is accessible to its cooperative transactions. The release should be temporary subject to the condition that the objects be restored to the virtual database on-abort.

Rule 2: A cooperative transaction may release any of its objects to the virtual database Φ_i which is accessible by its subcontractors, or to the virtual database Ω which is shared by the project and its cooperative transactions. However, the release should be temporary subject to the condition that the objects be restored to the virtual database on-abort.

Termination Effects On Objects

Rule 1: If the project transaction commits*:*

all its objects are published and released to the system database.

Rule 2: If the project transaction aborts*:*

all its objects are fully restored and released to the system database.

Rule 3: If a cooperative transaction commits*:*

all its objects are released to the virtual database Ω shared by the project and its cooperative transactions. The release should be permanent with the condition restored to the last checked state on-abort.

Rule 4: If a cooperative transaction aborts*:*

(i.) all its objects acquired from the virtual database Ω which is accessible to the project and its cooperative transactions, are restored and released to the virtual database Ω.

(ii.) the rest of its objects are fully restored and released to the system database.

Rule 5: If a subcontractor commits*:*

all its objects should be released to the virtual database Φ_i shared by its client transaction and its siblings subcontractors. The release should be permanent and with the condition restore to the last checked state on-abort.

Rule 6: If a subcontractor aborts*:*

(i.) all its objects acquired from virtual databases Φ_i accessible to its client transaction are restored and released to the Φ_i.

(ii.) the rest of its objects are fully restored and released to the system database.

4 Conclusion

ACTA, the comprehensive transaction model proposed in this paper, captures the spectrum of interactions among transactions defined by the competitive and cooperative environments. Each point in the space of interactions is characterized along four dimensions: *action dependencies, communication obligations, in-progress effects* and *termination effects.* Its ability to model all major previously proposed transaction-based schemes is indicative of its generality.

A formal model is currently under investigation. Such a model will allow us to characterize the correctness properties of a given model. For example, to determine if it produces only serializable computations, and if not, whether the computation is *acceptable*, i.e. the interactions in the computation do not conflict in such a manner as to produce object inconsistencies. Thus, with a formal model, points within the spectrum of interaction allowed by our model can be examined in order to understand the formal properties of the modeled systems.

Future steps include the development of concurrency control and recovery protocols appropriate for the ACTA model and a mechanism or a set of mechanisms for implementing these protocols. Intuitively it seems that a single mechanism will be complex and inefficient. However, a set of mechanisms each of which implements efficiently some subset of these protocols while supporting all of them, is possible. Obviously, the predominant interactions in the application will dictate the appropriate mechanism. Such an approach is referred to as the *toolbox* approach and allows the building of extensible systems.

In this way, when a facility is needed to support traditional or new applications, such as CAD/CAM, distributed AI and Operating Systems, the appropriate mechanism can be selected without confining the system into a specific set of interactions as is currently the case.

5 References

[Bachilon85] Bachilon F., Kim W., and Korth H., 'A Model of CAD Transactions,' *Proceedings of the 11th VLDB Conference,* Stockholm, 1985.

[Bernstein87] Bernstein, P., Hadzilakos, V., and Goodman. N., 'Concurrency Control and Recovery in Database Systems,' *Addison-Wesley Edition*, 1987.

[Eswaran76] Eswaran, K., Gray, J., Lorie, E., Traiger, I., 'The Notions of Consistency and Predicate Locks in a Database System,' *Communications of the ACM*, vol. 19, no. 11, November, 1976.

[Gray78] Gray, J., 'Notes on Data Base Operating Systems,' *IBM Research Report: RJ2188*, IBM Research, California, February 1978.

[Gray81] Gray, J., 'The Transaction Concept: Virtues and Limitations,' *Proceedings of the 7th VLDB Conference*, September 1981.

[Fernandez89] Fernandez, M., and Zdonik, S., 'Transaction Groups: A Model for Controlling Cooperative Transactions,' *Workshop on Persistent Object Systems: Their Design, Implementation and Use*, Newcastle, Australia, 1989.

[Moss81] Moss, J. E. B., 'Nested Transactions: An Approach to Reliable Distributed Computing,' Ph.D. Thesis, MIT/LCS/TR-260, 1981.

[Pu88] Pu, C., Kaiser, G., and Hutchinson N., Split-Transactions for Open-Ended Activities," *Proceedings of the 14th VLDB Conference*, Los Angeles, 1988.

[Vinter86] Vinter, S., Ramamritham, K., Stemple, D., 'Recoverable Actions in Gutenberg,' *Proceedings of Distributed Computing Systems*, May 1986.