# SPECIFYING FLEXIBLE CONCURRENCY CONTROL SCHEMES: AN ABSTRACT OPERATIONAL APPROACH

D. STEMPLE
*Computer and Information Science, University of Massachusetts,*
*Amherst, Massachusetts, MA 01003, U.S.A.*

and

R. MORRISON
*Department of Mathematical and Computational Sciences, University of St Andrews,*
*North Haugh, St Andrews, Fife, KY16 9SS,Scotland*

ABSTRACT

An abstract, operational model for specifying flexible concurrency control schemes within a persistent store is presented. The goal of the model is to allow concurrency control schemes to be specified in a manner that promotes understandability and supports their implementation. Schemes that support controlled sharing among autonomous computations are the primary targets of this work. An abstract machine programmed by a set of rules is employed to specify operational semantics for the concurrency control schemes. Examples of specifications are given.

## 1 Introduction

Coordinating a set of computations that share data is a complex undertaking. Mechanisms for such co-ordination have been designed for operating systems, programming languages and database systems. These include semaphores[6], monitors[8], mutual exclusion[5], path expressions[1], locks[7], and optimistic concurrency control systems[9]. It has been difficult to understand the power and behaviour of these mechanisms and to compare them with each other. This difficulty stems from both the low level nature of the mechanisms and the inherent complexity of the problem. This paper addresses the task of building systems for controlling the co-ordination of computations on shared data in a persistent programming environment and attempts to alleviate the difficulties of understanding and comparing different control schemes.

The goal of this approach is to allow programs to control the coherence of sequences of their operations on shared data, called actions, in an understandable and flexible manner. The actions are programmed by specifying algorithms for manipulating the data and by annotating the algorithms with markers that specify how the data sharing is to be controlled. An extreme example of the coherence of a sequence of operations is the atomic transaction in which the operations are isolated from the effects of any other concurrent transaction and only allowed to have an effect on the database as a single unit. In systems supporting atomic transactions this strong coherence can be programmed simply by inserting begin and end transaction markers around a set of operations to delineate an atomic transaction. The effect of begin and end has to be specified separately at a lower level in the system. Examples of more complex coherence occur in design systems where parts of a design are updated locally by multiple designers and the different changes are reconciled before their joint effect is made to the global design.

Many ways of achieving the isolation of atomic transactions have been devised and implemented and many mechanisms for co-ordinating co-operative computations have been proposed. This paper does not add another proposal to these efforts. Rather it presents an approach to specifying systems of concurrency control. The approach is based on the concepts of the ACTA framework[2, 3, 4] but is less general and more implementation oriented. It is flexible in that it allows more than one concurrency model to be specified, abstract in that it allows more than one implementation to realise a specification and operational in that specifications are written in terms of behaviour rather than invariants. We call the model embodying our approach the Communicating Actions Control System or CACS.

This paper presents a method of specifying flexible concurrency control schemes using the concepts of actions, objects, events, visibility control and dependencies. The approach is to define a control environment in which protocols of data sharing can be defined operationally but are sufficiently abstract to allow significant freedom in implementing the protocols. In the next Section the conceptual model of data sharing that is at the core of CACS is elaborated. Following this the details of CACS processing are presented. Finally two examples of specifications of concurrency control protocols are given: for atomic transactions and a co-operative sharing system.

## 2 The Conceptual Framework

The conceptual framework of CACS may be explained in terms of the components used to specify concurrency control schemes. These are actions, objects, events, visibility control and dependencies. Each is described in turn.

### *2.1 Actions*

The concept of an action is the starting point of CACS. An action is a sequence of operations on shared data that has some sense of cohesion; it is a unit of computation that needs some isolation from concurrent users of shared data. The exact nature of the unity and isolation of an action depends partly on the particular concurrency control scheme being used by the action and partly on the programming of the action itself. For example, in the atomic transaction model the isolation is complete and is not affected by the programming of the transaction beyond the placement of the transaction begin and end commands. In nested transactions[11], on the other hand, the parent and children transactions are actions that can affect each other in reasonably complex ways, some of which can be programmed into the actions themselves, programmed aborts for example.

Actions perform their computations by executing programs which are algorithms annotated by markers. The algorithms specify the manipulations of the data whereas the markers, called events in CACS, specify the points at which the actions must interact with the control system to operate correctly over the shared data.

### *2.2 Objects*

The data consists of uniquely identified objects. Objects are associated with types which define sets of operations. Actions perform operations on data objects. CACS works with an abstract view of the operations, a view that is limited to what operations are updates and what operations conflict, the latter being a concept defined in a particular control scheme. Operations may be simply the reading and writing of data or may be the operations associated with the types in the programming language used to program the actions.

### *2.3 Events*

In CACS there are two categories of events: action events and object/visibility events. All events are requests by an action to the control system to proceed and any particular concurrency control scheme will include all the events necessary to control the use of shared data.

Action events include action initiation and termination. Other events, such as spawning subtransactions in the nested transaction model are action events in particular concurrency control schemes. As will be seen later action events and their semantics are defined as part of concurrency control schemes. The semantics of these events is defined by rules that specify the effects of actions on the visibility and dependencies. For example, in the atomic transaction model the particular termination event called commit is defined as the commit to the database of all objects the action has changed. In the nested transaction model, the commit of a child transaction is defined to make the updated objects visible to its parent transaction.

Object/visibility events include object operations and object commits. Unlike in the atomic transaction model, the concept of commit in CACS is attached to objects: object commit is the operation that makes the effect of an action’s operations on an object globally visible. Object commit may or may not be explicitly invokable in an action’s programming language. For example, in the atomic transaction model object commits are not explicitly programmed and CACS would model the commit of a transaction by the commit of all the objects updated by the transaction. But in a control system that allows sharing, committing an object might be an explicitly invoked operation of an action.

### *2.4 Visibility*

The model of computation that forms the CACS abstraction is designed to focus on the visibility of data from different actions. The semantics of CACS control over visibility is expressed in terms of the database, which comprises the globally visible data, and conceptual stores called access sets. Each action is associated with a local access set and may use other shared access sets in order to effect communication between actions without using the database. When using CACS to specify a concurrency control scheme and when explaining the semantics of a particular control scheme the access sets may be thought of simply as stores holding data. When actions operate on shared data this is modelled in CACS by the effects of the operations being kept in shared access sets. Movement of data from access sets to the database, which is the semantics of object commit, is the way changes to visibility are made global in CACS. Movement among local and shared access sets occurs implicitly as a part of operation semantics and explicitly by association of operations with shared access sets.

The visibility controlled by CACS is not to be confused with that controlled by protection schemes in the programming language or operating system. In those schemes the focus is on who (programs or processes) can access what and how (what operations). CACS control is concerned with the sequencing of when access is allowed by actions and when the effect of an action is visible to other actions. From the point of view of specifying the concurrency control scheme the protection scheme is hidden and anything presented to CACS by an action can be assumed to have passed through the protection scheme's filters. The multiple store model of CACS is different from the one store model of some persistent systems but necessary to allow the flexibility in concurrency schemes.

## *2.5 Dependencies*

The final focus of CACS is on dependencies between actions. A dependency is a relationship between two actions that expresses the requirement of the dependent action to either perform or delay some operation or operations based on the behaviour of the action being depended upon. A dependency between two actions can be associated with an object and with a shared access set. An example of a dependency that one atomic transaction can develop on a second occurs when the first is allowed to read data that the second has written but not committed. In this case the first transaction depends on the second in that it must delay its commit until the second commits. In more permissive control systems the dependencies might be more complex and some could be under actions' control. The concept of a dependency is a part of CACS but no particular dependencies are built in; CACS allows dependencies to be defined and their behavioural semantics to be given as part of particular control scheme specifications.

## *2.6 Summary*

The model of computation used in CACS is that of autonomous agents, called actions, making requests to access data in a globally shared database. The database consists of uniquely identified entities called objects. Objects are associated with types which define sets of operations, the details of which are not known to the control system. Access to objects in the database is made by actions requesting that operations be performed on database objects. When a request is granted by the control system, the effect of the operation appears in the requesting action's local access set but not in the database. Objects' identities remain the same in access sets and the database. Thus an object may have different states in different scopes of visibility. In order for the effect of an operation on an object to be made globally available the object must be committed to the database. Conceptually this is a separate operation which makes the database state of the object the same as its state in the local access set of the action performing the commit. Effects of operations may be shared among actions without their objects being committed to the database. This is accomplished by committing objects into shared access sets which are separate from the local access sets.

Actions interfere and cooperate with each other through the mediation of the control system via dependencies. Dependencies are formed as a by-product of actions making requests, and cause the control system to delay, disallow, require, or invoke by proxy operations by actions involved in the dependencies. The behaviour caused or prohibited by dependencies is defined as a part of a particular concurrency control scheme specified in CACS terms. The concept of a correct interleaving of actions is specified in CACS in terms of the behaviour induced by dependencies.

The interaction between actions and a control system specified in CACS is only partially modelled. The control system's reception of requests from actions is modelled as the control system taking events from an event stream. The way in which events are placed on the event stream is not specified. It is assumed that the activities of the control system are made known to the actions also through unspecified means. The creation of identifiers for actions, objects and shared access sets is independent of the control system.

## 3 Overview of CACS Mechanisms

A particular concurrency control scheme is defined by giving a set of rules for the behaviour of the CACS abstract machine. This yields the operational, abstract semantics of a particular scheme. The rules specify the behaviour of the control system when its takes an event from the event stream.

The abstract state of CACS has the following components:

- the event stream including its current position
- the rule base, which can change as a result of executing rules
- the action histories containing traces of actions in progress
- the visibility structure containing the database and the local and shared access sets along with their relationship to actions
- the dependency structure recording all current dependencies among actions

The description of the CACS mechanisms starts with the dynamic input to a CACS session, the event stream. This is followed by the details of the rules and finally a description of the state maintained by the system is presented.

### *3.1 The event stream*

The main input to a CACS session is a stream of events. The events are all associated with actions, the identifiers of which are attached to each event. Normally events are generated by actions. CACS can, however, also insert events into the stream.

The effect of CACS processing on the producers of events is not directly modelled in CACS itself, except as it is related to actions' visibility of and effect on objects in the database and access sets. However, it is assumed that CACS effects are seen by actions and produce changes to the event stream as appropriate. Two cases are important.

Firstly, in the case of a CACS produced abort of an action it is left to the action to decide whether or not to re-run itself. If the action is to be re-run it will be seen by CACS as a new action unassociated with its previous incarnation.

Secondly, and more importantly, CACS can insert an action initiation event into the stream in order to start the flow of events of a previously identified action, one that could have been requested to be started on some contingent event, or one that CACS has delayed for some reason. Events in such "CACS initiated actions" are defined externally to CACS even if events from the actions have been seen previously by CACS. If the events have already been seen they might have affected the database but any relationship between them and the new stream of events of the initialised action is outside the control of CACS and no assumptions about the new events are made.

### *3.1.1 Events*

Events are of nine kinds in two categories. Three are called action events and six are called object/visibility events. The action events are

- action initiation events
- action termination events
- other events for particular concurrency schemes

The object/visibility events are

- operations on objects
- object commits
- object removals from access sets
- creations of a shared access sets
- attachments/detachments of actions to/from shared access sets
- destructions of shared access sets

The event stream will have only one initiation event for an action and at least one termination event after the initiation event. All other action and object/visibility events of an action will be between its initiation event and the first termination event. The semantics for events are provided by the rules for a specific concurrency control scheme.

### *3.1.2 Dynamics*

The evaluation loop of the CACS abstract machine takes the next event from the event stream and executes the bodies of all rules whose conditions are satisfied by the event and the current state. The process of finding the rules satisfying an event and executing the rules’ bodies is called realising the event.

Rule bodies contain instructions specifying manipulations of the state components.

## *3.2 Rules*

A rule consists of an event pattern, a guard predicate and a body. The body is executed whenever the current event matches the event pattern and the guard predicate is true with respect to the current state. Patterns consist of constants and variables. When a pattern is matched, the parts of the matching event that correspond to the variables are bound to the variables throughout the scope of the rule’s guard and body. Rule bodies can cause

- new rules to be created
- rules to be modified or destroyed
- the current event to be deleted and other events to be inserted immediately after the current position
- the creation and destruction of action histories and insertion of events into them
- the visibility structure to be modified
- the dependency structure to be modified

The bodies of rules whose patterns and guards are satisfied are executed in their order on the rules list. A single scan is made for each event. Changes made to the rule list during realisation of an event are all installed at the end of the realisation; rules to be removed are removed first, then changes to the rules are made, followed by the addition of new rules to the beginning of the list in the order they were created. Events to be inserted into the event stream are gathered until the end of a realisation and all inserted at once, immediately after the current event, in the order they were created in the realisation. Manipulations of the visibility structure and the dependency table are made at the time of the rule body execution. Rule execution continues until either the end of the rules is reached or until the current event is removed by executing the CACS abstract machine instruction **deleteThisEvent**.

## *3.3 The in-progress action histories*

An action is considered to be in-progress during the time that CACS is realising events between an initiation event and a termination event for an action. While an action is in-progress all of its events are kept in their occurrence order in a structure called an action history. Each event in the event stream has an owning action whose identifier is part of the event structure. This identifier links an event to its action’s history. Events are entered into their histories under the following circumstances:

- immediately after the realisation of an event unless the realisation terminated with **deleteThisEvent**.
- immediately upon execution by CACS of an object commit instruction.

Action histories can be accessed for various purposes, for example, in the case of atomic transactions to discover conflicts between actions and to determine the objects that must be committed at transaction commit time.

The separation of action histories from the event stream is done for the convenience of interrogating the histories during the realisation of rules.

### *3.4 Visibility structure*

Much of the semantics of events is defined by rules and thus in terms of the effects of executing rule bodies. However, object/visibility events have some predefined effects on the visibility structure. Operations and commits move objects between the database and access sets. The following define the basic semantics of operations, commits and removes.

- An operation on an object by an action affects the object in the action's local access set. This may entail copying the object from either the database or a shared access set into the action's local access set.
- Object commit moves an object in its current state from the invoking action's access set into either a shared access set or the database.
- Remove deletes an object from a local or shared access set.

Though these semantics of object events are unchangeable, hard-wired in a sense, they can be augmented by rules triggered by an object event becoming the current event.

### *3.5 Dependency structure*

The dependency structure contains the details of inter-action dependencies described in Section 2.5. It is a five dimensional structure. The five dimensions are parent action (the action being depended upon), the child action (the dependant action), the kind of dependency, the object and the shared access set. There are two instructions for updating the structure, one to set a dependency and one to remove a dependency. Access to this structure is via a set of predefined predicates and functions that allow rule bodies to determine what dependencies exist.

## 4 Examples of Specifications

In this Section an example of how concurrency control schemes may be specified is outlined. Particular schemes will follow in the next Section. First it is necessary to introduce concrete syntax for events, rules, the instructions of the abstract machine and the identifiers used to denote the state structures of the abstract machine. Space prohibits doing this completely and an overview is given instead.

### *4.1 Events*

Events are specified as tuples of the following form:

[eventType, eventName, actionId, other optional parameters]

The *eventType* is based on the ACTA event types:

- $\mathbb{IE}$ for initiation events

- TE for terminating events
- OE for other action events
- OP for operations
- OC for object commit events

The *eventTypee* is used during searches of action histories.

The *eventName* is a name given to the event in a particular scheme. For example, commit is the name ordinarily given to the normal termination event of atomic transactions in serialisable concurrency schemes.

The *actionId* is the unique action identifier defined externally to CACS. The optional parameters may include identifiers for objects, shared access sets and other actions.

The following are valid events in an atomic transaction scheme with read and write operations as the only object/visibility events:

[IE, begin, t1]

[OP, read, t1, o1]

[IE, begin, t2]

[OP, read, t2, o2]

[OP, write, t2, o2]

[TE, commit, t2]

[OP, read, t1, o2]

[OP, write, t1, o1]

[TE, commit, t1]

The identifiers used in the tuples are written in outline font to signify that they are constants in either CACS or a particular concurrency control scheme. The action identifiers, such as *t1*, and the object identifiers, such as *o1*, will be written as plain text in the examples and in italics in the running text.

## *4.2 Rules*

A rule consists of an event pattern, an optional guard predicate and a body. The pattern is bracketed by [ and ] in the same manner as events. If a guard predicate is present it is preceded by a colon. The pattern and optional predicate are separated from the body by $\rightarrow$. Whenever an event occurs, a rule body will be executed if the event matches the rule pattern and both the event and state satisfy the guard predicate. For example, a rule for a begin event with no guard in an atomic transaction scheme will start with:

[IE, begin, vt ] $\rightarrow$

This rule will be matched by any begin event and cause the variable *vt* to be bound to the transaction identifier in the event. A variable such as *vt* matches whatever is in its position in the event to be matched. Notice that the convention for this is to write *vt* in plain text within the rules and in italics in the running text. A pattern variable always starts with the letter *v*.

### 4.3 Abstract machine instructions

The abstract machine of CACS controls the co-ordination of sets of actions. The instructions used to perform this control constitute the abstract machine instructions.

The abstract machine instructions are used within rules to carry out the manipulations of the machine's state for that rule. Examples and descriptions of the abstract machine instructions will be given in the next Section where they arise. They include the ability to carry out the operations specified in Section 3.2.

The abstract machine instructions will be written in bold in the examples. For example, the instruction to create a local access set for the transaction *t* would be written

**createLocalAccessSet** (t)

## 5 Specifications of concurrency control schemes

In this Section specifications for two concurrency control schemes will be presented. The first is atomic transactions.

### 5.1 Atomic transactions

The action events in atomic transactions are begin, abort and commit. These are denoted by

[IE, begin, t]

[TE, abort, t]

[TE, commit, t]

since begin is the initialisation event and abort and commit are the termination events.

The object/visibility events are read and write. These are denoted by

[OP, read, t, o]

[OP, write, t, o]

since read and write are operations of a transaction on an object.

The visibility structure is affected by these events. read copies an object from the database to the local access set of the transaction if it is not already there. From the CACS point of view write updates an object in the local access set and declares the intention to commit it to the database. All other operations on objects are insignificant to CACS here.

In an atomic transaction model, the execution of operations, one of which is a write, on the same object by two atomic transactions creates dependencies: the second transaction to perform the operation on the shared object cannot commit until the first commits and in straightforward approaches if the first aborts, the second must abort.

Whenever two executing transactions operate on the same object and one writes the transactions must be sequenced at least in effect. The control scheme described here accomplishes this by recording a delay dependency from one such action on the other and delaying the dependent action's subsequent operations. This dependency will cause an abort to avoid deadlock in the case where a subsequent operation of the undelayed action forms a delay dependency on the delayed action. Otherwise the delay dependency merely delays the events on the delayed transaction until the undelayed transaction terminates.

Thus let *t1* and *t2* be two atomic transactions that operate on the same object and one operation is a write. *t1* operates on the object first and when *t2* wishes to operate on the object a delay dependency of *t2* on *t1* is created. This causes all further operations of *t2* to be delayed until *t1* terminates by abort or commit. *t1* may however subsequently try to access another object that *t2* has already operated on. Where one of these operations is a write then deadlock would occur if a delay dependency of *t1* on *t2* were set up. Instead this system merely aborts *t2*, the already delayed transaction, in such a circumstance.

### *5.1.1 Atomic transaction rules*

In order to specify atomic transactions rules must be written for each event. The first is the begin event.

```
[IE, begin, vt]→
begin
      createActionHistory (vt)
      createLocalAccessSet (vt)
end
```

This rule will be matched by any begin event and cause the variable *vt* to be bound to the transaction identifier. The body of the rule when executed creates an action history and the local access set for this transaction.

The following rule defines commit in this atomic transaction model.

```
[TE, commit, vt]→
begin
      for each vObject in localAccessSet (vt)
           where exists vEvent in actionHistory (vt)
           matching [OP, write, vt, vObject]
                do commitObject (vt, vObject, DB)  ! DB is the database
      deleteActionHistory (vt)
      deleteLocalAccessSet (vt)
      removeActionRules (vt)
end
```

That is, commit every object in the local access set that has been written by the committing transaction. A **commitObject** causes the object in the access set of the committing action to be copied to either a shared access set or as in this case the database atomically. The action history and the local access set of the matching transaction are then deleted. **removeActionRules** (vt) removes all rules in the rule list specifically tied to the action identified by *vt*.

The following defines the abort event.

```
[TE, abort, vt]→
begin
        deleteActionHistory (vt)
        deleteLocalAccessSet (vt)
        removeActionRules (vt)
end
```

The rules for read and write can be combined in one in the atomic transaction model. Whenever an operation event is encountered all active transactions are checked for conflicting operations in their histories. Conflict occurs if either the operation being checked or the operation in an active transaction is a write of the object being operated on. When a conflict is first encountered by the control system, it is modelled in the dependency structure and causes the conflicting operation and all subsequent events of its transaction except abort to be delayed until the termination of the first action, the first one to get its operation in its history. The completed operations in the delayed transaction are left in its history as there is no conflict with them. If a later operation of the undelayed transaction conflicts with one of the operations left in the history of the delayed transaction, the complete delayed transaction is caused to restart, via the insertion of a begin event, when the undelayed transaction terminates. Delaying the restart involves all of the abort activities since the events of the later restarted transaction may be changed by virtue of new values read from the undelayed transactions results. This is specified by the following rule:

```
[OP, vt1op, vt1, vobj]→
for each actionHistory in actionHistoryList where actionId (actionHistory) ≠ vt1
do
    for each event in actionHistory
        matching [OP, vt2op, vt2, vobj] : (vt1op = write) or (vt2op = write)
do
        if depends (delayDependency, vt2, vt1) then
        begin           ! abort vt2 and restart when vt1 terminates
            abortAndDelayRestart (vt2, vt1)
            removeDependency (delayDependency, vt2, vt1)
        end else
        begin           ! delay vt1 until vt2 terminates
            setDependency (delayDependency, vt1, vt2)
            delaySubsequent (vt1, vt2, thisEvent)
        end
```

In the **matching** clause the predicate following the colon acts in the same manner as a rule guard. The predefined identifier *thisEvent* refers to the current event.

This rule uses a predefined predicate called *depends*. The *depends* predicate returns **true** if a pair of actions have the specified dependency. The dependencies are built during rule execution by **setDependency** commands. The particular dependencies can have various names in different concurrency control schemes. Here the name *delayDependency* is used since in this modelling of atomic transactions there is only one kind of dependency. Dependencies can be associated with objects and shared access sets in order to define a fine granularity of behaviour.

There are two cases in the above specification. The first case is where a *delayDependency* already exists and causes an abort to the already dependent transaction and also causes it to restart when the undelayed transaction terminates. This keeps a cyclic delay dependency for forming and is done by inserting an abort event from the dependent transaction and inserting a rule to issue a begin event for it when the undelayed transaction terminates. It is specified as follows.

```
abortAndDelayRestart (a1, a2)
begin           ! abort a1 and restart it when a2 terminates
    insertEvent ([TE, abort, a1])
    insertRule ([TE, vn, a2])→ insertEvent ([IE, begin, a1])
end
```

The second case is where a dependency does not already exist for conflicting actions. In this case the action whose operation is being realised, *vt1*, is suspended until the dependent action, *vt2*, terminates. The *delaySubsequent (vt1, vt2, thisEvent)* performs this by inserting a rule that will be activated by a termination event for *vt2*. The rule when activated inserts an event into the event stream for redoing the delayed event. The delayed event must also be deleted from the current action history. Finally any subsequent events in *vt1* must also be delayed. This is performed by inserting another rule for this purpose. The specification of *delaySubsequent* is

```
delaySubsequent (a1, a2, event)
begin                ! delay a1 until a2 terminates
        ! delay this event
        insertRule ([TE, vn, a2]→ insertEvent (event))
        deleteThisEvent
        ! delay subsequent events
        insertRule ([vE, vn, a1, vobj]→
        begin
                insertRule ([TE, vn, a2]→ insertEvent ([vE, vn, a1, vobj]))
                deleteThisEvent
        end)
end
```

## *5.2 Actions with shared commits*

A simple model of sharing data is now used to demonstrate the utility of CACS for specifying the control of shared access. In this model actions declare objects to be shared in shared access sets. Declaring an object to be shared registers the action as a sharer of the object in the shared access set and also causes the object to be copied into the local access set if it is not already there. This is done by searching first the shared access set and if unsuccessful the database. If the object was not originally in the shared access set it now is copied there from the local access set. Changes will only be made to the object in the local access set.

An approval protocol controls the committment of shared objects to the database. To commit a shared object an action first commits the object to the shared access set. This causes each sharer of the object to be notified that a commit has been requested. The commit may be approved or disapproved by the sharing actions but while they are doing that further no commits of the object to the shared access set are allowed. The approval/disapproval phase ends when each action has either approved, disapproved or terminated.

Disapproval occurs where one action disapproves the commit but does not cause any rollback of the object in the shared access set. An approval occurs when all sharers have approved the commit or aborted. The approval of all sharers does not cause the commit to the database, it only enables it and any sharer can now commit the object to the database.

The action events in the sharing model are the same as those in the atomic transaction model described in Section 5.1. The object/visibility events are share, commitShare, dropShare, dropLocal, approve, reject and commitToDB. These are denoted by the following in which *a* stands for an action identifier, *o* an object identifier and *sas* a shared access set identifier:

[OP, share, a, o, sas]

[OC, commitShare, a, o, sas]

[OE, dropShare, a, o, sas]

[OE, dropLocal, a, o, sas]

[OE, approve, a, o, sas]

[OE, reject, a, o, sas]

[OC, commitToDB, a, o, sas]

A share event by an action declares an object to be shared by the action in a shared access set. The movement of data is described above.

commitShare copies an object from a local access set to a shared access set and starts the approval process described above. Sharers are notified that this approval is needed and subsequently sharers approve or disapprove via the approve or reject events. Shared objects with approved states in a shared access set can be committed to the database by commitToDB.

The events dropShare and dropLocal remove an object from shared and local access sets, respectively.

Commits to the database and shared access sets are mediated by use of two dependencies denoted by approvalDependency and newCommitDependency. An action has an approvalDependency on every other sharer action during a shared commit. The dependency is set by commitShare and removed on reject or approve. A newCommitDependency arises whenever a sharer action disapproves of an object committed to a shared access set. In this case, no commit to the database is allowed until a new commitShare to the shared access set and its approval is accomplished.

### *5.2.1 Rules for actions with shared commits*

The rules for events are presented in the order best suited to understanding the dynamics of this sharing control system.

Most of the effect of the share event is produced by the standard movement of objects in the realisation of an object operation event as described briefly above. A share also causes the sharing action and the shared object to be added to the actions and objects associated with the shared access set. The action inherits the approvalDependencys and newCommitDependencys in effect at the time of the share. This will only happen when a commitShare is already in progress. share is illustrated below.

```
[OP, share, va, vo, vsas]→
begin
    if ~ vsas in sharedAccessSetList then createSharedAccessSet (vsas)
    addActionSAS (va, vsas)
    addObjectSAS (vo, vsas)
    ! the sets s and s1 are empty if a commitShare is not in progress
    let s = depWithDepSASObj (approvalDependency, vsas, vo)
    for each dep in s where parent (dep) ≠ va do
        setDependency (approvalDependency, va, parent (dep), vsas)
    let s1 = depWithDepSASObj (newCommitDependency, vsas, vo)
    for each dep in s1 where parent (dep) ≠ va do
        setDependency (newCommitDependency, va, parent (dep), vsas)
end
```

This illustrates some of the manipulation of dependencies. Each dependency is associated with two actions, the dependent action called the child and one depended upon called the parent. Additionally these dependencies are associated with an object and a shared access set. Several set returning functions are provided for querying the dependency structure, as well as other structures. A naming convention has been used to clarify the meaning of these functions. They all start with a mnemonic for the elements returned, *dep* here standing for dependencies. This is followed by *With* and a sequence of mnemonics for the inputs to the function used in selecting the returned values. Here the set of dependencies matching the dependency name, shared access set and object given as input are returned by *depWithDepObjSAS*. The *parent* function returns the parent action identifier of a dependency.

The commitShare event is only allowed if no approval dependencies exist on the object in the designated access set. It copies the local object to the shared access set, sets up the approval dependencies and removes any new commit dependencies.

```
OC, commitShare, va, vo, vsas]→
if ~ (exists (depWithDepObjSAS (approvalDependency, vo, vsas)) and
        vo in objWithSAS (vsas) and
        va in actWithSAS (vsas)) then
begin
    let s = actWithObjSAS (vo, vsas)
    for each action in s do
    for each depAct in s do
    begin
        removeDependency (newCommitDependency, depAct, action,
vsas)
        if depAct ≠ va then
        setDependency (approvalDependency, depAct, action, vsas)
    end
    commitObject (vo, vsas)
end else deleteThisEvent
```

The function *actWithObjSAS* returns the set of action identifiers associated with the shared access set and object whose identifiers are given.

The drop events cause objects to be removed from access sets. dropShare also causes any dependency involving the dropping action, the object and the shared access set to be removed. dropShare is specified by the following.

```
[OE, dropShare, va, vo, vsas]→
begin
        let s = depWithParentObjSAS (va, vo, vsas)
        for each dep in s do deleteDependency (dep)
        let s1 = depWithChildObjSAS (va, vo, vsas)
        for each dep in s1 do deleteDependency (dep)
        deleteObjectSAS (vo, vsas)
end
```

**deleteDependency** acts the same as **removeDependency** but takes the dependency as a single input. dropLocal is specified by:

```
[OE, dropLocal, va, vo]→ deleteObjectLocal (vo, va)
```

The sole effect of the approve is to remove the approvalDependencys on the approving action.

```
[OE, approve, va, vo, vsas]→
begin
      let s = depWithDepParentObjSAS (approvalDependency, va, vo, vsas)
      for each dep in s do deleteDependency (dep)
      let s = depWithDepChildObjSAS (approvalDependency, va, vo, vsas)
      for each dep in s do deleteDependency (dep)
end
```

The effect of reject is to remove the approvalDependencys on its action and to add newCommitDependencys for all sharers.

```
[OE, reject, va, vo, vsas]→
if exists (depWithDepParentObjSAS (approvalDependency, va, vo, vsas)) then
begin
        let s = depWithDepParentObjSAS (approvalDependency, va, vo, vsas)
        for each dep in s do deleteDependency (dep)
        let s = depWithDepChildObjSAS (approvalDependency, va, vo, vsas)
        for each dep in s do deleteDependency (dep)
        let s1 = actWithObjSAS (vo, vsas)
        for each a1 in s1 do
        for each a2 in s1 where a1 ≠ a2 do
              setDependency (newCommitDependency, a1, a2, vo, vsas)
end
```

The commitToDB event causes the commit of an object to the database from the shared access set if no dependencies prevent it. As a side effect the object is copied to the local access set.

```
[OC, commitToDB, va, vo, vsas]→
if exists (depWithDepObjSAS (approvalDependency, vo, vsas)) or
  exists (depWithDepObjSAS (newCommitDependency, vo, vsas))
then deleteThisEvent else commitObject (vo, DB, vsas)
```

The abort event of an action releases all dependencies on it, in effect causing a tacit approval of any shared commits of shared objects.

```
[TE, abort, va]→
begin
        let d = depWithParent (va)
        for each dep in d deleteDependency (dep)
        let d1 = depWithChild (va)
        for each dep in d1 deleteDependency (dep)
        deleteActionHistory (va)
        deleteLocalAccessSet (va)
        removeActionRules (va)
end
```

## 6 Conclusions

An abstract, operational model for specifying flexible concurrency control schemes within a persistent store has been presented. Two models have been specified using CACS: one for atomic transactions and one for actions with shared commit.

The long term success of CACS will be judged by its ability to improve the understanding of control schemes and by its ability to aid the implementation of such schemes. A particular challenge will be using CACS to specify schemes in which different protocols are combined enabling different protocols to be used on the same data at different times as well as allowing the coexistence and interleaving of protocols. Techniques for specifying transactions by means of an abstract machine have also been proposed by[12].

An implementation of CACS will be performed in the Napier persistent environment[10]. Crucial to the efficiency of CACS, and not specified within CACS, is the feedback mechanism by which the control system communicates with the actions. The use of the ACTA formalism to prove properties of CACS specifications will also be studied.

## Acknowledgements

We are grateful for discussion with Panos Chrysanthis, Krithi Ramamritham, Quintin Cutts and Leo Fegaras on ACTA and CACS. We are also grateful to Craig Baker, Fred Brown, Richard Connor, Quintin Cutts, Graham Kirby and Dave Munro for extensive proof reading of the paper. The work was undertaken during a visit by Ron Morrison to the University of Massachusetts and supported in part by National Science Foundation grant IRI-8822181 and in part by ESPRIT II Basic Research Action 3070 - FIDE. Ron Morrison is currently a Visiting Fellow at the Australian National University.

## References

1. Campbell, R. H. and Habberman, A. N. "The specification of process synchronisation by path expressions", *Lecture Notes in Computer Science*, vol 16, Springer-Verlag, 1974.
2. Chrysanthis, P. K. "A Framework for Modeling and Reasoning about Extended Transactions", University of Massachusetts, 1991.
3. Chrysanthis, P. K. and Ramamritham, K. "ACTA: A Framework for Specifying and Reasoning about Transaction", *ACM SIGMOD*, pp. 194-203, 1990.
4. Chrysanthis, P. K. and Ramamritham, K. "A Formalism for Extended Transaction Models", *VLDB-17*, Barcelona, 1990.
5. Dijkstra, E. W. "Cooperating sequential processes", *Programming Languages*, Academic Press, London, pp. 43 - 112, 1968.
6. Dijkstra, E. W. "The Structure of T.H.E. Multiprogramming System", *Communications of the Association for Computing Machinery*, vol 11, 5, pp. 341 - 346, 1968.
7. Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the Association for Computing Machinery*, vol 19, 11, pp. 624 - 633, 1976.

8. Hoare, C. A. R. "Monitors : an operating system structuring concept", *Communications of the Association for Computing Machinery*, vol 17, 10, pp. 549 - 557, 1974.

9. Kung, H. T. and Robinson, J. T. "On Optimistic Methods for Concurrency Control", *Transactions on Database Systems*, vol 6, 2, pp. 213 - 226, 1982.

10. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. "The Napier88 Reference Manual", University of St Andrews, PPRR-77-89, 1989.

11. Moss, J. E. B. "Nested Transaction : An Approach to Distributed Computing", MIT Press, Cambridge, Mass, 1985.

12. Nodine, M., Skarra, A. H. and Zdonik, S. B. "Synchronisation and Recovery in Cooperative Transactions", *Implementing Persistent Object Bases : Principles and Practice*, Morgan Kaufmann Publishers Inc, Palo Alto, Calif., pp. 329 - 344, 1990.