The CORD approach to Extensible Concurrency Control*

George T. Heineman
Worcester Polytechnic Institute
Department of Computer Science
Worcester, MA
heineman@cs.wpi.edu

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY
kaiser@cs.columbia.edu

WPI-CS-TR-96-1

Abstract

Database management systems (DBMSs) have been increasingly used for advanced application domains, such as software development environments, network management, workflow management systems, computer-aided design and manufacturing, and managed healthcare. In these domains, the standard correctness model of serializability is often too restrictive. We introduce the notion of a Concurrency Control Language (CCL) that allows a database application designer to specify concurrency control policies to tailor the behavior of a transaction manager. A well-crafted set of policies defines an extended transaction model. The necessary semantic information required by the CCL run-time engine is extracted from a *task manager*, a (logical) module by definition included in all advanced applications. This module stores task models that encode the semantic information about the transactions submitted to the DBMS. We have designed a rule-based CCL, called CORD, and have implemented a run-time engine that can be hooked to a conventional transaction manager to implement the sophisticated concurrency control required by advanced database applications. We present an architecture for systems based on CORD and describe how we integrated the CORD engine with the Exodus Storage Manager to implement Altruistic Locking.

keywords: Extended Transaction Models, Extensible Concurrency Control, Transaction Manager Component, DBMS architecture

*This paper is based on work sponsored in part by Advanced Research Project Agency order B128 monitored by Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, and in part by the New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare 94013. Heineman was also supported in part by an AT&T Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the government, ARPA, NSF, NYSSTF, or AT&T.

1 Introduction

Advanced database applications (henceforth applications) require more sophisticated concurrency control mechanisms than the standard ACID transaction model provides [4, 29, 11]. For this reason, many *extended transaction models* (ETMs) have been developed [14, 31, 23] that rely on special *semantic information* about the transactions and their operations. There is no consensus, however, as to which ETM is appropriate for advanced applications; most likely, there never will be, since each ETM is optimized for a particular behavior. Therefore, a database management system (DBMS) cannot implement an ETM suitable for all applications. One possible goal is to design a DBMS whose transaction manager (TM) can be tailored to provide the desired ETM for a given application. An even better direction is to show how to extend the TM for existing DBMSs to provide such ability.

Advanced applications include software development environments, network management, workflow management systems, computer-aided design and manufacturing, and managed healthcare. These diverse applications have one feature in common – they have a task manager that stores rich semantic information about the transactions submitted to DBMS. The FlowMark Workflow system [26], for example, stores a workflow process as a directed acyclic graph of activities. In the Oz Process-Centered Environment [5], a process engine interprets task models encoded in planning-style rules. Since the actual implementation of the task manager changes from one application to the next, we do not present details for any particular task manager, nor do we cover situations where the semantic information is implicit and/or arbitrarily spread across multiple parts of the application. We also focus our attention on concurrency control, rather than recovery issues.

Extensible concurrency control is the ability for the TM of a DBMS to alter its decisions regarding how transactions are allowed to behave. It is commonly accepted that semantic information about the transactions is necessary to realize extensible concurrency control. This paper investigates how the TM can acquire semantic information from the task manager of an application, and how to flexibly direct TM to incorporate this information when making concurrency control decisions. This research is performed in the context of showing how to augment existing TMs to support the necessary advanced transaction behavior.

In an application for a large bank, for example, a user's Withdrawal transaction should not be forced to wait while the bank runs a long-duration Balance transaction. The bank application designers could directly modify the existing TM of their DBMS (including rewriting it) to implement the behavior in Figure 1a, but this effort would be costly and have to be repeated for each such scenario. Alternatively, the application could be tightly integrated with the TM (e.g., transactional workflows [15]), granting the application fine-grained control over transaction behavior. The original

if (*Conflict between Balance and Withdrawal*) **then**
  if (*Total-Off* + *Withdrawal Amount* < *1 Million*) **then**
    Total-Off += *Withdrawal Amount*
    Ignore Conflict
  fi
fi

(1a) Simple Case for Bank Policy

if (*Read/Write conflict*) **then**
  if (*can tolerate increase in inconsistency*) **then**
    Update inconsistency totals
    Ignore conflict
  else
    Abort conflicting transaction
  fi
else if (*Write/Write conflict*) **then**
  Abort conflicting transaction
fi

(1b) **ESR** CCL specification

reason for introducing transactions, however, was to avoid such solutions that often reduce to low-level concurrent programming; also for practical reasons, the application and TM should remain separate entities.

Motivating Example

Consider solving this banking example to allow the Balance transaction to observe temporarily inconsistent data. If the TM has a sophisticated interface, such as Encina [12], it might be possible to modify and reimplement the application for an individual case. As more and more special cases arise, however, some model is needed to reduce complexity; as an example, Epsilon Serializability [30] (**ESR**) is an ETM that increases concurrency by allowing bounded inconsistencies to occur. The TM could be reimplemented to support **ESR**, but if the behavior changed yet again, more reimplementation would be necessary. The goal of our research is to provide a solution whereby the application designer need only produce a specification, such as the simplified **ESR** specification in Figure 1b, that tailors the behavior of the TM.

This paper introduces a component used by the TM to tailor its behavior based upon an ETM specification written in a concurrency control language (CCL). Because supporting an ETM requires semantic information from the application, this component employs a generic interface to *extract* the semantic information; a *mediator* layer of special-purpose code insulates the CCL engine from the application. An application designer can thus extend a TM by providing an ETM specification and mediator code, as needed, to extract the necessary semantic information from the target application. We envision that such CCL engines can be attached to existing DBMSs (with only slight modifications to the DBMS) to provide immediate extensibility.

The basic building block of an ETM is a *concurrency control policy* (henceforth, policy) that defines how a TM should react to non-serializable access exhibited by two conflicting transactions. The ETM specification enumerates the differences from serializability, the standard correctness model for most DBMSs. We view approaches that model every database access by all transactions (such

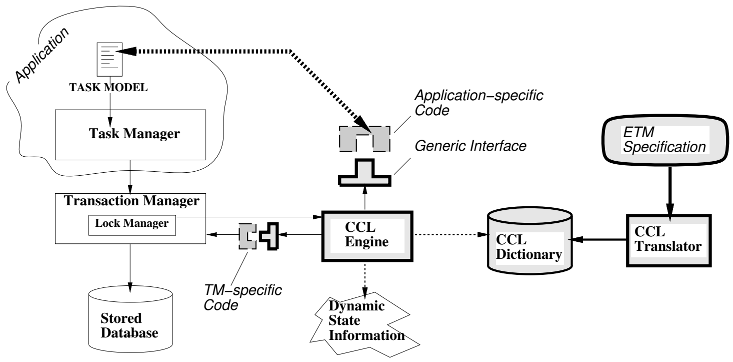

Figure 1: CCL extension to DBMS architecture

as pattern machines [32] or Relative Serializability [1]) as impractical since each transaction that wishes to relax atomicity would first have to analyze the operations of all other potentially affected transactions.

We present the features of a rule-based CCL called CORD (for COoRDination) and its CORD engine, and show how to implement Altruistic Locking (**AL**) [31], a well-known ETM from the literature. We then discuss our experience integrating the CORD engine with the Exodus Storage Manager [9] to implement **AL** within Exodus. Finally, we evaluate our efforts and related work, and summarize our contributions.

2 Architecture

Figure 1 shows the integration of a CCL engine into a DBMS used by an application. A well-defined task manager module stores semantic information about the transactions submitted to the TM. The ETM specification is first translated into a machine-readable format that is loaded by the CCL engine upon initialization. We assume that the DBMS is dedicated entirely for use by the application. All operations submitted to the TM that remain serializable are processed without invoking the CCL engine. When serializability conflicts occur, the TM invokes the CCL engine to locate a policy (if any) that matches the observed conflict. The CCL engine employs a generic interface to extract semantic information from the application using special mediator functions provided by the application designer (shown in dashed boxes).

The CCL engine places certain requirements on the TM, which we assume already has a well-defined API of primitive operations, such as Begin and Commit. First, we must modify the TM to invoke the CCL engine when it detects a conflict (typically by modifying the TM's lock manager). Second, we need `before-` and `after-` callback functions for each API operation so that the CCL

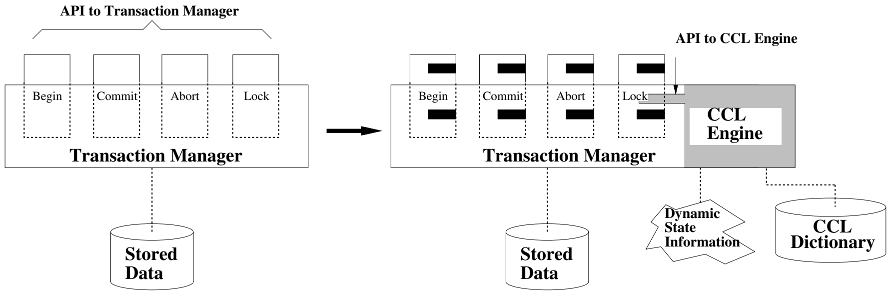

Figure 2: CCL engine integrated with Transaction Manager

```
forward function resolve_conflict (in obj_list, out resolved, out info) : lock_list
type service_status = (SERVICE_OK, SERVICE_DENY, SERVICE_OVERRIDE)

function Lock(in t, in obj_list, in mode) : boolean
    rc := lock_before(t, obj_list, mode);
    if (rc = SERVICE_OVERRIDE) then return (true); fi
    if (rc = SERVICE_DENY) then return (false); fi
    if Lock can be granted then
        Normal transaction behavior
    else
        CS := construct_scenario (t, obj_list, mode);
        locks := resolve_conflict (CS, resolved, cord_info);      ⇒Interface to CCL Engine
        if (resolved = false) then return (false); fi
    fi
    lock_after (t, obj_list, mode);
    return (true);
end
```

Figure 3: Modified **Lock**$(t, obj\_list, mode)$

engine can alter and extend the functionality of the TM. When the TM is requested to lock an object, for example, the `lock_before` callback can invoke a mediator function to determine if the ETM allows the transaction to access the desired objects, and possibly deny the primitive operation. Similarly, a `lock_after` callback can trigger other actions as required by the ETM. These changes are represented by the thin black rectangles in Figure 2; Figure 3 shows the modified **Lock** primitive in more detail. We feel these two features should be part of any DBMS that provides extensible transaction management; in fact, this interface is already very similar to Encina [12]. In Section 5 we describe how we modified the Exodus Storage Manager to include these features. Our success at being able to modify a foreign system leads us to believe that DBMS designers themselves would be able to modify their systems accordingly.

3 ETM Specification

A CORD ETM specification contains a preamble and a set of CORD rules that the CORD engine loads when initialized by the TM. The CORD language is a rule-based CCL influenced by Barghouti's

type	*attribute*
transaction	tid, lockset, parent, subtransactions, top_level
object	oid, name, lockset, class
lock	lock_mode, tid, object

Figure 4: Default CORD semantics

Control Rule Language [3]. The preamble defines mediator functions in an objectcode file that will be dynamically linked with the CORD engine to extend its functionality. Each CORD rule consists of a sequence of policies defined as condition/action pairs that contain the knowledge of how the TM should behave under certain circumstances; CORD rules are thus similar to planning-style rules.

When a conflict is detected between two accesses to an object, the TM invokes the CORD engine to resolve the conflict, constructing a *scenario* containing the object's unique identifier (oid) and class name, and the unique transaction identifiers (tid) of the two conflicting transactions (i.e., if three transactions conflict with each other, the conflicts are handled in pairwise fashion; [19] presents an approach for handling sets of conflicts at once). The CORD engine acts like an expert system, reacting to conflicts by invoking the appropriate policy. If no suitable policy is found, the TM responds to the conflict in its usual fashion.

The CORD language defines an extensible set of data types to model the dynamic state information needed by the policies. The standard data types, shown in Figure 4, model information from the TM: *transaction*, *object*, and *lock*. For clarity of presentation, we assume the TMs are lock-based. For a given transaction T_{17}, for example, the TM may keep a large data structure storing log records, lock sets, and other pertinent information that it needs. The CORD engine maintains its own dynamic state information about T_{17}, separate from the TM, by instantiating an object from its transaction type; since the *tid* is the same, the engine can communicate with the TM to extract detailed information about, and perform actions on, transactions. These data types can be extended to store additional information needed to support a set of policies. The ETM specification, for example, might specify that each transaction has a `task` attribute; the CORD engine would then store this task information within its transaction type.

Each CORD rule is parameterized by the class of object (within the DM) to which it applies, since conflicts occur on individual objects (ENTITY applies to all classes). These CORD rules can be viewed as concurrency control *methods* that an object employs to resolve conflicts (similar to an approach suggested by [18]). Multiple rules defined for the same class are differentiated by a *selection criterion*, allowing the CORD engine to select at run-time the most applicable CORD rule.

Each CORD rule can optionally bind variables (shown as `?var`) that refer to semantic information required by its policies. There are five default variables describing the conflict scenario: `?ConflictObject`, `?Lconflict` (i.e., the conflicting lock being requested), `?Tconflict`, `?Lactive` (i.e., the existing lock), `?Tactive`. For example, if a policy needs to refer to the parent of the transaction causing the conflict, the following variable would be defined within the CORD rule: `?Tpar = ?Tconflict.parent`.

The condition for a policy specifies logical expressions on the rule's variables to determine which one is valid. It can perform simple comparisons of attribute values, such as checking whether the lock mode requested by the active transaction is in read mode. The CORD engine can dynamically load in new code, as determined by the ETM specification, to introduce new functions to be used when evaluating these conditions; as we will see, this is a powerful mechanism.

3.1 Cord Actions

Most TMs can only suspend or abort a transaction to resolve serializability conflicts. In contrast, CORD policies can perform arbitrary actions on transactions as needed to implement a particular ETM. There are two ways that a conflict can be resolved: first, it can be ignored, because it is only a serializability conflict, not a conflict according to the specified ETM; second, the TM can take action, such as suspending or aborting transactions, creating dependencies between transactions to maintain integrity, or dynamically restructuring transactions. In addition to CORD's default actions, described next, new actions can be implemented and dynamically linked with the CORD engine.

The most basic CORD action, `ignore()`, allows non-serializable accesses as directed by the ETM. This action, for example, allows transactions to share partial results with one another or commuting operations to be performed. If there are side-effects of the non-serializable accesses (as determined by the ETM designer), then the CORD engine can maintain *dependencies* between the conflicting transactions. The CORD language allows commit and abort dependencies to be formed between transactions. Briefly, if T_i has an abort dependency on T_j, then if T_j aborts, T_i must also abort. If T_i has a commit dependency on T_j, then T_i cannot commit until T_j finishes (either commits or aborts). The `add_dependency (?Tconflict, ?Tactive, abort)` action, for example, ensures that if the TM ever aborts `?Tactive`, the CORD engine will abort `?Tconflict`. Removing dependencies between transactions, for example, allows a sub-transaction to be treated as top-level. ACTA [10] defines twelve types of dependencies between transactions, but we limit CORD to these two since most of the ACTA dependencies are the domain of the database application, and too tightly bind the TM with the task manager. `suspend (?t1, ?t2)` blocks transaction `?t1` until `?t2` has either committed or aborted, `abort (?t)` aborts a particular transaction, while `notify (?t1, msg)` action delivers a message to the application on behalf of transaction `?t1`. Other actions are defined in [19].

The DBMS engineers are responsible for integrating the TM with the CORD engine. In addition to the effort outlined in Section 2, this means that mediator functions need to be written for each of CORD's default actions to interface to the specific TM. For example, the CORD engine must map its `suspend` action to specific capabilities in the TM, as shown by example in Appendix B. The mediator for the `notify` CORD action is written by the application designer to interface TM with the application. Appendix B contains an example of this mediator function. These examples of mediation show how the CORD engine is insulated from the details of the other system components.

3.2 Motivating Example Revisited

To return to our opening example, we now present a CORD implementation of **ESR** [30]. Each Epsilon Transaction (ET) has a specification (called an ϵ-spec) of its allowed *import* and *export* inconsistency. A transaction imports inconsistency by reading the uncommitted results of an update transaction; this update transaction is then considered to have exported inconsistency. Separate from these ϵ-spec values, each data item has its own data-ϵ-spec for the amount of inconsistency it allows. Note that **ESR** is equivalent to Serializability if all transaction-ϵ-spec values are 0. In this paper, we implement a restricted form of **ESR** that does not allow update transactions to import consistency; a more general form of **ESR** has been implemented in [19].

Each ET maintains a fixed *ImpLimit* (*ExpLimit*) as part of its ϵ-spec that determines the bounded amount of inconsistency it can import (export). Each ET also maintains a running import (export) accumulator that it updates whenever it imports (exports) inconsistency. When an ET attempts to read and write a data item x, the inconsistency inherent in x is added to the ET's inconsistency counters. Each data item maintains an accumulator of inconsistency used to check against its data-ϵ-spec. Summing up, the CORD engine must store four new pieces of information with each ET (*ImpLimit, ExpLimit, import_accumu, export_accumu*) and two new pieces of information with each data item (*data-ϵ-spec* and *data_accumu*). We assume here that the data-ϵ-spec value is stored in the DBMS itself (as an optional attribute for each object).

The CORD engine maintains this state information about ETs and enforces the inconsistency limits. Using callback functions, the task manager (that is requesting the locks) can be queried to find out how much each ET will alter the data item's value. For example, **ESR::lock_after** in Figure 6 is invoked to create a CORD data structure of type `ESR_accumu` to store the inconsistency introduced for each object as ETs proceed. The `DBMS::` functions in Figure 6 retrieve the necessary information using the API of the underlying DBMS.

```
cord_rules
    object_code   esr.so
    condition     valid_tolerance (object, transaction, transaction)          Preamble
    action        increment_accumu (object, transaction, transaction)

epsilon_extension [ ESR_ENTITY ]
    selection_criterion:                                                      Rules
        lock.lock_mode: W, R
    body:
        # Conflict between an update (that requests the lock) and a query (that has the lock).
        # Verify that the resulting increase in inconsistency will be tolerated.
        if (and (?Lconflict.lock_mode = W)
                (valid_tolerance(?ConflictObject, ?Tconflict, ?Tactive))) {
            increment_accumu(?ConflictObject, ?Tconflict,?Tactive)
            ignore()
        }

        # Conflict between ?Tconflict (requesting query lock) and ?Tactive (has update)
        if (and (?Lactive.lock_mode = W)
                (valid_tolerance(?ConflictObject, ?Tactive, ?Tconflict))) {
            increment_accumu(?ConflictObject, ?Tactive, ?Tconflict)
            ignore()
        }

        # The following conditions match when inconsistency is too much
        if (?Lconflict.lock_mode = W) { abort(?Tconflict) }
        if (?Lactive.lock_mode = W) { suspend(?Tconflict, ?Tactive) }
end_body
```

Figure 5: CORD rule for Epsilon Serializability

```
procedure ESR::lock_after (in t, in obj_list, in mode)
    for oid in obj_list do
        if (DBMS::get_att_value (o, "d_espec", d_espec)) then
            DBMS::get_att_value (o, "value", d_value);
            if (not ESR_globals::member (oid)) then
                one = ESR_accumu::new (obj_id:oid, consistent_value:d_value, espec:d_espec, accumu: 0);
                ESR_globals::insert (one);
            fi
        fi
end
```

Figure 6: ESR::lock_after mediator algorithm

To complete our **ESR** implementation, the `epsilon_extension` CORD rule (in Figure 5) contains four policies to allow a query ET and an update ET to conflict if the transaction ϵ-spec values of the involved ETs are satisfied. The mediator functions referenced in this CORD rules are dynamically loaded from `esr.so` and are shown in pseudo-code in Appendix D. The TM does not to be reimplemented to support the **ESR** behavior; only special-purpose mediator code needs to be written that extends its behavior as desired.

3.3 Extracting Semantics

The novel feature of the CORD language is that it allows the application designer to model the desired semantic information in the application. For each piece of semantic information, an *access* mediator

AL1 Two transactions may not simultaneously hold *conflicting* locks on the same object unless one of the transactions first donates the object.

AL2 If T_a is *indebted* to T_b, then it must be completely in the wake of T_b until T_b performs its first Unlock operation.

$d(a)$ The set of transactions that have donated (and not unlocked) a.
$in(a)$ The set of transactions that readers of a must be in the wake of.
$W(T)$ The set of transactions whose wakes T is completely within.
$J(T)$ The set of transactions whose wakes T must be completely within (based on *AL1* and *AL2*).

Figure 7: **AL** requirements

is implemented (by the application designer) that will extract the information at run-time if needed by the CORD engine. At startup, the CORD engine dynamically loads in the code for the access mediators from the ETM specification. The CORD engine employs a generic mediator interface to extract the desired semantic information (as shown in Figure 2). If either the application or the TM is replaced, only the specific mediator functions need to be rewritten; the CORD engine remains unchanged.

4 Example Extended Transaction Model

We now present a full CORD solution to extending a TM for Altruistic Locking (**AL**) ETM [31]. **AL** is an extension to two-phase locking (**2PL**) [13] that accommodates long-lived transactions. Under **2PL**, short transactions will encounter serious delays since database resources can be locked for significant lengths of time. In **AL**, several transactions can hold conflicting locks on a data item if constraints *AL1* and *AL2* in Figure 7 are satisfied. In this example, read and write locks have the usual semantics. Using the *Donate* operation – a new TM primitive operation – a transaction announces to the database that it will no longer access a given data item, thus allowing other transactions to access it (constraint *AL1*). A donate is not an unlock and the transaction must still explicitly unlock data items that it has donated (typically at the end of the transaction) – the transaction is free to continue locking data items even after some have been donated.

A transaction enters the *wake* of transaction T_i when it locks an object that has been donated (and not yet unlocked) by T_i. A transaction is *completely* in the wake of T_i if all the objects it locks are donated by T_i. If T_j locks an object that has been donated by T_i, T_j is *indebted* to T_i if and only if the locks conflict or an intervening lock by a third transaction T_k conflicts with both. Even though two read locks are compatible, the second read becomes indebted to the first when an intervening write occurs. Initially, for all a and T, $J(T) = d(a) = in(a) = \emptyset$. By default, as each

```
procedure AL::begin_after (in T)
    W(T) := ActiveSet;
    ActiveSet := ActiveSet ∪ {T};
end

procedure AL::complete_tx (in T)
    // T can no longer impact any transaction, so update the appropriate W(t) sets
    ActiveSet := ActiveSet − {T};
    foreach t ∈ ActiveSet do
        if (T ∈ W(t)) then W(t) := W(t) − {T};
end

function AL::lock_before (in T, in a, in mode) : service_status
    // If a hasn't been donated, SERVICE_OK is returned if T is not completely within the wake of another
    // transaction. Otherwise, SERVICE_OK is returned if T remains completely in the wake of J(T).
    w := W(T) ∩ d(a);
    i := J(T) ∪ in(a);
    if (i ⊆ w) then return (SERVICE_OK); fi
    return (SERVICE_DENY);
end

procedure AL::lock_after (in T, in a, in mode)
    // Now that lock is assure, we update the appropriate sets. Cannot do this in lock_before
    J(T) := J(T) ∪ in(a);
    W(T) := W(T) ∩ d(a);
    return (SERVICE_OK);
end

function AL::unlock_before (in T, in a, in mode) : service_status
    // Removes downstream transactions from wake of T, and maintains in(a) and d(a)
    d(a) := d(a) − {T};
    in(a) := in(a) − {T};
    foreach t ∈ ActiveSet do
        if (T ∈ J(t)) then J(t) := J(t) − {T};
    return (SERVICE_OK);
end

procedure AL::donate (in T, in a)
    // New operation in DBMS API
    d(a) := d(a) ∪ {T};
end

function AL::is_donated (in a) : boolean
    // New CORD condition
    return (not empty (d(a)));
end

procedure AL::update_in_set (in a, in Lact)
    // New CORD action
    if (Lact.lock_mode = R) then in(a) := in(a) ∪ { Lact.tid }; fi
end
```

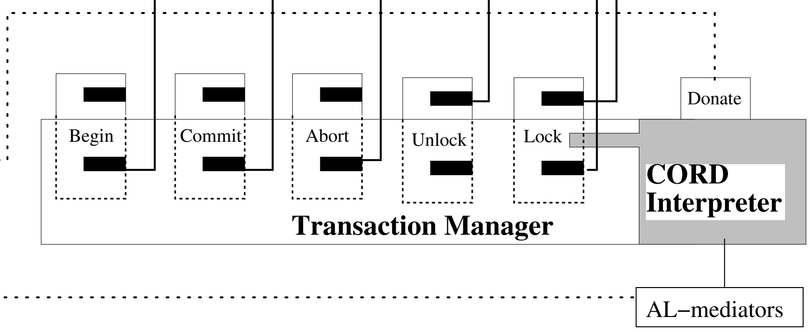

Figure 8: Mediator extensions for **AL**

```
cord_rules
    object_code   alt.so                          # Dynamically linked mediator code
    condition     is_donated (object)             # New AL conditions here          Preamble
    action        update_in_set (object,lock)     # New AL actions here

AL_rw [ ENTITY ]
    selection_criterion: lock.lock_mode: W, R                                        Rules
    body:
        # Allow conflict on donated object; maintain indebted relationship
        if (is_donated (?ConflictObject)) {
            add_dependency( ?Tconflict, ?Tactive, abort)
            update_in_set( ?ConflictObject, ?Lactive)
            ignore()
        }
    end_body

AL_ww [ ENTITY ]
    selection_criterion: lock.lock_mode: W, W
    body:
        # Allow conflict on donated object
        if (is_donated (?ConflictObject)) {
            add_dependency( ?Tconflict, ?Tactive, abort)
            ignore()
        }
    end_body
```

Figure 9: CORD rule to support **AL**

transaction begins, it enters the wake of all active transactions; elements are removed and inserted into $W(T)$ based upon the behavior of T.

The CORD engine maintains dynamic state information about the wakes of transactions (i.e., $W(T)$ and $J(T)$) and enforces the indebted constraint *AL2*. $W(T)$ is calculated by tracking the set of active transactions with the `begin_after` mediator **AL::begin_after**. We extend the **Lock**($T, obj\_list, mode$) primitive operation (shown in Figure 3) by binding the `lock_before` mediator to **AL::lock_before**. The $J(T)$ and $W(T)$ sets are updated by the `lock_after` mediator, **AL::lock_after**. These sets cannot be updated in the `lock_before` mediator, otherwise a locking conflict that failed to set a lock would incorrectly update this information. The **Unlock** operation is extended by binding its `unlock_before` mediator to **AL::unlock_before**; this mediator and the new **AL::donate** primitive operation manage $d(a)$. When transactions commit (abort), the `commit_after` (`abort_after`) mediator, bound to **AL::complete_tx**, updates *ActiveSet*. The **AL** protocol presented in [31] upgrades read locks to write locks solely to preserve the indebted relationship between transactions. Instead of altering the locks held by the lock manager, our solution maintains several sets for each database object a and transaction T, as shown in Figure 7.

Our implementation is completed by two CORD rules. The `AL_rw` CORD rule in Figure 9 is invoked for all read/write conflicts on any object. The policy of this rule handles situations when a write lock is requested on an object that `?Tactive` previously read and donated; this carefully maintains the indebted relation, *AL2*. The policy in `AL_ww` allows multiple writers if the conflicting object was donated first. Figure 8 shows the interaction between TM and the CORD engine. As transactions request primitive operations from the TM's API, the various mediator functions (encapsulated by ovals) are invoked through callbacks. When the CORD engine evaluates its policies, it employs the new **AL::is_donated** condition and **AL::update_in_set** action as defined in the preamble. Neither the TM nor the CORD engine were altered in any way; only the ETM specification and mediator code (included dynamic code) were added for this solution.

5 Example Integration with DBMS

We next integrated the CORD engine with the Exodus Storage Manager [9]. In Exodus, client applications share memory pages from a virtual "volume" residing on a storage manager server. A locking conflict, therefore, occurs on a particular page and volume. Objects in Exodus can be small enough to fit several to a page, or one object can be spread across many pages. Client applications requests objects from the server by page location.

We modified the Exodus lock manager (LM) to invoke the CORD engine when it detects a lock conflict. The CORD engine then uses the available semantics (in this case, only the lock modes being requested) to determine if a CORD rule matches the conflict scenario. LM had twelve individual locations where it checks a lock matrix to determine if two lock modes are compatible; for example, `if (!LM_Compat[lockEntry->lockMode][requestMode])` was replaced with:

```
if (!cord_compatible(lockEntry->headerList.transRec, lockEntry->lockMode,
                     transRec, requestMode, lockEntry->lockHeader))
```

The extra parameters refer to the actual transaction structures in Exodus. Five such expressions were replaced with calls to `cord_compatible()`, listed in Appendix A. Five others were changed to call a similar function, `cord_compatible_upgrade()`, used when the client upgrades its lock; the last two points were changed to `cord_compatible_list()`.

In `cord_compatible()`, a conflict scenario is created, with the two conflicting locks `lock1` and `lock2`. Note how the necessary information for CORD is extracted from the `TRANSREC`, `LOCKID`, and `LOCKHEADER` Exodus data structures, an example of directly inspecting the data structures of the TM. The entry point to the CORD engine is the `resolve_conflict()` function, which takes a conflict scenario and returns two values: `resolved` determines whether the conflict was successfully resolved, and `cord_info` records the actions CORD performed to resolve the situation.

To complete the infrastructure, we augmented the interface for each primitive operation in the transaction API to allow callback functions. At various points in Exodus, calls were inserted:

```
  if (lock_before (transRec, lockid->page.pid.page, requestMode) != 1)
    return (esmFAILURE);
```

that allowed `before-` and `after-` mediators to be invoked.

5.1 Altruistic Locking in Exodus

To complete the **AL** implementation in Exodus, we extended the communication protocol between the client and server to provide a new *DonatePage* operation. Exodus allows client programs to scan through a collection of objects (called a file) in the storage manager, guaranteeing that all objects in the file will be accessed exactly once during the traversal. Since several objects can reside on a page, we need to be careful to donate a page only when it is no longer needed by the client: whenever the scan iterator retrieves a new page from the server, the client application calls a new Exodus operation, *DonatePage*, directing the server to donate the previous page which will no longer be used by that client. When the scan iterator is complete, the client donates the last page in the scan before committing. In conjunction with this new function, the same CORD rule from Figure 9 is used to allows particular behavior in the TM.

We encountered two "feature interaction" problems. The first, which we call the *double-buffering* problem, occurs when a client donates a page it has updated. Recall from Section 4 that under **AL**, the transaction that donates a page does not unlock it. If the transaction only flushes its pages onto durable storage when it commits, future transactions that read this page will see the old value. This problem could have been foreseen since **AL** does not guarantee failure-atomic transactions and Exodus implements log-based recovery based on the ARIES algorithm [27]. We therefore need to flush these pages to storage whenever a page is donated. Also, since the client forces pages to the server when a transaction commits, we must not force already donated pages. Thus, implementing the Donate operation itself required some effort. In [31], the authors discuss other problems, related to recovery, that **AL** might introduce. The second problem reveals a subtle interaction between granularity locking [17] and **AL**. When scanning a file, Exodus acquires a file lock instead of acquiring a separate lock for each page in the file. The transaction scanning a file, however, cannot donate this file lock until it completes the scan. To work around this problem, we programmed a client application to mimic a scan by manually requesting each page in order.

6 Evaluation

In a previous paper [20], we presented an architecture for integrating a TM component into environment frameworks. We described how a mediator architecture allowed us to implement distributed

ETM	Number of CORD rules	Number of mediators (and length)
strict **ESR** in Oz	1	8 (325 lines)
general **ESR** in Oz	2	10 (446 lines)
AL in Exodus	2	10 (429 lines)

Figure 10: Statistics on implementing ETMs

two-phase commit on top of a set of centralized (but extensible) TMs. In this paper, we have shown how to extend a TM to tailor its behavior using our CORD engine. The statistics shown in Figure 10 summarize our effort to use the CORD engine to support a particular ETM. The size and number of mediator code is reasonable, especially considering the extra benefits of being able to tailor an ETM on top of an existing DBMS. The **ESR** CORD experiment was implemented and tested within Oz [5], a rule-based process-centered environment. The **AL** solution was first designed and tested for a small demonstration environment and then was reproduced within Exodus. The same CORD engine was used for all experiments: only the mediator code and ETM specifications changed.

6.1 Support for Locking

The TM can detect serializability conflicts using either locking, timestamp ordering (**TO**) [7], optimistic concurrency control (**OCC**) [24], or any other equivalent method. In addition to being the most popular, we feel that locking is most suitable for extensible concurrency control. **OCC** is inappropriate for several reasons. First, **OCC** determines conflicts after they occur, when a transaction, T, attempts to commit. If T conflicts with a previous transaction, T_c, that has already committed, it might not be possible for the TM to extract any semantic information about T_c, since the information for the task that employed T_c might no longer be available. Second, if negotiation were used to resolve the conflict, such interaction must occur as the conflict occurs, not (possibly) long after the transactions conflict. A timestamp-based protocol would not be as efficient, either. If the CORD engine needs to inspect the objects a transaction has accessed, for example, locking already provides lock sets for each transaction, but there is no similar concept in **TO**; the CORD engine would have to duplicate this information. As much as possible, we want the CORD engine to only maintain dynamic state information that is not already managed by the TM. One limitation of our approach is that it does require changes within the TM, but the API changes are minimal and in-line with standard APIs (as in Encina).

6.2 Effects on application

The CORD actions necessarily affect the advanced database application. In client/server architectures, the client typically waits synchronously for a reply from the server; to suspend a transaction, the TM can simply delay its response. If the TM is bundled together with the database application in one single-threaded operating system process (for example, a workflow engine combined with a database), suspending a transaction is not easy at all, since multiple contexts need to be carefully maintained and restarted at the correct times. In the context of Barghouti's Control Rule Language, we successfully implemented the CORD `suspend` action in the MARVEL process-centered environment [25], but this required significant portions of process engine (i.e., MARVEL's task manager) to be reimplemented to be aware that a process task could be suspended at any point during its execution. General solutions to the problem of how an application should react to ETMs are outside the scope of this paper.

The actions in the CORD language must be matched to the capabilities present in the TM and the task manager. When attaching the CORD engine to an existing TM, the CORD primitive actions (i.e., abort, suspend) are parameterized to invoke corresponding primitives from the API for the TM. If the TM cannot suspend transactions, for example, no CORD rule can use this primitive.

7 Related Work

The ACTA framework [10] constructs a theoretical model that helps reason about and compare different ETMs. An ETM can be completely characterized by a list of axiomatic definitions. This specification, however, cannot readily be used by a DBMS to *implement* an ETM for an application. Inspired by ACTA, Asset [8] allows users to define custom transaction semantics for specific applications. It provides transaction primitives that can be composed together to define a variety of ETMs. Asset still needs some higher layer, however, to appropriately organize its primitives based upon the available semantic information.

The Transaction Specification and Management Environment (TSME) [16] is closest to our approach. TSME provides a transaction specification language <u>and</u> a programmable transaction management mechanism (TMM) that configures a run-time environment to support a specified ETM. TMM translates a transaction model specification into a set of instructions and assembles run-time support from a transaction processing toolkit. One drawback is that all the components of the resultant system appear to be built from scratch, and there seems to be no way to integrate a TMM with an existing DBMS.

Barga and Pu have designed a Reflective Transaction Framework to implement extended transaction models [2]. Using transaction adapters, add-on modules that are built on top of an existing TM, they show how to extend the underlying functionality of the TM. In their case, they extended the Encina [12] transaction processing monitor by capitalizing on the callback functionality provided by Encina. This is very similar to our approach at utilizing the mediator architecture of our TM component. The primary difference as compared to our work is that we have designed the CORD language for specifying the extensions to TM, while they still followed a programming approach. We foresee that our engine can be easily integrated with Barga and Pu's framework.

In lock-based TMs, the most common means of extension is to provide additional lock modes, or allow new ones to be defined. Most lock-based systems use a matrix to record lock compatibility information (e.g., Exodus and ObServer [21]). Modifying this information would be difficult in systems where there is either no defined "matrix" of locks (e.g., the logic for compatible locks is spread throughout the system), or the defined matrix is not meant to be altered (e.g., the matrix is stored in a C header file and lock modes are pre-defined constants). If new lock modes can be added to a matrix table, the core functionality of the system will be affected when new lock modes are requested. To use these new lock modes, however, the application designer might have to modify and rebuild parts of the system.

Some DBMSs provide support for defining new lock modes as needed, without any recompilation. The TM in the MARVEL process-centered environment [6], for example, determines its lock modes from a fully-configurable lock matrix file; each MARVEL task encodes the lock modes it will request from the TM. A configurable lock-matrix, however, is not powerful enough to provide fine-grained control; for example, **AL** could not be implemented solely by a complex matrix (as in Papyrus [28]). The TM could always be modified to acquire semantic information when determining lock conflicts. Barghouti [3] designed a TM that employed a special-purpose language for programming concurrency control policies for rule-based software development environments (RBDEs). His TM extracted seven pieces of semantic information from RBDEs and had a language for specifying concurrency control policies. This approach was hard-wired in that the TM directly inspected data structures from the RBDE and the language was specially designed for RBDE. Our work generalizes and extends Barghouti's ideas for wider applicability.

An alternative to serializability as a correctness model is the *checkout model*. In checkout, transactions operate on private copies of data that are checked out from a repository. The only contention for shared objects occurs when a transaction checks in/out an object. We view checkout, versions, and configurations as the domain of the application rather than something to be imposed by the TM. See [22] for a survey of extended checkout models.

8 Contributions and Future Work

Advanced database applications use databases to store information but they require more sophisticated concurrency control policies than standard DBMSs provide. Fortunately, such applications contain semantic information that describes their transactional needs. The transaction manager needs to incorporate such semantic information to provide the appropriate services to these advanced database applications.

Our main contributions are:

- A mediator architecture that allows generic extraction of semantic information from an advanced database application, as needed to support an extended transaction model.
- The CORD language, a sample Concurrency Control Language that specifies the extensions to serializability needed for an extended transaction model.
- A CORD run-time engine that incorporates the semantic information to extend the transaction manager for a DBMS. The CORD engine uses the semantic information extracted from the application to match concurrency control policies in a CORD specification.
- Successful application of the CORD approach to implementing **AL** within Exodus.

For future work, we plan on carrying out more experiments with CORD and existing DBMSs. Once a particular set of CORD rules becomes fixed for an ETM, the run-time support would be more efficient if the rules could be compiled into native code, thus avoiding the cost of interpretation; we are currently investigating such an approach. We have focused our attentions on the concurrency control aspects of ETMs, but have not discussed the interaction and relationship that concurrency control has on recovery. In the same way that CORD rules can tailor concurrent behavior, it seems likely that a similar language-based approach can be used to program the recovery of ETMs.

References

[1] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transactions. In *ACM-SIGMOD/PODS 1994 International Conference on Management of Data*, pages 139–149, Minnesota, USA, May 1994. ACM.

[2] Roger Barga and Calton Pu. A practical and modular method to implement extended transaction models. In *21st International Conference on Very Large Data Bases*, Zurich, Switzerland, 1995.

[3] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February 1992. CUCS-001-92.

[4] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[5] Israel Z. Ben-Shaul and Gail E. Kaiser. *A Paradigm for Decentralized Process Modeling*. Kluwer Academic Publishers, Boston, MA, 1995.

[6] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.

[7] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In Lochovsky and Taylor, editors, *Proceedings of the 6th Conference on Very Large Databases*, Montreal, Canada, October 1980. Morgan Kaufmann Publishers.

[8] A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *1994 ACM SIGMOD International Conference on Management of Data*, pages 44–54, Minneapolis MN, May 1994. Special issue of *SIGMOD* Record, 23(2), June 1994.

[9] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, chapter 7.3, pages 474–499. Morgan Kaufman, San Mateo CA, 1990.

[10] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[11] Umesh Dayal, Hector Garcia-Molina, Mei Hsu, Ben Kao, and Ming-Chien Shan. Third generation TP monitors: A database challenge. *SIGMOD Record*, 22(2):393–397, June 1993.

[12] Encina Product Overview, Transarc Corp, Pittsburgh, PA. `http://www.transarc.com`.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.

[14] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irv Traiger, editors, *ACM SIGMOD 1987 Annual Conference*, pages 249–259, San Francisco CA, May 1987. Special issue of *SIGMOD Record*, 16(3), December 1987.

[15] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[16] Dimitris Georgakopoulos, Mark Hornick, Piotr Krychniak, and Frank Manola. Specification and management of extended transactions in a programmable transaction environment. In *10th International Conference on Data Engineering*, pages 462–473, Houston TX, February 1994.

[17] J. Gray, R. Lorie, and G. Putzolu. Granularity of locks and degrees of consistency in a shared database. In *International Conference on Very Large Data Bases*, pages 428–451. Morgan Kaufmann, 1975.

[18] Thanasis Hadzilacos and Vassos Hadzilacos. Transactions synchronization in object bases. *Journal of Computer and System Sciences*, 43:2–24, 1991.

[19] George T. Heineman. *A Transaction Manager Component Supporting Extended Transaction Models*. PhD thesis, Columbia University, 1996. Forthcoming.

[20] George T. Heineman and Gail E. Kaiser. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, pages 305–313, Seattle WA, April 1995.

[21] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Automation Systems*, 5(1):70–95, January 1987.

[22] Gail E. Kaiser. Cooperative transactions for multi-user environments. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 20, pages 409–433. ACM Press, New York NY, 1994.

[23] Gail E. Kaiser and Calton Pu. Dynamic restructuring of transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 8, pages 265–295. Morgan Kaufmann, San Mateo CA, 1992.

[24] H. T. Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[25] Programming Systems Lab. Marvel 3.1.1 administrator manual. Technical Report CUCS-038-93c, Columbia University Department of Computer Science, 1993.

[26] F. Leymann and D. Roller. Business processes management with FlowMark. In *39th IEEE Computer Society International Conference (CompCon), Digest of Papers*, pages 230–233, San Francisco, March 1994.

[27] C. Mohan, D. Haderle, B. Lindsay, H. Piradesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 1991.

[28] Marie-Anne Neimat and Kevin Wilkinson. Extensible transaction management in Papyrus. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 503–511, Kona HI, January 1990.

[29] Erich Neuhold and Michael Stonebraker (editors). Future directions in DBMS research. *SIGMOD Record*, 18(1):17–26, March 1989.

[30] Calton Pu. Generalized transaction processing with epsilon-serializability. In *Proceedings of the 1991 International Workshop on High Performance Transaction Systems*, 1991.

[31] Kenneth Salem, Hector Garcia-Molina, and Jeannie Shands. Altruistic locking. *ACM Transactions on Database Systems*, 19(1):117–165, March 1994.

[32] Andrea Helen Skarra. *A Model of Concurrency Control for Cooperating Transactions*. PhD thesis, Brown University, May 1991.

A Exodus Mediator Code

```
\*begin*********************************************************************/
int
cord_compatible(TRANSREC *transActive, int activemode, TRANSREC *transConflict,
                int conflictmode, LOCKHEADER *thelock)
\*
*end***********************************************************************/
{
  int        resolved, rc;
  DS_PTR     (conflict_list, PERN_OBJ_LIST) = NULL;
  DS_PTR     (results, PERN_LM_LIST);
  DS_PTR     (lock_active, PERN_LM_LOCK);
  DS_PTR     (lock_conflict, PERN_LM_LOCK);
  LOCKID     *exodus_lock;
  void       *cord_info;

  /* Original Exodus check for compatible locks */
  if (LM_Compat[activemode][conflictmode])
    return (TRUE);

  /* If we are here, then Exodus thinks a conflict has occurred. */
  exodus_lock = &(thelock->hashList.lockid);
  conflict_list = DS_instantiate (PERN_OBJ_LIST, NULL);

  /* use the Exodus lockid as the object identifier */
  lock_active = DS_instantiate(PERN_LM_LOCK,
                               "mode",    activemode,
                               "tid",     transActive->tid,
                               "obj_id",  exodus_lock, NULL);

  lock_conflict = DS_instantiate(PERN_LM_LOCK,
                                 "mode",    conflictmode,
                                 "tid",     transConflict->tid,
                                 "obj_id",  exodus_lock, NULL);

  (void) add_conflict_to_list (conflict_list, lock_active, lock_conflict, (int) exodus_lock);

  /* Resolve the conflict: results is a list of un-resolved conflicts */
  results = resolve_conflict (conflict_list, &resolved, &cord_info);

  return (resolved);
}
```

B Cord Action Mediator

The `suspend(?t1, ?t2)` operation in CORD suspends transaction `?t1` until `?t2` completes. This mediator function interfaces the CORD engine with an underlying TM. In this example, CORD interfaces with the PERN transaction manager component [19].

```
/* Function in Pern's API */
extern void tx_SUSPEND (int tid1, tid2);

/*begin**********************************************************************************************/
int
CORD_suspend(DS_PTR (scenario, CORD_CONFLICT), DS_PTR (t1, PERN_TX_LIST), DS_PTR (t2, PERN_TX_LIST)
/*
   The first parameter is always the conflict scenario.  Return TRUE on success
*end************************************************************************************************/
{
```

```
  int tid1, tid2;

  tid1 = DS_get_int (t1, "tid");
  tid2 = DS_get_int (t1, "tid");

  tx_SUSPEND(tid1, tid2);
  return (TRUE);
}
```

The `notify(?t1, msg)` operation in CORD sends a message to the user controlling the task that is responsible for the given transaction. In this case, the task manager has a function in its API to send a message to the client application. This mediator function retrieves the "client_id" attribute from the CORD base type and uses that as the first argument to the API invocation.

```
/* Function in task manager API */
extern void send_client_message (int client_id, char *buf);

/*begin*******************************************************************************/
int
CORD_notify(DS_PTR (scenario, CORD_CONFLICT), DS_PTR (pern_tx, PERN_TX_LIST), char *buf)
/*
   The first parameter is always the conflict scenario.  Return TRUE on success
*end*********************************************************************************/
{
  send_client_message (DS_get_int (pern_tx, "client_id"), buf);
  return (TRUE);
}
```

C Cord Extensible Action

`update_in_set(?object, ?lock)` first checks that the conflicting lock is a read. In this case, an intervening write has just been allowed, so the indebted set $in(a)$ is updated accordingly.

```
/*begin******************************************************************************/
int
update_in_set (DS_PTR (conflict, CORD_CONFLICT), int oid, DS_PTR (lock, PERN_LM_LIST))
/*
  Increment obj.in if this lock is a read.
*end********************************************************************************/
{
  int     tid;
  DS_PTR  (tidrange, PERN_TID_RANGE);

  if (DS_get_int (lock, "mode") == READ_MODE)
    {
       obj = Hashtable_member (GLOBALS, "ID_HASH", oid);
       if ((tidrange = DS_get (obj, "in")) == NULL)
         {
           tidrange = DS_instantiate (PERN_TID_RANGE, NULL);
           DS_set (obj, "in", tidrange);
         }
       tid = DS_get_int (lock, "tid");
       (void) Range_reserve_integer (tidrange, "tids", tid);
    }
}
```

D Cord ESR mediators

```
function ESR::valid_tolerance(in oid, in UET_tx, in QET_tx) : integer
    delta := ESR::TM_get_delta(UET_tx);
    new_exp := delta + UET_tx.export_accumu;
    if (new_exp > UET_tx.ExpLimit) then return (false); fi
    new_imp := delta + QET_tx.import_accumu;
    if (new_imp > QET_tx.ImpLimit) then return (false); fi
    obj := ESR_accumu::member(oid);
    if (obj.accumu + delta > obj.espec) then return (false);fi
    return (true);
end

procedure ESR::increment_accumu(in conflict, in oid, in UET_tx, in QET_tx)
    delta := ESR::TM_get_delta(UET_tx);
    UET_tx.export_accumu := UET_tx.export_accumu + delta;
    QET_tx.import_accumu := QET_tx.import_accumu + delta;
    obj := ESR_accumu::member(oid);
    obj.accumu := obj.accumu + delta;
end
```