A Practical and Modular Method to Implement Extended Transaction Models

Roger Barga and Calton Pu
email: {barga,calton}@cse.ogi.edu
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
P.O. Box 91000 Portland, OR 97291-1000

Abstract

Although many extended transaction models have been proposed [Elm93], few practical implementations exist and even fewer can support more than one model. We present the *Reflective Transaction Framework*, as a practical and modular method to implement extended transaction models. We achieve modularity by applying the Open Implementation approach [Kic92] (also known as meta-object protocol [KdRB91]) to the design of the reflective transaction framework. We achieve practicality by implementing on top of a commercial transaction processing monitor. For our implementation of the reflective transaction framework, we introduce *transaction adapters*, add-on modules built on top of existing commercial TP components, such as Encina, that extend their functionality to support extended transaction features and semantics. Since our framework design is based on the transaction processing monitor architecture [GR93], it is widely applicable to many modern TP monitors. The reflective transaction framework enables us to implement a wide range of independently proposed extended transaction models, which we demonstrate by implementing the split/join model [PKH88] and cooperative transaction groups [MP92, RC92].

1 Introduction

Although the ACID properties(atomicity, consistency, isolation, and durability) [Reu82] of traditional transactions in Online Transaction Processing (OLTP) systems have proven very useful in banking and airline reservations, they are stronger than necessary for many applications and in some cases prevent desirable sharing of information. Numerous extended transaction models have been proposed [Elm93] which relax the ACID properties provided by transactions, replacing them with weaker guarantees. Despite their popularity, relatively little has appeared in the literature on implementing extended transaction models, and a key remaining question is whether or not they are practical.

In this paper, we present the *Reflective Transaction Framework* as a practical and modular method to implement extended transaction models. We achieve modularity by applying the Open Implementation approach [Kic92], also known as meta-object protocol [KdRB91], to design the reflective transaction framework. We achieve practicality by basing the implementation of the reflective transaction framework on the Transaction Processing (TP) Monitor Architecture [GR93], which is widely applicable to many modern commercial TP systems.

One goal of our research is to bring together research advances in extended transaction models and commercial TP monitors, an interaction from which both sides may benefit. To this end, our implementation of the reflective transaction framework introduces *transaction adapters*, add-on modules built on top of existing commercial TP components to extend their functionality in support of extended transaction features and semantics. Transaction adapters take advantage of existing transaction services to the extent possible, eliminating unnecessary infrastructure development and facilitating technology transfer. Insight that a commercial TP monitor could be used was derived, in part, from previous research in which we extended Encina [Encina], a commercial TP facility distributed and supported by Transarc, to implement Epsilon Serializability [PC93]. In this paper, we again take advantage of Encina's modularity to implement the reflective transaction framework.

The reflective transaction framework enables us to implement a wide range of extended transaction models, and we illustrate this with the implementation of two independently proposed extended transaction models for collaborative work (split/join [PKH88] and cooperative groups [MP92, RC92]). The ability to describe different

Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

extended transaction models in a common framework has been demonstrated previously in theoretical frameworks such as ACTA [CR90]. However, the practical implementation of such independently proposed extended transaction models in an industrial-grade transaction management system is new and significant.

The rest of this paper is organized as follows. We present the Reflective Transaction Framework in Section 2. Section 3 illustrates the flexibility of the reflective transaction framework through the example implementation of two extended transaction models. Section 4 introduces transaction adapters, describes their functionality, data structures, and their implementation on top of Encina. In Section 5 we discuss our implementation and review previous work and compare it with our approach. We conclude with a summary, and directions for future work.

2 Reflective Transaction Framework

Classic transactions are bracketed by the control operations `Begin-Transaction`, `Commit-Transaction` and `Abort-Transaction`, while extended transactions can invoke additional operations to control their execution, such as `Split-Transaction`, `Join-Transaction` or `Join-Group`. A particular transaction model defines both the control operations available to transactions that adhere to that model and the semantics of these operations. For example, whereas the `Commit-Transaction` operation of the standard transaction model implies the transaction is terminating successfully and that its effects on data objects should be made permanent in the database, the `Commit-Transaction` operation of a member transaction in a cooperative transaction group implies only that its effects on data objects be made persistent and visible to other member transactions. To capture this distinction, we first separate the programming interface of the transaction facility in order to keep the basic function of a transaction independent of the advanced operations required for extended transactions, and to control implementation level concerns.

2.1 A Separation of Interfaces

The Reflective Transaction Framework separates the programming interface to transactions into distinct levels, where each level presents a different view of transaction functionality. This separation follows the Open Implementation approach [Kic92], in which the *functional interface* is separated from the *meta interface*, and the purpose of the meta interface is to modify the behavior of the functional interface. In our separation of interfaces, presented below, Level 1 and Level 2 are functional, subdivided for clarity only. Level 3 is the meta interface that modifies the semantics of the transaction functional interface (Levels 1 and 2).

Level 1 The transaction demarcation interface: `begin-E-transaction`, `commit-E-transaction`, and `abort-E-transaction`. The addition of letter `E` in front of `transaction` indicates that these operations extend transaction semantics beyond ACID.

Level 2 The extended transaction interface (operations defined by each extended transaction model):

- For the split/join transaction model, it is `Split-Transaction` and `Join-Transaction`.
- For the cooperative group transaction model, it is `Begin-Group`, `Join-Group`, `Commit-Group`, and `Abort-Group`.

Level 3 The meta-transaction interface: extends the implementation of the TP monitor to support the extended transaction interface (Level 2). For the extended transaction models considered in this paper, the operations needed are: `instantiate`, `reflect`, `delegateOp`, `delegateLock`, `formDependency`, and `noConflict`.

The *transaction demarcation interface* (Level 1) exports the basic transaction interface. When used alone (Level 2 and Level 3 not involved) it provides classic ACID transaction semantics. The *extended transaction interface* (Level 2) exports a model-specific transaction interface when extended transaction functionality and semantics are required. Finally, the *meta-transaction interface* (Level 3) exports a modifiable interface to the underlying transaction processing facility for implementing extended transaction models.

This separation of the programming interface to the transaction processing system defines an extensible framework which can be used to develop applications requiring extended transactions and to implement extended transaction models. That is, a *TP system programmer* can implement extended transaction models by using the extended transaction interface to introduce new transaction control operations, and specify their implementation using the meta-transaction interface. An *application programmer* can then use both the transaction demarcation interface and the extended transaction interface to develop transactional applications. Reflection [Mac87] plays a crucial role in the reflective transaction framework, making it possible to open up the transaction processing system's functionality without revealing unnecessary implementation details. The meta-transaction interface makes reflection practical to use, by enabling the TP systems programmer to extend the underlying transaction processing system's behavior and implementation incrementally.

2.2 Metatransactions

One example of how reflection is applied in the reflective transaction framework is *metatransactions*. Metatransactions provide an extensible implementation of transactions that can be used to realize extended transactions. For example, using a metatransaction, one can redefine the general behavior of a transaction: how it handles conflicts, what control operations are available to the transaction, what happens at commit and abort time, etc. In our framework, each extended transaction, referred to as an *E-transaction*, is causally connected with a metatransaction. Metatransactions can be manipulated in the same manner as "normal" transactions, but more importantly, changes made to a metatransaction through the meta-transaction interface will be automatically reflected to the E-transaction. This enables the modification and extension of the behavior of the associated E-transaction.

The association of an extended transaction and a metatransaction is made when the E-transaction is *instantiated*, and ACID properties are initially assigned to the E-transaction by default. After instantiating an E-transaction, the application can add extended transaction control operations and semantics to its metatransaction at runtime using the `reflect` meta-transaction operation. For example, an alternative definition of the `Commit-Transaction` operation can be assigned to an E-transaction through its metatransaction. This makes it possible to adjust the computational behavior of an E-transaction to meet the needs of a particular application without modifying the underlying implementation of the transaction facility, but rather by changing the E-transaction's metatransaction.

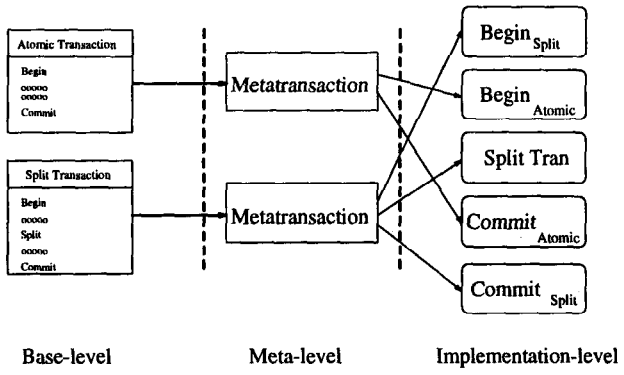

Figure 1: Transaction/Metatransaction Separation

When an E-transaction invokes a control operation, such as `commit-Transaction`, the call is trapped and handled at the meta-level by the metatransaction(see Figure 1). Thus, the operation implementation specified by the metatransaction, instead of the default implementation embedded in the TP system, is used to execute the invoked operation. Usually, the metatransaction will simply invoke the operation from the meta-level, but it may perform some extra processing before or after calling the implementation-level operation and perhaps not even call the operation at all. At the end of the operation execution, any results are returned to the E-transaction exactly like a normal transaction control operation call. As illustrated, an E-transaction's execution semantics are cooperatively provided by the control operations assigned to its corresponding metatransaction, and by semantic properties of these operations which have been fixed by the meta-transaction interface. In this sense, a suitable grouping of E-transaction control operations form an extended transaction model.

3 Realizing Extended Transaction Models

In this section, we demonstrate the use of the meta-transaction interface and the flexibility of the reflective transaction framework by describing the implementation of two independently proposed extended transaction models. Our objective is to identify functional components required to implement extended transaction models, so that we can then proceed to extend the underlying transaction processing facilities via transaction adapters. As such, the many variations that exist on the split/join and cooperative group transaction models were considered outside the scope of this paper.

3.1 Split/Join Transaction Model

In the split/join transaction model [PKH88] it is possible for an E-transaction to split into two serializable E-transactions or join another E-transaction. E-transactions in the split/join model are associated with five transaction management operations: `Begin`, `Split`, `Join`, `Abort`, and `Commit`. The `Begin`, `Abort`, and `Commit` operations have the same semantics as the corresponding operations of the atomic transaction. We will focus the remainder of our discussion of the split/join model on the definition and implementation of the extended `Split` and `Join` operations.

When an E-transaction T_1 splits, by executing the transaction management operation `split`(T_2), it must first create a new E-transaction (T_2) and then delegate responsibility for executing some of its operations to this new E-transaction. To be more precise, T_1 transfers to T_2 responsibility for all uncommitted operations on a particular set of data objects, referred to as the *DelegateSet*. In practice, users define the DelegateSet by selecting the objects to split from the re-structured E-transaction. At the time of the split, a new E-transaction is created, instantiated, and then operations invoked on objects in the DelegateSet by T_1 are

delegated to T_2. E-transactions T_1 and T_2 can then commit or abort independently. Split E-transactions can further split, creating new E-transactions. Here, it is interesting to note, that the split operation provides users with a mechanism to release data objects that are no longer needed by an E-transaction and to release intermediate results. The **split** operation is synthesized as follows:

```
E_splitOperation{
  // instantiate new transaction.
  instantiate(T2);
  // add transaction semantics through reflection.
  reflect(T2, sj_model);
  // delegate locks related to objects in DelegateSet.
  delegate_lock(T2, DelegateSet);
  // delegate ops related to objects in DelegateSet.
  delegate_op(T2, DelegateSet);
  // begin execution of the new transaction.
  begin(T2);
  // return control to invoking transaction
  return; }
```

The join transaction operation is the inverse of a split transaction operation. When E-transaction T_1 executes the transaction management operation **join**(T_2), it must delegate its uncommitted operations and associated locks to T_2 and then terminate its execution; E-transaction T_2 must already exist and be instantiated. E-transaction T_2 is now responsible for committing or aborting these operations, and the updates of T_2 must be committed together with the effects of T_1. In joining an E-transaction, the DelegateSet is simply all uncommitted operations and associated locks. In this regard, a joining transaction behaves similar to a child transaction in the nested transaction model. We synthesize the **join** operation as follows:

```
E_joinOperation{
 // delegate locks related to objects in DelegateSet.
 delegate_lock(T2, DelegateSet);
 // delegate ops related to objects in DelegateSet.
 delegate_op(T2, DelegateSet);
 // terminate execution of T1.
 commit(T1);
 // return control to invoking transaction.
 return; }
```

3.2 Cooperative Group Transaction Model

In the cooperative group transaction model [MP92, RC92], individual transactions may join a transaction group designed to facilitate cooperative access to a set of data objects. The cooperative group model supports two types of transactions, namely, *group transactions* and *member transactions*, each having its own set of transaction management operations. Only a group transaction can create a cooperative transaction group and it is then responsible for committing or aborting the results of transactions that are members of the group. Member transactions can join a specific cooperative group and share access to all data objects held within that group, while executing atomically with respect to the group. Member transactions can abort independently without causing the abort of the whole group, but only when the group transaction commits are the effects of the member transactions made permanent.

A group transaction in the cooperative group transaction model is associated with the following three unique transaction control operations: **beginGroup**, **commitGroup**, and **abortGroup**. A member transaction is associated with the following four transaction control operations: **Begin**, **Commit**, **Abort**, and **joinGroup**, where only the **Begin** operation has the same semantics as the corresponding operation of the atomic transaction. When a member transaction **commits**, all locks on data objects acquired by the transaction are delegated to the group transaction, as is the responsibility to make the effects on data objects permanent in the database when the group transaction commits. In this sense, the member transaction is commit-dependent on the group transaction and it only *pseudo-commits* its results when it **commits**. When a member transaction **aborts**, all locks on data objects acquired by the transaction are delegated to the group transaction and the transaction's effects on data objects are discarded. Now, we will synthesize the extended transaction management operations for the transactions in the cooperative group transaction model.

```
E_joinGroupOperation(GID){
 set group = GID
 // Ti can not commit until the group commits.
 create_dependency(Commit, Ti, GID);
 // Ti is abort dependent on the group.
 create_dependency(Abort, GID, Ti);
 // Ti permits group access to the locks it holds.
 no_conflict(Ti, GID);
 // begin execution.
 begin(Ti);
 // return control to invoking transaction.
 return; }

E_commitMemberOperation{
 // delegate locks to the cooperative group.
 delegate_locks(group);
 // group is responsible for committing operations.
 delegate_ops(group);
 // wait until group terminates.
 commit(self);
 // return control to invoking transaction.
 return; }

E_abortMemberOperation{
 // delegate locks to the cooperative group
 delegate_locks(group);
 // terminate execution
 abort(self);
 // return control to invoking transaction.
 return; }
```

These examples served to demonstrate how the meta-transaction interface can be used to implement two independently proposed extended transaction models. We describe the implementation of other extended transaction models in the full version of this conference paper [BP95]. Through the introduction of metatransactions we have enabled transactions to exhibit different extended semantics simply by binding to different extended transaction control operations. This binding can be done dynamically at run time using the meta-transaction interface. Metatransactions are realized by *transaction adapters*, and in the following section we introduce transaction adapters and describe their implementation in Encina.

4 Details of Implementation Method

In this section, we first present an overview of transaction adapters and then discuss the motivation and design strategy behind our transaction adapters implementation. Next, we describe the structure of the reflective transaction framework in terms of layering transaction adapters over the TP Monitor Architecture. Finally, we introduce the individual transaction adapters, and for selected adapters we outline their implementation in Encina. For reasons of space, we must limit our description of transaction adapters, however more detailed descriptions of the reflective transaction framework and transaction adapters can be found in our other papers [BP95, BPZH95].

4.1 Overview of Transaction Adapters

Transaction adapters are add-on modules built on top of existing commercial TP components to extend their functionality in support of extended transaction features and semantics. Each transaction adapter provides a representation (or model) of the underlying transaction processing component for use by the meta-transaction interface, mechanisms for reasoning about and with such a representation, and a set of commands for controlling both the representation and the underlying transaction facility. This set of commands is referred to as **TRACS**, for **TR**ansaction **A**dapter **C**ommand **S**et. TRACS expose features such as operation and lock delegation, dependency tracking between transactions, and relaxed definitions of conflict, as explicit commands by which extended transaction models can be implemented. Thus, instead of applying operations in the meta-transaction interface directly to the underlying transaction system, we base them on an abstract and enhanced description of the underlying transaction system provided by transaction adapters.

Our principal goal in designing transaction adapters was to build on top of existing TP monitor software to take advantage of existing transaction services to the extent possible. Although in retrospect this would seem to be the logical approach, it was not at all obvious that this was feasible because, in general, TP monitor components are tightly tuned for traditional transactions with ACID properties.

To reveal the design of the transaction adapters, we followed three simple design steps. The first step was to analyze extended transaction models to identify the required modular functional components. Part of this first step was summarized in Section 2. The second step, presented in Section 4.2, was to analyze the TP Monitor Architecture to identify the main modules that provide basic functionality required for extended transaction models. After mapping the required extended transaction model components into the existing TP monitor modules, the functionality identified as missing is exactly what needed to be provided by transaction adapters. The third, and final step, was to expose the new extended transaction functionality through a small number of commands which constitute the transaction adapters. In many cases, the functionality required for extended transactions was provided directly by the underlying TP facility or easily constructed, but in certain cases new data structures and functions had to be provided by the appropriate transaction adapter. As such, transaction adapters expose not only new extended functionality but also certain aspects of the TP monitor implementation, allowing users to adjust the implementation to better suit their needs.

4.2 TP Monitor Architecture

In order to discuss the implementation of our transaction adapters, we first need to establish a common basis for the transaction processing mechanisms involved. For this purpose we have chosen the standard transaction processing monitor architecture, introduced in Bernstein [Ber90] and detailed in Gray [GR93], which we abbreviate as the "TP Monitor Architecture". The TP Monitor Architecture is abstract enough to allow observations on TP systems in general, and yet concrete enough to make implementation details obvious in a modern TP monitor, such as Transarc's Encina or Novell's Tuxedo. We share the same assumptions made by the TP Monitor Architecture such as two-phase locking (2PL) concurrency control and write-ahead logging recovery. These assumptions are prevalent in existing TP monitors and many database systems.

The major functions of the TP Monitor Architecture, with respect to the implementation of E-transactions, are the execution of transaction management operations that control the transaction and concurrency control. Hence, we focus our description on the interaction among four components: a *Transaction Manager*, a *Lock Manager*, a *Log Manager* and a *Resource Manager*

(e.g., DBMS). The relationships among a transactional application and these four components are depicted in Figure 2. In a commercial setting, we might find a TP system such as Transarc Encina or Novell Tuxedo providing access to various resource managers, such as an Oracle or Informix relational DBMS.

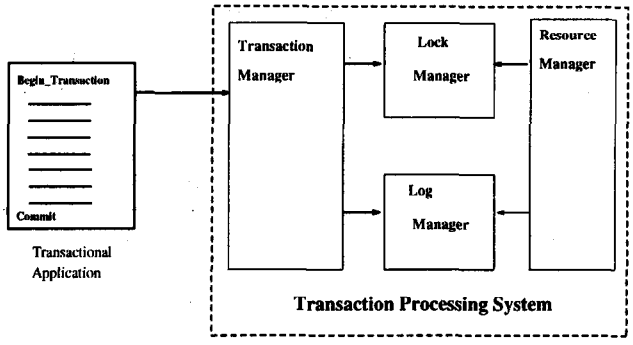

Figure 2: Components of a TP System

In the TP Monitor Architecture, ACID transactions are initiated by a **Begin-Transaction** call and terminated by either a **Commit-Transaction** or an **Abort-Transaction** call. When initiated, each transaction is assigned a unique identifier and entered into a transaction table managed by the Transaction Manager. Each entry in the transaction table contains the transaction identifier (TRID), the transaction status, and other information. When a transaction calls a transaction control operation, such as **Commit-Transaction**, the Transaction Manager is responsible for carrying out the execution of the command and recording information in the transaction table. However, the Transaction Manager in the TP Monitor Architecture does not allow a transaction to redefine the implementation of transaction management operations, nor does it allow the transaction to extend the set of transaction control operations available to it. And, while information on all active transactions is available to the Transaction Manager, it does not allow transactions to dynamically restructure their operations or locks, nor does it provide support for intertransaction dependency management. Thus, we introduce a *transaction management adapter* to extend the functionality of the Transaction Manager and expose its underlying services to implement extended transactions.

The Lock Manager maintains a lock table which contains an entry for every data item on which a lock has been requested (each request corresponds to an operation). Two functions, **Lock** and **Unlock**, are supported as the interface to the Lock Manager. The Lock Manager only detects basic conflicts between operations and does not consult any other source of information to determine if a potential conflict can be relaxed. Thus, our work is to extend this functionality to support the semantics of E-transactions and their relaxed notions of conflict. In addition, E-transactions often cooperate by sharing access to data objects, allowing the effects of their operations to be visible without producing conflicts, and by delegating locks to one another. The Lock Manager does not directly support this functionality, so we introduce a *lock adapter* and a *conflict adapter* to expose and extend the functionality of the Lock Manager.

4.3 Transaction Management Adapter

The *Transaction Management Adapter* is responsible for directing the execution of extended transaction control operations during the life of an E-transaction. The execution of an E-transaction consists of four steps: *instantiation*, *reflection*, *execution*, and *termination*. An E-transaction is entered into the reflective transaction framework through instantiation. After an instantiated E-transaction has been assigned a set of extended transaction control operations (semantics) through reflection it is said to be *ready* [1]. The E-transaction is now prepared for execution by the TP facility. An E-transaction is said to be *active* if it is executing operations but has not yet completed. An E-transaction is said to have *completed* if it has finished executing operations but is waiting to commit or abort, and considered *terminated* after it has been *committed* or *aborted*.

4.3.1 Design of the Transaction Management Adapter

Transaction management control operations are organized in the metatransaction descriptor under named categories, to enable the transaction management adapter to recognize the role each operations plays in the execution of an E-transaction. For example, the following metadescriptor fragment describes the control operations available to a split/join-transaction.

```
metatransaction Descriptor{
  myid is TRID;
  execMode is Active;
  initiateOperations: {<Begin,atomicBegin>}
  processOperations: {<Split,splitOperation>}
  terminateOperations: {<Commit,atomicCommit>,
                        <Abort,atomicAbort>,
                        <Join,joinOperation>}}
```

Here, **Descriptor** describes the metatransaction properties of an E-transaction in which the slots are defined as follows. The **initiateOperations** slot lists all control operations that can be called to initiate the execution of the E-transaction, the **processOperations** slot lists control operations that can be called during

[1] In contrast, when an ACID transaction is initiated by the TP system it enters the *ready* state

execution, and the **terminateOperations** slot lists control operations that will result in the termination of the E-transaction.

When an extended transaction control operation is invoked by an E-transaction, the actual code executed is determined by its metatransaction (see Figure 3). Suppose that a **split** operation is invoked by an E-transaction. Processing involves first verifying this E-transaction control operation is permitted for the E-transaction. This check is performed by simply checking the metatransaction descriptor to verify that **split** is listed in the extended transaction control operation set. In addition, *preTest* and *postTest invariants* can be defined for each extended transaction control operation. These invariants represent predicates that have to hold at the beginning and the end of the execution of the operation, respectively. Thus, if the operation name is found in the metatransaction descriptor and the preTest invariant predicate is satisfied, then the function is executed. Afterwards, the postTest invariant is evaluated, and then any results from the transaction control operation are returned to the E-transaction as if for a normal operation call.

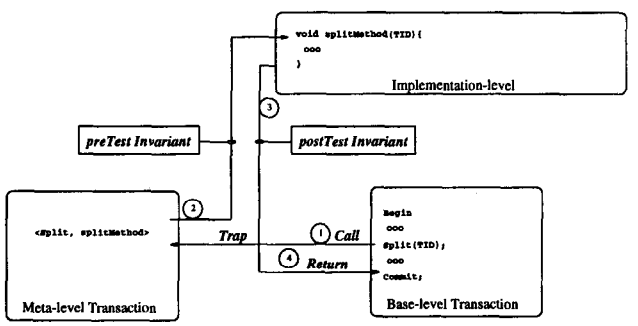

Figure 3: E-transaction control operation redirection

During the course of its execution, an E-transaction may form a dependency with another E-transaction, such as a commit or an abort dependency. The transaction management adapter provides commands to record and track transaction dependencies in support of extended transactions, such as the pseudo commit of cooperative group transactions. In addition, the transaction management adapter provides commands for restructuring an E-transaction through operation delegation. Before discussing the Encina implementation of the transaction management adapter, we summarize commands in its TRACS.

- **instantiate(TRID)**: Create a metatransaction descriptor for the E-transaction whose transaction identifier is TRID. If successful, **instantiate** returns a *reflective transaction identifier* (RID); otherwise it returns an error. The E-transaction does not start executing—execution is started by calling **exec**.
- **reflect(RID,semantics)**: Assigns a set of transaction management operations to the metadescriptor of the E-transaction whose reflective identifier is RID. For the purposes of this paper, the semantics field will be one of SJ – split/join model, CG – cooperative group model, or SJCG – split/join cooperative group model.
- **exec(TRID)**: Start execution of the E-transaction whose transaction identifier is TRID.
- **delegate_ops(TRID,opSet)**: Delegate all uncommitted operations listed in the set **opSet** to the E-transaction whose identifier is TRID.
- **form_dependency(type,$TRID_i$,$TRID_j$,opName)**: form a dependency of the specified *type* between $TRID_i$ and $TRID_j$. Many types of dependencies can be recorded but the only two required for the extended transaction models examined in this paper are CD – commit dependency and AD – abort dependency.

4.3.2 Encina Implementation of the Transaction Management Adapter

The Encina transaction manager assigns a unique transaction identifier (TRID) to each E-transaction when it is *initiated*, creating an entry in the transaction table to record the TRID with other pertinent information, and tracks the E-transaction through its execution. For E-transactions we augment the information stored in the transaction table using a Reflective Transaction Table (RT). An entry is created in the RT when the E-transaction is *instantiated*, and is used to store information relevant to the management of the E-transaction. While every transaction in Encina will have an entry in the transaction table, only E-transactions have an entry in the reflective transaction table. Data stored in both the transaction table entry and the reflective transaction table entry permit bidirectional access to the information stored in these tables.

Though each E-transaction will define its own function for processing **commit** or **abort** transaction management operations, the Transaction Management Adapter must provide commit and abort preprocessing to manage transaction dependencies. Thus, we define two functions, **PreCommit** and **PreAbort**, and register each with the appropriate callback function [ETPR]. The definitions of these new functions are outlined below:

PreCommit(T_i) Function Execution Steps:

1. Scan the list of dependencies emanating from T_i and for each such dependency D between T_i and some E-transaction T_j do the following:
   - Abort Dependency – if D is an abort dependency, then T_i cannot commit because if T_j aborts then T_i must abort as well. T_i blocks and retries later.
   - Commit Dependency – if D is a commit dependency T_i can only commit after T_j completes (either commits or aborts). T_i blocks and retries later.

2. At this point, T_i does not depend on any other E-transaction. Commit preprocessing is complete and control can be returned to the associated commit procedure. Change the status of T_i to *terminated* in the RT.

3. Scan the list of dependencies pertaining to T_i and remove each such edge from the dependency graph. This will effectively remove all dependencies of other E-transactions on T_i.

PreAbort(T_i) Function Execution Steps:

1. Scan the list of dependencies impinging on T_i and for each such dependency D between some E-transaction T_j and T_i do the following:
   - Abort Dependency – if D is an abort dependency, then T_j must also abort. Invoke the abort procedure on T_j.
   - Commit Dependency – if D is a commit dependency, then simply remove this dependency.
2. At this point T_i does not depend on any other E-transaction. Abort preprocessing is complete and control can be returned to the associated commit procedure. Change the status of T_i to *terminated* in the RT and return *success*.
3. Scan the list of dependencies pertaining to T_i and remove each such edge from the dependency graph. This will effectively remove all dependencies of other E-transactions on T_i.

To summarize the Encina implementation of the transaction management adapter, the data structures and implementation of each command in the TRACS are described below:

Reflective transaction table (RT): Each entry in the RT is assigned a unique identifier, or reflective transaction identifier (RID), and contains the following information:

- TRID: transaction identifier of E-transaction T.
- Group: TRID of $T's$ parent or cooperative group, in any.
- Status: the operational status of the E-transaction, which will be one of the following: *instantiated, ready, active, completed,* and *terminated.*
- Transaction Management Operations: In the form of a property list of *(operationName, function)* pairs. Property values are retrieved by using the operation name, providing the address of the function that is to be executed by the E-transaction.

The reflective transaction table is created by placing the reflective descriptors in a chained hash table based on the transaction's TRID.

The transaction dependencies graph (TRAND): This is a directed graph where the nodes in TRAND represent E-transactions and an edge from node T_i to T_j labeled with type represents a dependency of *type* between E-transaction T_i and E-transaction T_j. TRAND is composed of node structures that contain the following information:

- TRID_{from}
- RID_{from}
- dependency list of (*type*, TRID_{to}, RID_{to}, opName)

Nodes in TRAND include pointers into both the transaction table and the reflective transaction table for both E-transactions involved in the dependency, as well as information on the type of dependency. The node and edge data structures composing TRAND are doubly hashed on the TRID of the two E-transactions involved so that dependencies emanating from or impinging on an E-transaction can be located efficiently.

The Encina implementation of commands in the Transaction Management TRACS are described below:

`instantiate(TRID)`: Create a reflective transaction descriptor (RD) for the E-transaction and generate a reflective identifier (RID), storing the RD in the reflective transaction table (RT). Create an entry in the transaction table (TT) for the new E-transaction and generate a transaction identifier (TRID). Record the TRID in the RT entry corresponding to the E-transaction, and create a property list for the E-transaction using the command **tran_PropertyAdd (ReflectiveID, RID)** to record the RID. Register preabort and precommit callback functions for dependency management using the functions **callbackBeforeAbort(preAbort)** and **callbackBeforePrepare(preCommit)**, and set the status of the E-transaction in the RT to *instantiated.*

`reflect(RID, context)`: The transaction management operations associated with the named *context*, such as Split-Join or Cooperative Group, are assigned to the E-transaction. This assignment takes the form of a property list of (*operation-name, function*) pairs. For each transaction management operation associated with *context* a property pair is created and the memory addresses of the function associated with the operation is registered. The function (address) associated with the transaction management operation will be executed by this E-transaction when the operation is called. Set the status of the E-transaction in the RT to *ready.*

`form_dependency(type, Ti, Tj, opName)` : Insert a new edge in TRAND. Before this new edge is added to the graph a check is performed to prevent a dependency cycle from being created. If successful, that is, no cycles were detected by the addition of this new edge in TRAND, the function returns `success`; otherwise it returns `fail`.

`remove_dependency(type, Ti, Tj,opName)` : Removes an edge from the dependency graph TRAND.

`delegate_ops(Tj,DelegateSet)` : Transfer uncommitted operations listed in DelegateSet from T_i to T_j, and adjust dependencies in the TRAND graph accordingly.

4.4 Conflict Adapter

Depending on the semantics of an E-transaction and its relationship to other E-transactions, not all conflicts between E-transactions need to produce dependencies or serialization orderings. To capture this, the *conflict adapter* can selectively present and change the definition of conflict for one or more underlying data objects or E-transactions. By adapting the definition of conflict offered by the underlying TP system, the conflict adapter is able to provide support for a variety of extended transaction models and semantics-based concurrency control protocols [BPZH95].

The conflict adapter relaxes conflicts between E-transactions by two means: a compatibility table defining conflict relationships between operations, and a no-conflict table that records all conflicts explicitly *relaxed* between E-transactions. Based on these two sources of information, the conflict adapter uses the following rule to determine whether there is a conflict between two E-transactions:

Definition 1 ***(Relaxed Conflict Rule):*** A conflict detected by the basic conflict detection mechanism can be relaxed if either of the following conditions hold true:

1. the semantics of the data object indicate that the operation for which the lock is being requested is compatible with all uncommitted operations holding a lock in an incompatible mode;
2. the E-transaction holding the lock on the data object has explicitly indicated that the E-transaction requesting the lock has permission to perform the operation, regardless of the basic conflict;

Thus, the relaxed conflict rule used by the conflict adapter states that an E-transaction may acquire a lock if all other E-transactions owning the lock in a mode incompatible with T are relaxed by either operation semantics or explicit agreement between the E-transactions. The generality of the relaxed conflict rule allows the conflict adapter to capture many semantics-based concurrency control protocols discussed in the literature [BPZH95], and combine them with extended transactions models.

If an incompatible lock request is granted to an E-transaction T_i because of a relaxed conflict, a dependency $T_i \rightarrow T_j$ may be created for each E-transaction such that T_j owns a lock in a mode incompatible with T_i. The *type* of dependency formed between E-transaction T_i and T_j will be provided by either the operation compatibility table or the no conflict table, and recorded in the dependency graph using the transaction management adapter command **form_dependency**(*type*,T_i,T_j,*opName*). These dependency relationships will be tracked across both operation and lock delegation by the transaction management adapter. For example, if there is a commit dependency $T_i \rightarrow_{CD} T_j$ and T_i delegates its locks to T_k through a **join** operation, then the dependency is updated to $T_k \rightarrow_{CD} T_j$.

For the extended transaction models considered in this paper, we have utilized only one command in the conflict adapter TRACS, namely **noConflict**:

noconflict(T$_j$,[dataObjects]): when issued by E-transaction T_i it indicates that even if T_i has a data object in the set [dataObject(s)] locked in a mode that normally conflicts with T_j, T_j can still perform operations on the data object as far as T_i is concerned. If the list of data objects is empty, then T_i permits T_j access to any data object on which it holds a lock.

A complete description of the conflict adapter and its associated TRACS, along with a description of its Encina implementation is described elsewhere [BPZH95].

4.5 Lock Adapter

Locks on data objects can restrict the ability of a transaction to see the effects of other transactions on data objects while they are executing. The lock adapter allows greater control over the visibility of data objects by enabling an E-transaction to grant other E-transactions access to data objects on which they hold locks. The lock adapter enables an E-transaction to delegate ownership of its locks to another E-transaction prior to termination through the **delegate_lock** command. The **delegate_lock** command allows the E-transaction to specify whether it wishes to delegate all the locks it currently holds or only those for specified data objects. The conflict adapter records access rights granted between E-transactions in the no-conflict table, while the lock adapter provides the E-transactions access to the locked data object(s). It is the responsibility of the transaction programmer to guard against unwanted non-serializable behavior when using this feature.

The Lock Adapter provides this enhanced access to the lock table through commands which extend the functionality of the underlying lock service. Specifically, it provides E-transactions the ability to both delegate locks and share access to locked data objects. The principal command supported by the lock adapter TRACS is **delegate_lock**:

delegate_lock(T$_j$,[dataObjects]) when issued by E-transaction T_i, it releases all locks that T_i owns on the data objects listed in the set dataObjects and transfers ownership to E-transaction T_j. If the field listing the data objects is empty then this corresponds to all locks that T_i holds. The lock adapter supports additional options for specifying what locks are to be delegated, which are described elsewhere [BPZH95].

5 Discussion

In this section, we first discuss aspects of layering our implementation of the reflective transaction framework on the Encina TP Monitor. We then attempt to place the contribution of our work in perspective, and to clarify its relationship to work similar in spirit to ours.

5.1 Extending the Encina TP Monitor

TP monitors provide a general framework for transaction processing, supplying the "glue" to bind the many software components of a TP system through services like multithreaded processes, interprocess communication, queue management, and system management [Ber90]. While early TP monitors were constructed from tightly integrated product-specific services, modern TP monitors, such as Transarc's Encina [Encina], are layered on modular transaction middleware services. These transaction middleware services provide the basic building blocks for many of the features that a TP monitor must provide, and have a wide variety of uses. For example, IBM recently built a new implementation of its CICS TP monitor layered on transaction middleware developed by Transarc, the same middleware used in the Encina TP monitor.

The transaction services for the Encina TPM are provided by the Encina Toolkit, which is composed of separate transaction middleware service modules. Each module provides its transaction services through a relatively simple and uniform application programming interface (API). The core transaction services of the Toolkit are provided by the following modules: *Transaction Service Module(TRAN)* provides transaction demarcation (begin, commit, abort), distributed two-phase commit management, and nested transactions; *Lock Service Module (LOCK)* provides a logical locking package that guarantees serializability; *Log Service Module (LOG)* provides write-ahead log support for transaction updates, archiving, and crash recovery; *Transactional RPC Module (TRPC)* extends DCE remote procedure call facility (RPC) to have exactly-once transaction semantics. Together, these modules provide the basic building blocks for the services of the standard TP monitor architecture [GR93, pp. 21].

It was a basic tenet that our reflective transaction framework should be built as a relatively thin layer over the transaction middleware services provided by the Transarc Toolkit. In our design, transaction adapters are simply higher-level compositions of the transaction middleware services in order to realize extended transaction models. Specifically, the transaction management adapter extends the *TRAN* module to manage extended transactions, while the conflict and lock adapters extend the lock services provided by the *LOCK* module. In some cases, the Toolkit module functionality is directly exposed by the adapters. For example, the lock adapter uses the standard API to *LOCK* to release and acquire locks in support of delegation. In other cases, the Toolkit module functionality is essentially hidden and the necessary functionality is provided by the transaction adapter. For instance, *TRAN* callbacks are used to pass transaction specific information from Encina to the reflective transaction table in the transaction management adapter, and to provide a convenient point for performing transaction dependency checks.

In general, our experience with the Encina Toolkit was that it is well designed and provided a good foundation upon which to implement the reflective transaction framework. To date, we have only used the Toolkit as defined through the API and callback facility. The modularity and extensibility of the Toolkit, along with the functionality it provides have simplified our development effort and made it possible for us to focus on the design of the Reflective Transaction Framework.

5.2 Comparison With Related Research

Since the introduction of extended transaction models, research towards their realization has focused primarily on proposing specialized execution facilities, or extending programming and database languages with primitives for extended transactions. We refer the reader to Elmagarmid [Elm93] for a collection of recent work.

In contrast, the reflective transaction framework represents an evolutionary approach. Rather than attempting to develop a specialized transaction execution facility, the reflective transaction framework seeks to build support for extended transactions from transaction facilities which support classic transactions. From a purely pragmatic standpoint, our research differs from previous work in that we can use existing commercial products that runs across different platforms to implement different extended transaction models. Our research poses and answers the question "Is it possible to extend the TP monitor architecture in a practical and modular manner in order to realize extended transaction models". By doing so, it opens up the possibility of implementing a wide range of extended transaction models on industrial-grade transaction management systems where they can be applied in real-world applications.

The ability to specify different extended transaction models using a small set of modeling primitives was first demonstrated in the formal framework of ACTA [CR90]. In ACTA five simple building blocks are used to specify the essential components of extended transaction models, namely *history*, *inter-transaction dependencies*, *transaction conflict*, *transaction view*, and *delegation*. Even though ACTA was not intended to be executable, it provided us with valuable insight into the design of the reflective transaction framework. We distilled the

essence of selected ACTA building blocks into a functional realization for implementing extended transaction models. The result, in part, was the functionality provided by transaction adapters: the transaction management adapter includes transaction dependency management and operation delegation functions, the conflict adapter includes transaction conflict and transaction view funtions, while the lock adapter includes lock delegation functions. It is this close alliance with ACTA that enables the reflective transaction framework to implement a wide range of extended transaction models. On the design side, the meta-transaction interface and functionality of transaction adapters are close enough to ACTA to be applicable to many different extended transaction models. And on the implementation side, transaction adapters are close enough to the TP monitor architecture to support a practical implementation on top of commercial software.

An example of the language primitives approach is ASSET [BDG+94], in which the ability to implement extended transaction models is provided at a very low level by embedding ACTA-based primitives in the host language of an object-oriented database. Using ASSET, an application programmer can construct extended tranactions from scratch by properly composing the available languistic primitives. In contrast, TSME [GHKM94] represents the specialized transaction facility approach, in which the ability to implement extended transaction models is provided at a high level through a transaction specification language and mechanisms which configure the run-time transaction facility to realize extended extended transactions. Using TSME, an application programmer can construct certain extended transactions using certain expressions in the specification language, which are then mapped to certain *pre-built* configurations in the transaction management mechanism. In a sense, the reflective transaction framework represents a fusion of these two approaches. The meta-transaction interface provides the low-level flexibility of language primitives, enabling an application programmer to construct extended transactions from scratch, while the extended transaction interface and transaction adapters provide the high-level interface and functionality of a specialized transaction facility, enabling an application programmer to construct extended transaction from existing components.

Prototype implementations of special-purpose extended transaction models can be found in the literature. Some well known representatives include the APRICOTS system [Sch93] (A PRototype Implementation of a COnTract System [WR93]), the multi-level transaction model [WH93], and the Flex transaction model used in the InterBase project [BEK93]. Also related, an approach to implement extended transaction models using a commercial workflow manager has been suggested in [MAG+95], though the generality of this approach remains to be seen.

In a final note on related work, a feature which further distinguishes our work is the application of the Open Implementation approach. We found three tangible benefits in taking the Open Implementation approach in designing the reflective transaction framework. First, the separation of the functional and meta interface to the transaction processing facility allows programmers to adjust and extend the design and implementation of the system to suit their particular needs easily. Programmers can introduce new transaction control operations by simply defining new operations in the extended transaction interface, or they can redefine the semantics of an existing transaction control operation using the meta-transaction interface. Second, using metatransactions to deal with the wide range of different transaction models enables us to rely on an underlying transaction facility for the basic implementation. This not only simplified our development effort, but also enables us to evaluate extended transaction models on an industrial-grade transaction management system in real, working environments. Third, permitting each transaction to have its own metatransaction makes it possible for an application to assign different extended transaction semantics to different transactions according to the needs of their application.

6 Concluding Remarks

We have presented the reflective transaction framework as a practical and modular method to implement extended transaction models. We achieved modularity by applying the Open Implementation approach to design the reflective transaction framework, and we achieved practicality by extending commercial TP monitor software to implement the reflective transaction framework. Although the implementation details were product specific (Transarc's Encina), our framework was designed in the context of the TP Monitor Architecture, so it is applicable to many modern commercial TP monitors. Our early experience shows that the reflective transaction framework is general enough to implement a wide range of extended transaction models [BP95].

While the importance of extended transaction models has been known for many years, their use in real-world applications has been hampered by the lack of practical implementations. Furthermore, since most extended transaction models have been merely theoretical constructs, there are a number of important design issues that have generally not been discussed in the literature [Moh94]. Our hope is that the reflective transaction framework will remedy this situation, providing a clear migration path to incorporate research advances in

extended transaction models into commercial TP monitors. This will enable us to draw conclusions from direct experience in applying extended models in real, working environments.

We are proceeding with active research based on the Reflective Transaction Framework and transaction adapters. We plan to refine the reflective transaction framework, evaluate the performance of our Encina implementation for selected extended transaction models, and study the broader issues of the concurrent execution of different extended transaction models to better understand their interference and synchronization.

References

[BDG+94] A. Biliris, et al. Asset: A system for supporting extended transactions. In *Proceedings of 1994 ACM SIGMOD*, pages 44–53, May 1994.

[Ber90] Philip A. Bernstein. Transaction processing monitors. *CACM*, 33(11):75–86, 1990.

[BP95] R. S. Barga and C. Pu. A practical and modular implementation of extended transaction models. Technical Report OGI-CSE-95-004, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1995.

[BPZH95] R.S. Barga, C. Pu, T. Zhou, and W.W. Hseush. A practical method for implementing semantics-based concurrency control. Technical Report OGI-CSE-95, Department of Computer Science and Engineering, Oregon Graduate Institute, May 1995.

[BEK93] O. Bukhres, et al. Implementation of the Flex Transaction Model. *Bulletin of the IEEE Technical Committee on Data Engineering*, 16(2):28–32, June 1993.

[CR90] P.K. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of 1990 ACM SIGMOD*, pages 194–203, June 1990.

[Elm93] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1993.

[Encina] Transarc Corp. *Encina Product Overview*. Transarc Corp, Pittsburgh, PA., 1991.

[ETPR] Transarc Corp. *Encina Toolkit Server Core Programmer's Reference*. Transarc Corp, Pittsburgh, PA., 1991.

[GHKM94] D. Georgakopoulos, et al. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the 1994 IEEE Conference on Data Engineering*, pages 462–473, Feb 1994.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[KdRB91] G. Kiczales, et al. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA Workshop on Reflection and Meta-level Architectures*, 1992. See `http://www.xerox.com/PARC/spl/eca/oi.html` for updates.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1987.

[MAG+95] C. Mohan, et al. Exotica: A research Perspective on Workflow Management Systems. *Bulletin of the IEEE Technical Committee on Data Engineering*, pages 19–26, June 1995.

[Moh94] C. Mohan. Advanced transaction models — survey and critique. Tutorial Presented at the ACM SIGMOD International Conference on Management of Data, 1994.

[MP92] B. Martin and C. Pederson. Long-lived concurrent activities. In Amar Gupta, editor, *Distributed Object Management*, pages 188–206. Morgan Kaufmann, 1992.

[PC93] C. Pu and S.W. Chen. ACID properties need fast relief: Relaxing consistency using epsilon serializability. In *Proceedings of Fifth International Workshop on High Performance Transaction Systems*, Asilomar, California, Sept. 1993.

[PKH88] C. Pu, et al. Split-transactions for open-ended activities. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 27–36, Los Angeles, August 1988.

[RC92] K. Ramamritham and P.K. Chrysanthis. In search of acceptability criteria: Database consistency requirements and transaction correctness properties. In Amar Gupta, editor, *Distributed Object Management*, pages 212–230. Morgan Kaufmann, 1992.

[Reu82] A. Reuter. Concurrency on high traffic data elements. *ACM Principles of Database Systems*, 8(2):186–213, 1982.

[Sch93] F. Schwenkreis. APRICOTS - A prototype implementation of a ConTract System. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, Princeton, NJ., IEEE Computer Press, 1993.

[WR93] H. Wächter and A. Reuter. The ConTract Model. In [Elm93].

[WH93] G. Weikum and C. Hasse. Multi-level transaction management for complex objects: Implementation, performance, parallelism. In *VLDB Journal*, 2(4), 1993.