(Extended Abstract)

A Practical and Modular Implementation of Extended Transaction Models

Roger Barga
Calton Pu

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
P.O. Box 91000
Portland, OR 97291-1000

email: {barga,calton}@cse.ogi.edu

1 Introduction

The last five years have witnessed the introduction of numerous extended transaction models [Elm93]. These models 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 transactions. We present the *Reflective Transaction Framework*, a practical and modular framework which provides the basic features of extended transactions and can be used to implement a wide range of extended transaction models [BP95]. 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 basing the implementation of the reflective transaction framework on Gray and Reuter's architecture [GR93], which is widely applicable to many modern transaction processing (TP) systems.

Our proposed implementation introduces *transaction adapters*, add-on modules built on top of existing commercial TP components that extend their functionality to support extended transaction features and semantics. We further demonstrate practicality by incorporating transaction adapters within Encina, a commercial TP facility. This modular and practical design enables us to implement a wide range of extended transaction models, and we illustrate the implementation of two independently proposed extended transaction models for collaborative work (split/join model [PKH88] and cooperative groups [MP92, RC92]), and the synthesis of a new model from their combination [BP95].

2 The 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`. The semantics of a particular transaction model define 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 that the transaction is terminating successfully and all of 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 with respect to other member transactions. To capture this distinction requires a separation of programming interfaces to transactions in order to keep the notion of the basic function of a transaction independent of the advanced operations required for extended transactions, and also to be able to control implementation level concerns.

To design the reflective transaction framework, we first apply the Open Implementation approach [Kic92] to separate the programming interfaces to extended transactions, and identify modular functional components required to realize this separation. Next, we proceed to extend the underlying transaction processing facilities to support these modular functional components via transaction adapters. We outline these steps below, and their complete description is available in the full version of this paper [BP95].

2.1 A Separation of Interfaces

The Reflective Transaction Framework divides the programming interface to transactions into three groups. This division follows the Open Implementation approach [Kic92], which separates the *functional interface* from the *meta interface*, where the purpose of the meta interface is to modify the behavior of the functional interface. In our division presented below, Groups 1 and 2 are functional, subdivided for clarity only, while Group 3 is the meta interface that modifies the semantics of the transaction functional interface (Groups 1 and 2). The three groups are:

1. The transaction demarcation interface: `Begin-Transaction, Commit-Transaction`, and `Abort-Transaction`.
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`.
3. The meta-transaction interface: defined by the Reflective Transaction Framework, it extends the implementation of the TP monitor to support the extended transaction interface (Group 2. For the split/join transaction model and the cooperative group model, the operations needed are: `instantiate, reflect, delegateOp, delegateLock, formDependency`, and `noConflict`.

The *transaction demarcation interface* (Group 1) exports the basic transaction interface, and when used alone (Group 2 and 3 not involved) it provides classic ACID transaction semantics. The *extended transaction interface* (Group 2) exports a model-specific transaction interface for when enhanced transaction functionality and semantics are required. Finally, the *meta-transaction interface* (Group 3) exports a modifiable interface to the underlying transaction processing facility for implementing extended transaction models. This separation of the programming interface to transactions defines a framework which can be used to both develop advanced applications requiring extended transactions and to implement new extended transaction models. The transaction application developer codes transactional applications using both the transaction demarcation interface (Group 1) and the extended transaction interface (Group 2). In addition, the transaction systems programmer can define or synthesize new extended transaction models by using the extended transaction interface (Group 2) to specify transaction control operations, and control their implementation using the meta-transaction interface (Group 3).

Realization of extended transaction models is facilitated through the careful design of the meta-transaction interface and its implementation. The design of the meta-transaction interface was inspired by the ACTA framework and readers familiar with ACTA will recognize that it supports many of the ACTA basic building blocks for describing extended transaction models. On the design side, the meta-transaction interface is close enough to ACTA to obtain modularity and applicability to a wide range of extended transaction models. On the implementation side, it is close enough to the Gray and Reuter TP

architecture (GR Architecture) to support a practical implementation on top of commercial software. This design and implementation of the meta-transaction interface is captured in transaction adapters.

2.2 Implementation Through Transaction Adapters

Transaction adapters are modules built on top of an existing transaction processing facility that extend the underlying functionality. Each transaction adapter provides a representation (model) of the underlying transaction processing component for use by the meta-transaction interface (Group 3), 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.

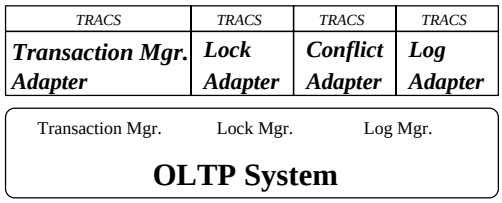

Figure 1: Transaction Adapters in the Reflective Transaction Framework.

Transaction adapters and their associated TRACS, as illustrated in Figure 1, are built on top of and use existing transaction processing services. For practicality we base their design on the GR Architecture, and are currently exploring their implementation via Transarc's Encina OLTP system. This way, TRACS provide similar reliability to mature OLTP systems software and minimize overheads often associated with increased flexibility. In fact, overhead associated with extended transactions is incurred only when the extended facilities provided by transaction adapters are used, and since an ACID transaction uses only the transaction demarcation interface (Group 1), it is executed without additional overhead.

2.3 Realizing Extended Transactions

In the Reflective Transaction Framework, a transaction is simply a partially ordered set operations on data objects, with calls to operations in any of the three groups of interfaces. Users are not limited to a predefined extended transaction model, which may or may not be appropriate for their application, nor is a transaction restricted to a single set of extended semantics and properties. Instead, a transaction declares its intention to use extended transaction properties and semantics via the `instantiate` command prior to beginning execution. The effect of this command is the creation of entries in the transaction adapters to represent this transaction, effectively creating a *metatransaction*. The meta-transaction command `reflect` assigns extended semantics to the metatransaction and supports the further refinement and extension of transaction properties during execution. For example, the sequence of transaction control operations presented below first creates a metatransaction for transaction T_i, assign it the semantics of the Split/Join transaction model [PKH88], and begins the execution the execution of transaction T_i.

```
instantiate(Ti);              // create metatransaction
reflect(Ti,Split-Join);       // assign split-join transaction semantics
begin-transaction(Ti);        // begin execution of extended transaction
```

A transaction with semantics beyond ACID properties is referred to as an *E-transaction*. When an E-transaction invokes a transaction control operation, the metatransaction is responsible for determining which function is actually executed based on the extended semantics of the transaction. Figure 2 illustrates the processing when the E-transaction T_i invokes the `Split-Transaction` control operation. Thus, from the system point of view, we consider the *metatransaction* as an object that supplies the extended transaction model semantics, separate from the application programmer's view of transactions.

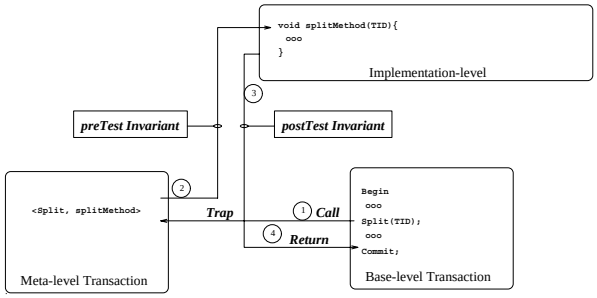

Figure 2: Transaction management method execution redirection.

The transaction systems programmer can define and specify the implementation of E-transaction operation by defining another metatransaction and then substituting it. In this sense, a suitable grouping of E-transaction operations form an extended transaction model. The sequence of metatransaction commands presented below specify the implementation of the `Split-Transaction` control operation. Metatransactions are thus realized through both the metatransaction interface and the transaction adapters and their associated TRACS.

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

3 An Encina Implementation

One of the salient features of design of the reflective transaction framework is its compatibility with the GR Architecture, which is widely applicable to many modern transaction processing systems. One major advantage of this compatibility is the ease for implementation of the reflective transaction framework. Instead of starting from scratch, we can extend an existing OLTP system through the definition and implementation of the transaction adapters. Concretely, we are currently implementing the Transaction Manager Adapter, Lock Adapter, and Conflict Adapter (Figure 1) on Encina, a commercial OLTP system distributed by Transarc. The full version of this paper [BP95] contains a complete description of the implementation details.

The goals of our implementation effort are: (1) to demonstrate the practicality of the reflective transaction framework and transaction adapters, (2) to evaluate extended transaction models in a real environment, (3) to determine how easy it is to implement a wide range of extended transaction models, and (4) to facilitate eventual technology transfer to real users.

4 Summary

We have proposed the Reflective Transaction Framework to analyze and describe the interface to extended transaction models. Early experience [BP95] shows that the framework is general enough for a wide range of extended transaction models, including split-transactions and cooperative groups. Using the framework, we outline a practical and modular implementation method for extended transaction models based on *transaction adapters*.

Our implementation method is practical because it builds on Gray and Reuter's architecture [GR93] and we have a concrete design on top of Transarc Encina. Our implementation is modular because independently proposed extended transaction models, such as split-transactions and cooperative groups, can be implemented individually and combined on the same transaction processing system.

Currently we are applying the framework and transaction adapters implementation method to other extended transaction models, including Epsilon Serializability [RP91]. At the same time, we are using Encina to both implement the designs outlined here and to implement extended transaction models, such as split-transactions, cooperative groups, and Epsilon Serializability.

References

[BP95] Roger S. Barga and Calton 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.

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

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

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *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'92 Workshop on Reflection and Meta-level Architectures*, 1992. See `http://www.xerox.com/PARC/spl/eca/oi.html` for updates.

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

[PKH88] C. Pu, G.E. Kaiser, and N. Hutchinson. 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 editor, editor, *Distributed Object Management*, pages 212–230. Morgan Kaufmann, 1992.

[RP91] K. Ramamrithan and C. Pu. A formal characterization of epsilon serializability. Technical Report CUCS-044-91, Department of Computer Science, Columbia University, 1991. To appear in IEEE Transaction on Knowledge and Data Engineering, June 1996.