A Reflective Framework for Implementing Extended Transactions

Roger S. Barga
B.S. Mathematics, Boise State University, 1985
M.S. Computer Science, University of Idaho, 1987

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

April 1999

The dissertation "A Reflective Framework for Implementing Extended Transactions" by Roger S. Barga has been examined and approved by the following Examination Committee:

Calton Pu
Professor
Thesis Research Adviser

Jonathan Walpole
Professor

Andrew Black
Professor

Phil Bernstein
Microsoft Corporation

Acknowledgements

First I would like to thank Calton. I could not have asked for a better advisor or mentor. He always made time to meet with me, despite his increasingly busy schedule, and he never failed to provide insightful feedback on my work. He was patient and encouraging when I struggled, and demanding when I needed to be challenged. He taught me that research is about asking the right questions and searching for answers that embody general principles or abstractions, and instilled in me the value of clearly expressing these questions and answers. Calton, working with you has been a privilege.

I would like to thank the members of my thesis committee, Phil Bernstein, Andrew Black, and Jon Walpole for their contributions to the ideas described here, and to the form of their presentation. Phil deserves especial mention for painstakingly reviewing various drafts of this document and providing detailed suggestions to improve the presentation.

Various students and faculty at OGI were an invaluable source of friendship, criticism, and advice to me over the years. In particular, I wish to acknowledge Crispin Cowan, Ashvin Goel, Jon Inouye, Buck Krasic, David Novick, Walid Taha, Jenny Orr, Bennet Vance, and Tong Zhou. They all deserve better recompense than a few words here. I'll get around to you all in time.

Finally, I wish to thank my parents, Jim and Cathy, whose unfaltering love and encouragement were an essential ingredient in the development of this thesis. Their *continual queries* as to whether or not "my paper was done yet" and, in general, their never-ending interest in my work and accomplishments, helped keep me oriented and on track.

Thank you all.

Contents

List of Tables

List of Figures

Abstract

A Reflective Framework for Implementing Extended Transactions

Roger S. Barga

Supervising Professor: Calton Pu

Databases are being deployed in more and more complex application domains to store and manipulate information that stresses the limits of the *performance* as well as *functionality* of traditional transaction processing techniques. In the past decade the topic of *extended transaction processing* has emerged and enormous strides have been made in improving the performance of traditional ACID transactions; at the same time, advances have been made in addressing their inherent limitations. Suggested extensions of ACID transactions abound in the literature. However, few of these extensions have ever been implemented, not even as research prototypes, and today most remain mere theoretical constructs.

In this dissertation we present the Reflective Transaction Framework to support the implementation of extended transactions on conventional TP monitor software. There are two key insights behind our work. The first is our observation that in most cases, the base functionality provided by TP monitor software is *"almost right"* for implementing extended transactions. While certain functions and structures are missing, the existing services of TP monitor software provide a useful substrate for implementing extended transactions. The second insight is that the services we have identified as essential for extended transactions can be implemented as extensions to base functionality of a TP monitor. To validate this thesis, we present the design of the Reflective Transaction

Framework, provide examples that illustrate how it can be used to implement extended transactions, describe its implementation on a commercial TP monitor, and present an evaluation of both framework design and resulting implementation.

This research is the first to demonstrate convincingly a method of extending conventional TP monitor software to support extended transactions, one that can readily implement a wide range of extended transactions. This research addresses three main issues in the implementation of extended transactions on a conventional transaction system. First, it identifies key extended services required to implement extended transactions. Second, it defines an effective interface to these extended transaction services and to the existing functionality provided by the underlying TP monitor. And third, it shows how to integrate these extended services with an existing transaction system in an extensible and incremental way.

Chapter 1

Introduction

Transactions have been used effectively in database systems to synchronize concurrent accesses to a shared database and to provide reliable access in the face of failures. A transaction is an atomic unit of work against the database. The ACID properties of transactions (atomicity, consistency, isolation, and durability) guarantee correct concurrent execution as well as reliability [HR83, BHG87, GR93].

In recent years, databases have been deployed in increasingly complex applications to store and manipulate information that stresses the limits of the *functionality* as well as the *performance* of traditional transaction processing techniques. The list of such applications includes computer-aided design and manufacturing (CAD/CAM) environments, multimedia, mobile computing, cooperative group software, and workflow management systems. This list is growing. Further, the ability of transactions to hide the effects of concurrency and failure makes them appropriate building blocks for structuring advanced distributed systems. Industry is embracing transactions, with a near explosion occurring in usage, requirements and sophistication of transaction processing [Moh94, SSU96]. Enormous strides have been made in improving the performance of traditional ACID transactions; at the same time, advances have been made in addressing their inherent limitations.

In the past decade the topic of *extended transaction processing*, also known as advanced or relaxed transaction processing, has emerged in the database community, to extend the transaction concept beyond conventional data processing and online transaction processing (OLTP) applications. Broadly speaking, recent accomplishments in extended transaction processing can be classified into two areas: *advanced transaction models* and *semantics-based concurrency control methods*. Advanced transaction models, such as the Split/Join model [PKH88] and Cooperative Transaction Groups [MP92], associate "broader" interpretations with the ACID properties to provide enhanced transaction processing functionality. Semantics-based concurrency control (SBCC) methods, such as commutativity [Wei88], recoverability [BR91] and cooperative serializability [RC92], exploit available semantic information to synchronize transactions in an attempt to obtain

additional concurrency and hence improve transaction processing performance. Suggested extensions of traditional ACID transactions abound in the literature. However, few of these extensions have ever been implemented, not even as research prototypes, and today most remain mere theoretical constructs.

This thesis rectifies this deficiency. Building on the functionality present in conventional transaction processing systems we focus on understanding the functionality required to implement extended transactions and define extended transaction services as modular extensions to conventional transaction processing structures and services. The software framework we present addresses three main issues in the implementation of extended transactions on a conventional transaction system. First, it identifies key extended transaction services required to implement a wide range of extended transactions in the literature. Second, it defines an effective interface to these new extended services and to the existing functionality provided by conventional transaction processing system software. And third, it shows how to integrate these extended services with an existing transaction system in an extensible and incremental way.

1.1 The Problem

Because of the practical import of advanced transaction models and semantics-based concurrency control protocols, one would expect their implementation to proceed apace. However, this has not happened. To date, the vast majority of the proposals for advanced transaction models and semantics-based concurrency control have remained just that — proposals. As a result, there is no way to readily apply these ideas to emerging database applications. Given that for many advanced applications extended transactions have been shown, on paper, to have the potential to improve transaction processing performance and functionality, we feel that the time has come to migrate these ideas into practice. Indeed, providing effective support for extended transactions has been identified as one of the key database research areas for the next century [SSU96].

Despite advances in advanced transaction models and semantics-based concurrency control over the past decade, the implementation of extended transactions remains difficult and expensive. Much of the effort and cost arises because researchers and application developers attempt to construct the extended transaction implementation from scratch rather than reusing conventional transaction processing software. This forces them to rediscover and reimplement core functions and components, which is time-consuming, error-prone, and expensive. However, to date this has been considered the only reasonable approach, as extended transactions would seem to require the replacement of conventional transaction services with new techniques and mechanisms for transaction processing.

Conventional transaction processing systems, in particular TP monitors, have accumulated large amounts of transaction implementation technology. We do not think it is particularly clever simply to throw this technology away and build an extended transaction facility from scratch. Indeed, TP monitors are *mission critical* – that is, they are essential to day-to-day business operations and must remain in use. Replacement of existing TP monitors is not an option for companies that rely on ACID transactions to run mission-critical applications. Moreover, advanced transaction models and semantics-based concurrency control protocols have largely been designed to complement conventional transaction processing and address an entirely new range of transaction requirements that would make their combination suitable for building advanced database applications. Consequently, we view conventional TP monitor software as a natural basis on which to build implementation support for extended transactions.

1.2 The Approach

In this dissertation we present the Reflective Transaction Framework to support the implementation of advanced transaction models and semantics-based concurrency control protocols on conventional TP monitor software. There are two key insights behind our work. The first is our observation that in most cases, the base functionality provided by a conventional TP monitor is *"almost right"* for implementing both advanced transaction models and semantics-based concurrency control protocols. While certain functions and data structures are missing, the existing services and data structures of the TP monitor software provide a useful substrate for implementing extended transactions. The second insight is that each of the extended services that we have identified as essential for implementing extended transactions can be implemented as an incremental extension to the base services of a TP monitor. This approach ensures that transactional applications using ACID transactions keep running, and facilitates the development of a software framework for implementing extended transactions in a systematic rather than an *ad hoc* manner.

We do not advocate that TP monitors should simply include more features to implement selected extended transactions. There is no consensus as to which extended transactions should be included for advanced application development: most likely, there never will be since each advanced transaction model and semantics-based concurrency control protocol has been optimized for a particular application. Furthermore, as application requirements continue to evolve, transaction processing requirements will change and *new* transaction models and semantics-based concurrency protocols will be proposed. Instead, the Reflective Transaction Framework is designed to expose selected aspects of the underlying transaction processing system and to enable a programmer to reach in and

adjust system functionality and tailor new extended transaction services to the needs of their particular application. This approach is called *open implementation* [Kic92].

The Reflective Transaction Framework draws from a variety of techniques to achieve the open implementation of a TP monitor. The framework uses *computational reflection* [Mae87] to offer principled, effective access to TP monitor system internals. A *metalevel interface* [KdRB91] is introduced in the framework to provide explicit descriptions of extended transaction behaviors. Good software engineering practices are followed for abstraction and modularity of the software modules that implement the framework.

The implementation of the Reflective Transaction Framework introduces *transaction adapters*, which are reflective software modules built on top of the TP monitor software. A transaction adapter leverages existing transaction services of the underlying TP monitor as building blocks for constructing extended transaction functionality. Each transaction adapter contains a *representation*, or metalevel description, of selected TP monitor functions, and maintains a *causal connection* [Mae87] between this representation and the actual behavior of the system. The causal connection is two-way; not only are changes in the TP monitor reflected in equivalent changes in the representation, but changes in the representation will also cause changes in the behavior of the TP monitor. Each extended transaction has a representation that is causally connected with a transaction running on the TP monitor. This representation holds information about the extended transaction and how it is used; in essence, this representation defines control and policy. The causal connection between the Reflective Transaction Framework and the underlying TP monitor is built on the ability to intercept transaction events, together with the means to access TP monitor functions through an available application programming interface (API).

1.3 The Thesis

The thesis is that conventional TP monitor software can be used to support the implementation of advanced transaction models and semantics based concurrency control protocols, through the provision of new extended services specifically designed for implementing extended transactions. These extended transaction services can be implemented efficiently as extensions of the functionality of the underlying TP monitor, and used to implement a wide range of advanced transaction models and semantics-based concurrency control protocols. The thesis claims that the ability to leverage, or reuse, the functionality of conventional TP monitor software more than makes up for the additional effort required in system design. To validate this thesis, we present the detailed design of the Reflective Transaction Framework, provide examples that illustrate how it can be used to implement

a number of extended transactions from the literature, describe its concrete implementation on a commercial TP monitor, and present an evaluation of both framework design and the resulting implementation.

1.4 Outline of the Dissertation

This dissertation is organized into five chapters. Chapter 2 provides the technical background for our work on the Reflective Transaction Framework. The chapter first sketches an overview of extended transaction processing and identifies functional extensions required to support the implementation of advanced transaction models and semantics-based concurrency control protocols. Following this, the chapter presents a review of related efforts to implement extended transactions, with particular emphasis on the range of extended transactions that they support. Then, an overview of the conventional TP monitor architecture is presented, along with a brief discussion on extending it to provide implementation support for extended transactions. The chapter concludes with an overview of the Open Implementation approach and a discussion of the development of an open implementation of a conventional TP monitor.

Chapter 3 presents the design of the Reflective Transaction Framework. The chapter begins with a discussion of our main design objectives, followed by an architectural overview of the Reflective Transaction Framework, and then presents a detailed description of the extended transaction services provided by the framework, specifically (1) dynamic transaction restructuring, (2) semantic transaction synchronization, and (3) transaction execution control. Where appropriate, we describe how the extended transaction services provided by the framework can be used to implement extended transactions, and we explain the relevant mechanisms from a user's perspective. Chapter 4 presents several examples of applying the Reflective Transaction Framework to advanced transaction models and semantics-based concurrency control protocols from the literature, to give a clearer overall picture of the framework and its uses.

Chapter 5 describes an implementation of the Reflective Transaction Framework on ENCINA, a commercial TP monitor, along with a performance evaluation of our Encina implementation. The evaluation measures the system resources consumed in supporting the extended transaction services and presents an evaluation of the framework design to augment the quantitative data. Chapter 6 concludes the dissertation with a summary of the main contributions of this research, and identifies opportunities for future work.

Chapter 2

Technical Background

In this chapter, we provide the technical background for our work on the Reflective Transaction Framework. We divide this chapter into three sections: extended transaction processing, conventional TP monitors, and Open Implementation.

In designing the Reflective Transaction Framework it was necessary to identify common services for advanced transaction models and semantics-based concurrency control that should be included in the framework. However, the lack of a general model of extended transactions hinders any meaningful discussion of the issues and approaches. Section 2.1 provides a basis by examining extended transactions from the literature. In doing so, key extended services required to implement extended transactions are identified in a natural way. We then present a brief review of related efforts to implement extended transactions and identify their main features, with particular emphasis on the range of extended services that they support. Finally, having identified key extended transaction services and reviewed related implementation efforts, we close the section by presenting our strategy for developing the Reflective Transaction Framework.

Following this, in Section 2.2, we present an overview of the conventional TP monitor architecture. Conventional transaction processing systems, such as TP monitors, have accumulated large amounts of transaction implementation technology that we would like to leverage in our implementation of the Reflective Transaction Framework. Thus, we examine the TP monitor architecture with an eye towards how we can leverage existing functions and incrementally extend available services to implement extended transactions.

Finally, in Section 2.3, we draw on work in computational reflection and Open Implementation to confront challenges that arise in designing a framework that builds on legacy TP monitor software and incrementally extend the existing functionality to implement extended transactions.

2.1 Extended Transaction Processing

In response to functionality and performance deficiencies of the traditional ACID transaction model, several new *extended transaction* proposals have emerged. Such proposals often start from a specific application, analyze its dynamic behavior, specify a fault model, and then add as many features to the classic ACID transaction model as necessary to support that application. Suggested extended transactions abound in the literature. In an informal literature survey, we identified over fifty distinct extended transaction types, with new proposals appearing in the database literature at an average rate of six per year.

Because this is an area of active research, this section can do little more than give an overview of the current state of discussion. As pointed out in Gray and Reuter [GR93], no *Grand Unified Theory of Extended Transactions* has yet been developed. To give a better impression of the differences between various advanced transaction models and semantics-based concurrency control protocols – beyond the fact that they are meant to support different types of applications – we present selected examples. These examples not only shaped our understanding of the functional requirements for implementing extended transactions, but are commonly identified in the literature as providing features useful for implementing advanced database applications.

For our background discussion on extended transaction processing, we present selected advanced transaction models in Section 2.1.1 and selected semantics-based concurrency control protocols in Section 2.1.2. We have attempted to provide a bird's-eye view of the functionality required for the different proposals. To this end, in Section 2.1.3 we identify key extended transaction services and relate advanced transaction models and semantics-based concurrency control protocols to them. Following this, in Section 2.1.4, we present an overview of related efforts to implement extended transactions. Finally, we put forth our strategy for developing the Reflective Transaction Framework in Section 2.1.5.

2.1.1 Advanced Transaction Models

Research on extended transactions was first motivated and necessitated by the functionality and performance deficiencies of traditional ACID transactions. Today, the area has attained some maturity, and a large number of advanced transaction models have been formulated. Before looking at specific examples, let us consider an informal definition. An extended transaction consists of either a set of operations on data objects that execute atomically in a predefined order, or a *set* of extended transactions with an explicitly given control related to the notions of atomicity, consistency, isolation, and durability [RC97]. This recursive formulation implies that an extended transaction may exhibit a rich and

complex internal structure; in contrast, traditional ACID transactions have a flat single-level structure.

The manner in which component extended transactions are combined to form an advanced transaction model typically reflects the semantics of the application for which it was originally designed. The application may allow the introduction of new, weaker notions of conflict among operations not possible with information available only on data objects and their types. For instance, operations invoked by two transactions can be interleaved as if they commuted, if the semantics of the application allow the dependencies between the transactions to be ignored. Such application-specific transaction synchronization might *not* achieve serializability, but still preserves consistency. Similarly, based on application semantics, in the event that an extended transaction fails, changes made by completed *components* of the transaction may be committed. The failed portions of the transaction can be retried, compensated, replaced by another (contingent) alternative transaction, or even ignored. These relaxed but controlled transaction guarantees provided by advanced transaction models potentially promise to cater to the functionality and performance needs of a wide range of emerging database applications.

The Nested Transaction model [Mos85], for example, has been proposed in the context of distributed languages to handle partial failures. However, Nested Transactions support only hierarchical computations, similar to the ones that result from procedure call invocations. The Recoverable Communicating Actions model [VRS86] supports arbitrary computation topologies, and proposed in the context of distributed operating systems, where interactions are more complex. In addition, Split and Join Transactions [PKH88], Compensating Transactions [KLS90], Cooperative Transactions [MP92, NZ90], and Sagas [GMS87] have been proposed for capturing the interactions found in advanced applications. In the remainder of this section we will review some of these advanced transaction models.

Possibly the best known advanced transaction model is the Nested Transaction model [Mos85]. In this model, extended transactions are composed of subtransactions or "child" transactions, which are designed to localize application failures and to exploit transaction parallelism. Each subtransaction can be further decomposed into other subtransactions, and thus an extended transaction may expand in a hierarchical manner. A subtransaction executes atomically with respect to its siblings and to other, nonrelated transactions, and is atomic with respect to its parent. A subtransaction can abort independently without causing the abort of the whole transaction; but if a parent transaction fails, then it will abort all active child subtransactions.

A subtransaction can potentially access any data object that is currently accessed by one of its ancestor transactions. In addition, any data object in the database is potentially accessible to the subtransaction. When a subtransaction commits, the data objects that

it modified are made accessible to its parent transaction. However, the effects on the data objects are made permanent in the database only when the root transaction commits.

There is an emerging trend in the use of databases in applications that involve long-running activities that possess transaction-like properties. These activities involve a number of steps, where subsequent steps in the activity are executed depending on the disposition of steps that have already executed, and depending on the state of the data and environment. A number of transaction models have been proposed to organize and manage such activities; one of the more popular is the Split/Join Transaction model.

In the Split/Join Transaction model [PKH88, KP92], it is possible for a transaction t_a to *split* into two transactions, t_a and t_b, and for two transactions, t_a and t_b, to *join* into one joint transaction t_b. For simplicity, we will discuss Split Transactions and Joint Transactions as two distinct advanced transaction models.

Split transactions allow a user to split a (long) transaction dynamically into two or more smaller transactions in such a manner that the two new transactions are serializable. This allows an application to release partial results to other transactions by committing the transaction that has been split off, even before the transaction from which it split is committed. Splitting also allows other short-duration transactions that are waiting for data objects to be released as a result of the partial commitment, to proceed. This approach has the potential for increasing concurrency, as short duration transactions would not be made to wait until the long transaction commits. Split transactions can further split, creating new split transactions. This leads to a type of hierarchically structured computation different from that of nested transactions. Such possibilities are especially beneficial for CAD/CAM, VLSI design, and software development applications because of their long-running activities [RC92, CR94].

In the Joint Transaction model, it is possible for a transaction, instead of committing or aborting, to join another transaction. The joining transaction releases its data objects to the *joint* transaction. However, the effects of the joining transaction are made permanent in the database only when the joint transaction commits. If the joint transaction aborts, the joining transaction is aborted too.

The Cooperative Transaction Group model was introduced to support collaborative work [MP92, RC92], primarily in design and software engineering environments. In this model, extended transactions can *create* and *join* a cooperative transaction group. Each cooperative group consists of a set of *member* transactions, whose interactions are structured to reflect the decomposition of the task they are working on. The execution of the member transactions in a cooperative group need not be serializable; rather, the transaction group defines the rules that regulate the interactions among member transactions. This correctness criterion is referred to as *cooperative serializability* [MP92, RC92].

Because of the cooperative nature of the transaction group, it is not assumed that the operations of a single member transaction necessarily leaves the database in a correct state. Instead, the effects of the member transactions are only made permanent in the database when the entire group commits. If the transaction that created the group aborts, then all member transactions are forced to abort, while member transactions can abort independently without causing the abort of the cooperative group.

2.1.2 Semantics-Based Concurrency Control Protocols

Concurrency control is the activity of coordinating the actions of different transactions when they simultaneously access a shared database. When two transactions are allowed to interleave their accesses to the database arbitrarily, anomalies can occur and the database can be left in an inconsistent state. The traditional approach to preventing such inconsistencies has been to provide a concurrency control mechanism that guarantees serializability [BHG87]: a concurrent execution is serializable if it is equivalent to some serial execution of the same transactions. Traditional concurrency control schemes, such as two-phase locking and timestamping, use a conflict-based serializability test in which the database is viewed as a set of records, operations read and write records, and two operations conflict if one is a write. However, these conflict tests are overly conservative and can seriously degrade performance.

Various techniques have been proposed to increase concurrency by effectively reducing the time a transaction must hold a lock. Examples are transaction splitting [PKH88, KP92], discussed in the previous section, and the altruistic locking protocol [SGMS94].

The *altruistic locking* protocol is an extension of two-phase locking that accommodates long-lived transactions. Under two-phase locking, short transactions can encounter serious delays, since a long-lived transaction may tie up database resources for significant lengths of time. In altruistic locking, a transaction t_i can *donate* a data object that it will no longer access, thus allowing other transactions to access it. Donating a data object does not release the lock t_i holds on the data object, but simply allows other transactions to acquire a conflicting lock on the data object. Transaction t_i must still explicitly unlock data items that it has donated – thus, t_i is free to continue locking data items even after some have been *donated*. The basis for altruistic locking is the recognition that a transaction t_j that obtains a lock released earlier by a transaction t_i must be serialized after t_i. This is ensured by ascertaining that t_j executes in the "wake" of t_i; that is, all accesses to data shared by t_i and t_j occur in the order t_i followed by t_j.

One advantage of altruistic locking is that transactions need not advertise their access patterns beforehand. Also, although transactions are not two-phase, it is compatible with the two-phase locking approach, since a transaction is not *required* to release locks early.

This protocol is especially beneficial when long-duration transactions coexist with short transactions, since the latter do not have to wait until the former are completed.

The search for higher concurrency has been carried further by viewing the database as a collection of objects that are instances of abstract types manipulated through abstract operations with known semantics. Whereas with untyped data, all operations on a particular data item conflict unless both are reads, the semantics of the abstract operations can be used to detect operations that, for example, modify the value of an object and yet do not conflict. Serializability is still the goal of this approach, but the use of operation semantics allows the notion of conflict to be narrowed and hence permits increased concurrency. One example of this approach is operation commutativity [Wei88].

Operation *commutativity* is the traditional semantic notion used to determine if two operations can be allowed to execute concurrently (for example, two reads commute). If two operations commute, then their effects on the state of a data object and their return values are the same irrespective of their execution order. For example, consider the increment and decrement operations defined on a data object, which do not return any value of the data object. Both increment and decrement operations update the value of the data object, but the conflict between them can be ignored because these are commuting operations. Moreover, if the concurrency control mechanism allows only commuting operations to execute concurrently, then it prevents cascading aborts.

Operation *recoverability* is another criterion used to define conflict among operations. An operation q is recoverable relative to another operation p if q returns the same value whether or not p is executed immediately before q. For example, a successful Push operation on a stack is recoverable relative to a preceding Push operation on the same stack. Even if the preceding Push operation is aborted and its pushed value is removed from the stack, the pushed value and the return value of the second Push operation are not affected. Transactions invoking operations p and q are required to commit in the order of the invocation of the two operations. When used with locking-based protocols, recoverability, like commutativity, avoids cascading aborts while also avoiding the delay in the processing of many noncommutative operations.

Epsilon Serializability (ESR) is a generalization of classic serializability that relaxes operation conflicts, to *explicitly* allow a bounded amount of inconsistency in transaction processing. The amount of inconsistency is given by some measure of the database operations or distance function over the database state space [RP95]. In a commercial banking application, for example, inconsistency would be measured in dollars. ESR enhances concurrency by permitting query transactions to read uncommitted data from a concurrent update transaction and by permitting update transactions to write to data items locked by a concurrent query transaction. For example, an epsilon transaction that can tolerate

a bounded amount of inconsistency, measured in dollars, can query the balance of bank accounts and execute in spite of ongoing concurrent updates to the database.

Let us try to consolidate the different concepts that we have introduced in this section. Two operations *conflict* when their effect on the data objects or the values they return are dependent on execution order. Nonconflicting operations are said to be *compatible*. One approach to reducing conflicts is to simply reduce the amount of time a transaction holds a lock, as illustrated by transaction splitting and the altruistic locking protocol. Another approach is to use operation semantics to define *semantic compatibility* between operations. The simplest compatibility relationship is the one based on *operation commutativity* and is typically used to determine whether two operations can execute concurrently while updating the objects in place. With *recoverability*, the conflicting operation is allowed to execute concurrently, provided that the abort of the first operation does not lead to the abort of the second operation executed later. Recoverability demands that the two transactions commit in the order that they executed the two operations. In addition, it is possible to utilize transaction semantics to define compatibility, as illustrated by epsilon-serializability and the proclamation method, both of which bound the amount of inconsistency of the result returned by a transaction. It is important to note that all of these semantics-based concurrency control protocols can be seen as extensions of conventional lock-based concurrency control, in which semantic information is used to grant *semantically compatible* lock requests, even though they conflict at the level of the implementation.

2.1.3 Common Extended Transaction Functionality

Many different extended transaction types have been proposed. In order to characterize the functional requirements of existing proposals, thereby shedding light on the similarities among and differences between them, we present Table 2.1. This table identifies three extended transaction services and relates specific extended transactions to these new services. Specifically, we identify the advanced transaction models and semantics-based concurrency control protocols that require the extended services of *transaction restructuring*, *semantic transaction synchronization*, and *execution control*. For concreteness, we offer a brief description of each service and provide an example of an extended transaction that requires this service, but defer the detailed description of these extended services until later in the dissertation. We can see that even though these advanced transaction models and semantics-based concurrency control proposals were motivated by different applications, they share common extended functional requirements. And from an implementation perspective, we can see that the functional requirements of a wide range of extended transactions can largely be satisfied by these three extended transaction services.

Table 2.1: Functional characteristics of extended transactions.

Extended Model	Trans. Restructuring	Semantic Synchronization	Execution Control
Nested Trans. [Mos85]	static	transaction	commit, abort
Sagas [GMS87]	static	NA	execution order, commit-on-abort
Split and Join Trans. [PKH88, KP92]	dynamic - partial dynamic - global	NA	NA
Chained Trans. [Chr91]	dynamic - global	NA	execution order
Reporting Trans. [CR94]	dynamic - partial	NA	NA
Compensating Transactions [KLS90]	NA	transaction	commit-on-abort
RCA [VRS86]	dynamic - partial	NA	commit, abort
NT/PV [KS88]	static	transaction, application	execution order
Flex [AYWM90]	static	transaction	commit, abort
ConTracts [WR92]	static	NA	execution order
Coop Groups [RKT+95]	dynamic - global	transaction	commit, abort
Patterns/TG [NZ90]	static	operation, transaction	commit, abort
Polytransactions [ASK92]	dynamic - global	operation	execution order
DOM-Transactions [BOH+92]	static	transaction	execution order, commit, abort
Commutativity [Wei88]	NA	operation	NA
Recoverability [BR91]	NA	operation	abort
DSR - Significant Dependencies [SS84]	NA	operation	serial order, commit, abort
ESR [RP95]	NA	operation, transaction	NA
SD - Serial Dependencies [Wei88]	NA	operation, transaction	serial order
QSR - QuasiSR [DE89]	NA	operation, transaction	serial order
Co-SR - Cooperative Serializability [MP92]	NA	transaction	commit, abort
Proclamations [JS92]	NA	operation, transaction	NA
Altruistic locking [SGMS94]	NA	operation, transaction	serial order

Transaction restructuring allows an advanced transaction model to impose an inherent structure on component transactions. We classify the restructuring required by an

advanced transaction model as either *static* or *dynamic*, depending on whether the structure is determined in advance or whether restructuring can occur dynamically at runtime. The Saga model is an example of static restructuring, in which operations and resources are specified in advance for each component transaction, as is the execution order between these component transactions. The Split and Join transaction models are examples of dynamic restructuring, in which the component transactions and resources are determined dynamically at runtime. Dynamic transaction restructuring can be further classified as *global*, in which a transaction releases all resources it holds, or *partial*, in which a transaction selectively releases resources to another transaction or the stable database.

Semantic transaction synchronization permits a transaction processing system to exploit semantic information to coordinate extended transactions. We can classify semantic synchronization requirements depending on whether the extended transaction model exploits the semantics of the operations, the individual transactions, or the application itself to determine semantic compatibility.

Execution control is the ability of a transaction processing system to control the execution order of transactions in an advanced transaction model. We can classify the execution control requirements of an extended transaction by the nature of control required over its component transactions. The Nested Transaction model, for example, allows child subtransactions to abort, but they cannot commit before the parent transaction commits; however, if the parent transaction aborts then all child subtransactions must abort as well. Transactions following the recoverability protocol can form abort dependencies when conflicts are relaxed, while transactions following the altruistic locking protocol form serial order dependencies when they share access to data objects.

It is not possible to capture in a single table all the nuances of the advanced transaction models and semantics-based concurrency control protocols in the literature. Furthermore, given the many papers in these areas, it is not possible to be all-inclusive. We believe, however, this is a good starting point for understanding the functional requirements of extended transactions. Transaction restructuring, semantic transaction synchronization, and execution control can be viewed as a common set of services for implementing many advanced transaction models and semantics-based concurrency control protocols that exist today, and if properly designed, these services can be tailored to meet the needs of a range of advanced database applications.

2.1.4 Related Extended Transaction Implementation Efforts

This section presents four extended transaction implementations of various sorts — Asset [BDG+94], TSME [GHKM94], Apricots [Sch93], and Pern [Hei97]. These systems were chosen because they represent leading edge solutions to the problem of implementing and

managing extended transactions. The implementations vary widely in both form and focus. In our discussion we shall present aspects of the structure and design of each, and identify what support, if any, they provide for the extended transaction services identified in the previous section.

ASSET

Using a C++ programming interface, Asset [BDG+94] (A System for Supporting Extended Transactions) allows a programmer to produce programs with extended transaction specifications compiled into application code. Asset consists of a set of transaction primitives which are classified as *basic* or *new* primitives. The basic primitives `initiate(f, args)`, `begin(t)`, `commit(t)`, `wait(t)` and `abort(t)`, are similar to transaction control operations found in most transaction processing systems. The new primitives, `delegate(`t_i`, `t_j`, obj)`, `permit(`t_i`, `t_j`)` and `form-dependency(type, `t_i`, `t_j`)`, are included in the system to enable the construction of advanced transaction models.

Briefly, the primitive `initiate(f, args)` creates a new transaction that executes the function f with the arguments *args*. The primitives `begin(t)`, `commit(t)` and `abort(t)` respectively start, commit, and abort the transaction whose transaction identifier is t. Waiting for a transaction t to complete is accomplished by using the primitive `wait(t)`, which returns the value 1 when transaction t commits and 0 when t aborts. The primitive `delegate(`t_i`, `t_j`, obj)` transfers the responsibility of operations performed on data object *obj* from transaction t_i to t_j. Cooperation among transactions is achieved by using the `permit(`t_i`, `t_j`)` primitive, which permits transaction t_j to perform conflicting operations on data objects held by t_i, without creating a conflict edge in the serialization graph from t_i to t_j. The permit operation can be used to implement semantic synchronization using transaction semantics. The last primitive `form-dependency(type, `t_i`, `t_j`)` establishes a dependency of the specified type between t_i and t_j, where *type* includes transaction commit and abort dependencies.

To illustrate how these primitives are used, consider the sample code fragment in Figure 2.1, taken from [BDG+94], that executes a simple reservation workflow involving hotel, car, and flight reservations. The function `t_conference` attempts to complete all the necessary reservations for a particular conference. First, a ticket is booked on the first airline that has available seats; Delta, United, and American Airlines are tried in order. This operation will require anywhere from one to three transactions. Next, the hotel reservation is attempted; on failure, the flight reservation is canceled through a compensatory transaction, `t5`, and 0 is returned. Finally, a car reservation is attempted for either National or Avis. If at least one succeeds, the arrangements are complete.

```
// the following two functions make (or cancel) the appropriate reservations;
// The last two functions in the example are compensations.
void flight_reservation (Airline air, Date d1, Date d2);
void hotel_reservation (Hotel h,  Date d1, Date d2);
void car_reservation (CarRent c,   Date d1, Date d2);
void cancel_flight_reservation (Airline air, Date d1, Date d2);
void cancel_hotel_reservation (Hotel h,    Date d1, Date d2);
// Using these functions, the desired workflow can be defined as follows:
void exclusive_car_reservation(CarRent car, Date d1, Date d2, tid t) {
  car_reservation(car, d1, d2);
  if (wait (self())) abort(t);
}
int t_conference(Date d1, Date d2){
  tid t1, t2, t3, t4, t5, t6;
  Airline *air;
  // Make some airline reservation
  t1 = initiate(flight_reservation, "Delta", d1, d2);
  begin(t1);
   if (!commit(t1)) {
     t2 = initiate(flight_reservation, "United", d1, d2);
     begin(t2);
     if (!commit(t2)) {
        t3 = initiate(flight_reservation, "American", d1, d2);
        begin(t3);
        if (!commit(t3)) return 0;          // Activity failed
        else air = "American";
     } else air = "United";
   } else air = "Delta";
  // Flight reservation has been made at this point
  t4 = initiate(hotel_reservation, "Equator", d1, d2);
  begin(t4);
  if (!commit(t4)) {
    do { t5 = initiate(cancel_flight_reservation, air, d1, d2);
         begin(t5); }
    while (!commit(t5)) ;
    //wait for commitment before proceeding
    // Compensate for the flight reservation already made
    return 0;
  }
  // At this point, hotel and flight reservations have both been made
  t5 = initiate(car_reservation, "National", d1, d2);
  begin(t5);
  t6 = initiate(exclusive_car_reservation, "Avis", d1, d2);
  begin(t6);
  if (wait(t5)) {  //whichever completes first wins
     abort(t6);
     commit(t5);
   } else commit(t6);
   return 1;  // Successful completion of all reservations
}
```

Figure 2.1: Sample Asset workflow program.

A shortcoming of the Asset approach is its low-level focus. Asset primitives allow for programming-in-the-small, but do little to aid using an extended transaction processing system. Indeed, while care was most likely exercised throughout in the development of the Asset example (presented in Figure 2.1), taken from [BDG+94], to guarantee that either all work was performed, or partial work was aborted or compensated for, this procedure still has a flaw. If both car reservation transactions fail, the procedure will return a successful status indicator; there is no way to recover from failure of transactions `t5` and `t6`. This gives rise to the argument that embedding extended transaction extension code in application code is inappropriate for systems requiring complex transaction behavior, since it is hard to prevent bugs like these from happening. Moreover, requiring application programmers to "step down" from application code to specify extended transaction

functionality is unreasonable; the task of correctly coding an application is hard enough, without also requiring the programmer to develop the necessary extended transaction support. However, the primitives do capture useful extended transaction behaviors and could be generated from a higher-level specification, showing promise for the approach.

TSME

The Transaction Specification and Management Environment (TSME) [GHKM94] is a transaction processing toolkit, specifically designed to be used in combination with the DOMS architecture [MHG+92] of GTE Laboratories. The toolkit was developed for formulating advanced transaction models in a workflow application domain [GHS95a]. The three main components of TSME are a specification language, a Transaction Dependency Specification Facility (TDSF), and a corresponding programmable Transaction Management Mechanism (TMM).

Advanced transaction models are specified in TSME as transaction dependencies between constituent ACID transactions. In TSME, transaction dependencies are described using 5-tuple elements of the form (t_i, T, O, E, P) where t_i is the dependent transaction, T is the set of transactions that t_i depends on, O is the set of data objects the dependency must consider, and E and P are logical predicates representing the enabling condition and postcondition, respectively. E denotes when the postcondition must be evaluated, while evaluation of the postcondition determines whether the dependency is satisfied or not. Transaction dependencies are classified into *state dependencies* and *correctness dependencies*. State dependencies express relationships between the states of transactions where a transaction can be in either the begin, prepare, commit or abort state. Three kinds of state dependencies are supported: *backward*, *forward* and *strong*.

TSME supports *static* structuring of transactions through the definition of *complex transactions*. In TSME, a complex transaction is defined by a collection of ACID transactions and a set of dependencies defined between these component transactions. TSEM does not, however, provide implementation support for *dynamic* restructuring. The only semantic synchronization supported by TSME is *transaction* level, in which the components of a complex transaction can access a shared set of DOM objects. One of the strengths of TSME is its support for execution control. By defining *backward*, *forward* and *strong* dependencies over transaction state, it is possible to coordinate the execution of the components in a complex transaction, imposing commit, abort and serial orderings.

Transactions dependencies are submitted to the TDSF which translates them into combinations of event-condition-action (ECA) rule definitions and instructions to transaction schedulers that will serve to constrain the execution structure of the individual transactions. Once processed, the extended transaction specification is stored in a repository

managed by the TDSF. The final component of the architecture, the TMM, supports the implementation of the advanced transaction model by configuring a DOM-specific transaction runtime environment to ensure the preservation of the transaction dependencies.

The TSME provides a promising framework for constructing simple workflows, separating workflow modelling from runtime implementation. The approach allows workflow modelers to reason about the correctness of a workflow based on the specified transaction dependencies and provides repository facilities for maintaining developed workflows. While a research prototype of the programmable TMM was implemented at GTE on DOMS, the TSME was never fully implemented and the project was eventually terminated when commercial workflow products became available.

APRICOTS

The next system we examine is Apricots [Sch93] (A PRototypical Implementation of a COnTract System). Apricots is not a general-purpose extended transaction implementation facility, but was developed to implement the ConTract model [WR92]. The ConTract model was proposed to provide a basis for defining and controlling long-lived activities. Specifically, it is an advanced transaction model with a mechanism for grouping traditional ACID transactions into a multi-transaction-like activity.

A *ConTract*, which is the basis for the model, consists of a set of predefined ACID transactions called *steps* and a separate explicitly specified execution plan called a *script*. In addition to the *relaxed isolation* that results from the division of a ConTract into multiple ACID transactions, ConTracts provide *relaxed atomicity*, so that a ConTract may be interrupted and reinstantiated. For a given ConTract it is guaranteed that execution will either successfully complete within a finite amount of time, or a state logically equivalent to the original state will be reconstructed via *compensating steps*.

Steps are the basic atomic building blocks of ConTracts. They represent elementary units of work and are implemented by conventional ACID transactions. There is no internal parallelism in a step (visible to the script level) and therefore the transaction can be coded in any arbitrary sequential programming language. Control flow between the steps is specified by a scripting language that includes the usual control elements, such as sequence, branch, loop and some parallel constructors, thus providing a means for explicitly specifying control flow for operations on shared persistent data objects. It is also possible to define dependencies between the steps (transactions implementing the steps).

Because ConTracts are built out of traditional ACID transactions, the results of which are externalized before the entire ConTract is finished (or compensated), there is a need for mechanisms to synchronize ConTracts that are running in parallel. In the ConTract model this is accomplished by defining so called *invariants*. Through invariants it is possible to

protect shared data from the concurrent access of other ConTract steps. These invariants do not need to prevent concurrent access of shared data items totally, but rather ensure that the value of the data items stay within the defined limits.

ConTracts define *static* structure between the transactions that make up the contract, but can not support *dynamic* restructuring of any kind. The ConTract model and Apricots architecture approach does *not* support any kind of cooperation between different users, and can only support *transaction* level synchronization between the components of a contract. Execution control of the transactions that make up a ConTract is defined explicitly in the ConTract script. The ConTract manager is responsible (among other things) for the execution of the script and the failure tolerant control flow management. The ConTract manager communicates with the Apricots transaction manager to implement the transactional semantics of a ConTract.

The Apricots implementation, described in [Sch93], is essentially a transaction processing monitor designed to support the implementation and management of ConTracts. Apricots consists of the following components: a *ConTract manager*, *step server*, *transaction manager* and a *resource manager*. The ConTract manager is responsible for the execution of the script. It has to guarantee the reliable execution of a started ConTract and is responsible for the forward recovery in the event of a crash. The step server manages the control flow. It decides which steps to activate and sends an asynchronous call to the transaction manager that will execute each step as an ACID transaction. The Apricots resource manager manages data collections and supports functions on data objects.

To illustrate, a user or application can start a ConTract of a specific type with the command `activate`(*contract-script-name*). The ConTract is assigned a system unique identification (cid). The execution of a ConTract, addressed by its cid, can be *suspended, resumed, migrated* to another machine in the network, or *compensated*. The textual description of the ConTract script is transformed by the ConTract manager into a predicate transition net for reasons of efficiency. If the start event of a step occurs, the ConTract manager gives an appropriate step-server the order to execute the step. The execution of the step-code is done asynchronously with the execution of the script. If the execution of the step is finished, a completion event is sent to the ConTract manager. The ConTract manager provides a function *stepFinished* that receives the return messages of steps and executes the script depending on the return values. The advantage of an event-driven script execution is that the ConTract manager does not have to wait synchronously (blocking) for the end of a step execution.

PERN

Pern is an external transaction manager developed at Columbia University [Hei97]. Pern supports ACID transactions, and provides the option to define application or project specific concurrency control using a coordination modeling language (CORD) [HK97].

Pern was designed to be incorporated into a software development environment, to provide transaction support for the process of developing software. Pern defines a basic transaction model that implements ACID transactions with a shared read and an exclusive write lock mode. Standard transaction operations such as tx_begin, tx_commit, tx_abort, tx_lock and tx_unlock are provided. The architecture of the Pern transaction manager defines a number of *events* related to the execution of a transaction, and allows users to define handlers that are to be invoked *before* and *after* each event. By defining an appropriate set of handlers, a programmer can alter the execution of the transaction operations in various ways to satisfy the needs of a particular extended transaction.

CORD includes tables for specifying the compatibility of application-specific locking modes. More significantly, it provides a rule-based (condition-action) notation for describing special-purpose conflict resolution when transaction conflicts arise; the CORD notation builds on the conflict resolution language introduced by Barghouti [BK91]. To illustrate, the CORD rule presented below specifies that when a lock conflict occurs on a MANUAL data object, the handler will check whether the two conflicting transactions are running the "revise_manual" task and "add_section" task respectively, and whether the "revise_manual" task is run by the owner of the manual and the "add_section" task is run by the co-author. If these conditions hold, then both transactions are allowed to access MANUAL. This particular CORD rule also specifies that the "revise_manual" transaction should receive a notification via the "EDIT_conflict" message.

```
EDIT_conflict [ MANUAL ]
bindings:
    ?t1 = holds_lock()
    ?t2 = requested_lock()
body:
    if (and (?t1.rule = revise_manual)
            (?t2.rule = add_section)
            (?t1.user = ?ConflictObject.owner)
            (?t2.user = ?ConflictObject.coauthor))
    then{
        notify(?t1, "EDIT_conflict")
        ignore()
    }
end_body;
```

By coupling CORD mechanisms with the Pern event architecture, it should be possible to implement semantic transaction synchronization policies that utilize operation, transaction and application level semantics. In addition, by defining the appropriate handlers it should also be possible to implement *static* transaction restructuring, though no evidence of this is presented in the literature. Pern, however, does not provide support for *dynamic* restructuring, nor does it provide explicit support for execution control. These extended services were simply outside the original design goal of Pern and CORD, which was support for cooperative transactions in a software design environment.

Summary

This quick tour through the related implementation efforts has revealed a host of mechanisms and approaches to implement extended transactions. In addition, these implementations use different mechanisms to allow their facilities to be tailored, controlled or adapted to the variety of extended transactions which they might support. As we pointed out earlier, these systems have largely implemented base transaction support from scratch, rather than building on conventional transaction processing software. We also note that few of the related implementation efforts address transaction restructuring and semantic synchronization, thereby limiting the range of extended transactions they can implement.

2.1.5 Reflective Transaction Framework Implementation Strategy

With an improved understanding of the functional requirements of extended transactions and the related implementation efforts, we turn our attention to choosing an implementation strategy. There are two options. First, we could start anew by building an extended transaction facility from scratch, similar to the TSME and PERN. Such an effort removes the burden of preexisting decisions and tradeoffs in an existing transaction processing system, and allows the use of knowledge of extended transaction requirements to guide the development of a new extended transaction facility. The danger, however, is of making new, more grievous design mistakes. In addition, so much time can be spent building up base transaction support and re-inventing wheels, that little innovation takes place.

Alternatively, an existing transaction processing system may be adopted as a platform upon which incremental development may take place. Extending an existing transaction system limits the scope of the work, but ensures that one is never far from a functioning system that can be tested to guide development, and secures a base of users upon completion. Moreover, if an incremental extension fails, only the failed step must be repeated, not the entire project. If the extensions are relatively modest in scope, such incremental steps do not need to promise dramatic new functions to get immediate results. On the

down side, such an approach may necessarily limit the amount of innovation and creativity brought to the process and possibly carry along any preexisting biases built in by the originators of the transaction system, reducing the impact the research might have on providing broad-ranging support for implementing extended transactions.

We have chosen the second direction, deciding to provide implementation support for extended transactions by extending the base services of a conventional transaction processing system. The key insight that shaped this decision was the understanding that each of the extended services essential for implementing extended transactions can be realized as an incremental extension of base transaction processing services. Indeed, it is a claim of this thesis that conventional transaction mechanisms can be used successfully to support the implementation of extended transactions. A second, more pragmatic, consideration is the recognition that conventional transaction processing systems, in particular TP monitors, have accumulated large amounts of transaction implementation technology. We don't think that it would be particularly clever simply to throw this technology away and build an extended transaction facility from scratch. Thus, we will leverage existing functionality, to the extent possible, and incrementally add functionality required to implement extended transactions. Moreover, we shall endeavor to do so in a manner that ensures that the TP monitor continues to function as before, so that applications built using ACID transactions do not have to be modified.

In the next section, we consider issues that arise when we attempt to apply our incremental extension strategy directly to a conventional TP monitor. Later, in Section 2.3, we introduce Open Implementation as a design approach to meet the challenges that arise, such as providing effective access to legacy interfaces, structures and functions; the maintenance of shared representations; and the treatment of unavoidable problems (e.g., scope control and the conceptual separation of various transaction extensions).

2.2 Conventional TP Monitor Architecture

For the last 20 years TP monitors have provided a general framework for transaction processing, supplying the "glue" to bind together the components of a transaction system through services such as multithreaded processes, interprocess communication, queue management, and system administration [Ber90]. For our background discussion, we use a simplified description of an OTLP monitor consisting of five components: 1) transactional application program, 2) transaction manager, 3) lock manager, 4) log manager, and 5) resource manager. The structure of these components is shown in Figure 2.2. In a commercial setting, we might find an TP monitor such as Transarc's Encina providing access to a resource manager such as Microsoft's SQL Server.

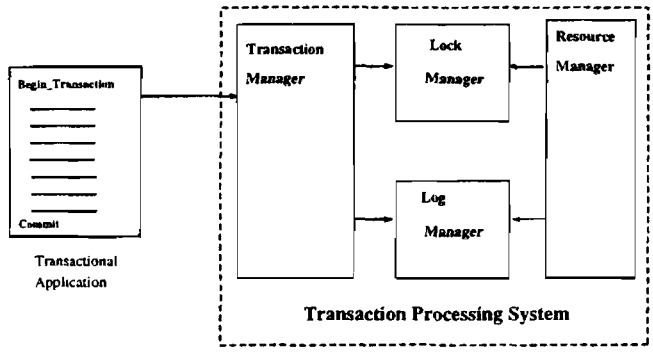

Figure 2.2: Modular Functional Components of an TP Monitor.

The architecture of commercial TP monitors varies widely on a spectrum from unstructured (i.e., a single monolithic software module, not decomposable into its component parts) to well-structured (i.e., modular, hence decomposable). These architectural differences reflect the history of TP monitor development. When early TP monitors, such as IBM's CICS, were first developed, they were not as complex as today's TP monitors. Not only was functional decomposition unnecessary for implementing the TP monitor, it was not even conceived. As functional requirements increased (such as three-tier client server), and new architectural forms (like distribution and heterogeneity) were introduced, implementation required more functional decomposition, thus prompting methods such as structured design, which resulted in the TP monitor software being "architected" into functional components or modules. Today, modern TP monitors, such as Transarc's Encina, DEC's ACMSxp, and IBM's CICS/6000, are modular and constructed from open transaction processing middleware [Ber96]. Each of these middleware modules provides a specific transaction service, such as *transaction management*, *lock management*, and *log management*, and exports its transaction processing services through a relatively simple and uniform "application programming interface" (API).

In the remainder of this section we will describe the services provided by the TRANSACTION MANAGER and the LOCK MANAGER. The descriptions are high-level, but serve to advance our claim that these base services provide useful functionality for implementing extended transactions, and that design decisions have limited their applicability by committing to a particular approach to transaction support.

2.2.1 Transaction Manager

The Transaction Manager processes the basic transaction control operations for transactional applications, such as `Begin`, `Commit`, and `Abort`. An application calls `Begin` to start executing a new transaction. It calls `Commit` to ask the Transaction Manager to

commit the transaction. It calls **Abort** to request the Transaction Manager to abort the transaction. All operations within the scope of a transaction in an application go through the Transaction Manager, whereas operations outside the scope of a transaction may be issued to the Resource Manager directly.

The Transaction Manager is primarily a bookkeeper that keeps track of all active transactions and available Resource Managers, and maintains information on the transaction accesses to Resource Managers – the status of each transaction; for example, *active*, *prepared*, *aborted*, *committed* – and the resources held by a transaction. This requires some cooperation with the transactional application and the Resource Managers.

When an application calls **Begin**, the Transaction Manager creates a unique identifier for the transaction called a *transaction identifier* (TID) and allocates a descriptor for the transaction. The transaction descriptor is used to hold all information used in processing the transaction. Subsequent calls submitted by the application have the transaction's TID attached. The descriptor is the focal point during a transaction's execution. Depending on the scheduler implemented in the TP monitor, the descriptor may also be used to maintain a list of the locks held and requested by a transaction, and a list of data objects read and written by a transaction. These lists are often referred to as *locks held*, *locks requested*, *read set* and *write set*, respectively.

A Transaction Manager may perform a number of other functions depending on the specific concurrency control and recovery algorithms implemented by the Resource Manager. For example, if two-phase locking is used for the concurrency control algorithm, then the Transaction Manager will participate in enforcing the protocol, and may be involved in detecting and resolving lock deadlocks. A deadlock between two transactions occurs when each transaction holds a lock on a data object which the other transaction is attempting to acquire. The Transaction Manager can detect deadlocks by examining the *locks held* and *locks requested* lists in the transaction descriptor of active transactions, and constructing a wait-for graph [BHG87]. A cycle in the wait-for graph indicates that a transaction deadlock exists.

When a transactional application finishes execution and issues the commit operation, the commit operation goes to the Transaction Manager, which processes the operation by executing a two-phase commit protocol. Similarly, if the Transaction Manager receives a message to issue the abort operation, it tells the Resource Manager to undo all the transaction's updates; that is, to abort the transaction at each database system.

Recall from our discussion on extended transactions in Section 2.1 that many advanced transaction models and semantics-based concurrency control protocols require explicit control over the execution of member transactions. However, the set of services provided by the Transaction Manager does not provide this level of support. There are two major

shortcomings. First, while an application can control the execution of the transaction it is currently running, it can not influence other transactions running on the TP monitor. This is because an TP monitor provides each transaction with the illusion that it is executing in isolation, and thus the Transaction Manager does not export the necessary services for an application to view other concurrently executing transactions or to explicitly control their execution. Second, the Transaction Manager does not allow applications access to the state information on active transactions, such as the transaction descriptor, nor does it allow applications to update or store additional information in these structures. However, if an application is going to execute a *cooperative transaction group*, it will require all of these services to designate a transaction as the group coordinator, identify other active transactions that are members of the group, and control their execution in order to implement group commit and abort dependencies.

2.2.2 Lock Manager

Conventional TP monitors typically use a locking protocol to synchronize transactions. The protocol allocates locks to requesting transactions, and detects conflict and deadlock among the requesters. Traditional protocols support just two basic lock types, *Read* (Share) and *Write* (Exclusive), and every data access is automatically cast as one or the other, regardless of the operation, the type of data, or the application context.

The Lock Manager is a major component in synchronizing transactions. However, a Lock Manager does not enforce the locking protocol. Enforcing the locking protocol is the responsibility of the resource manager[1]. In fact, the Lock Manager is essentially a black box that manages locks in the manner prescribed by the software modules that invoke the Lock Manager. Conceptually, the scheduler invokes the Lock Manager to determine whether there are conflicts that prevent the granting of the locks and ensuing actions from being scheduled or executed immediately. A Lock Manager may delay granting some locks and thus delay the corresponding actions when conflicts occur. In addition, it manages the data structures necessary to handle deadlock detection.

The interface to a typical Lock Manager exports the following functions:

1. *lock:* Executes a lock request for a single transaction;
2. *unlock*: Removes a previously granted lock on a data object for a transaction;
3. *unlock_all*: Releases all previously granted locks for a transaction.

[1]It is difficult to isolate a single module that implements the scheduler. For performance reasons, a scheduler's functionality is typically distributed among different pieces of the TP monitor software such as the Transaction Manager, access method routines, and Lock Manager.

A lock request for a specific transaction involves specifying values for the identifier of the requesting transaction (*tid*), the identifier for the lock being requested (*lid*), and the mode in which the lock is requested (*mode*). The Lock Manager will grant transaction *tid* a lock on *lid* in mode *mode* if no other transaction currently holds a lock on *lid*, or the mode of the request does not conflict with the lock mode(s) currently granted on the lock. Transactions record all the locks they own in a bookkeeping structure referred to as a *lockset*. Once a transaction *tid* has acquired a lock *lid*, it adds it to its lockset.

If a lock conflict exists and a lock request is blocked, then the lock request must wait for all previously blocked lock requests to be granted. The only exception to this rule occurs when a transaction makes a lock request on a lock that it already holds. In this situation, the Lock Manager converts the requested lock mode to the weakest lock mode that is greater than or equal in strength to both the granted lock mode and the requested lock mode, and tests if the new request is compatible with all lock holders.

Figure 2.3 illustrates the implementation structures of a Lock Manager as commonly described in the literature [GR93], where each flag represents a latch.

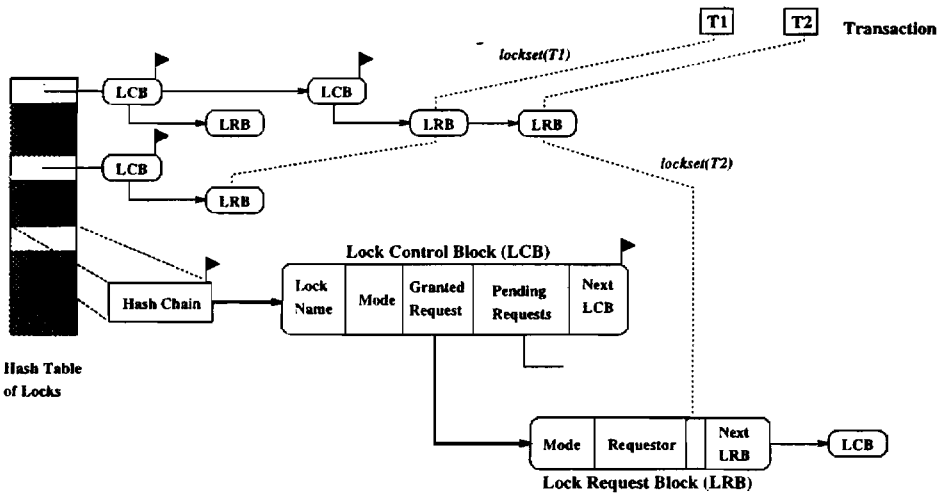

Figure 2.3: Conventional Lock Manager implementation structures.

A lock is implemented by a Lock Control Block (LCB) which contains information such as its name, its current mode (read or write), a latch, and links. A fixed-size hash table is used to store and retrieve LCB's using their name. The LCB is also the head of two doubly-linked lists of Lock Request Blocks (LRBs). One list implements the granted requests. The other list holds pending requests, and corresponds to blocked transactions. Each LRB relates to one transaction, and contains information such as the transaction's identity, the requested locking mode, and the links to its lockset.

To set a lock on a resource (e.g., a page or an object), the Lock Manager first selects a hash chain using the resource's name as the hashing key. If there is no LCB for the resource, it initializes a new one and appends it to the hash chain. Otherwise, it scans the LCB's LRB chain to see if the requester already has a LRB. If there is no LRB for the requester, it allocates a new LRB, chains it to the requester's lockset, and chains it to the right LCB chain according to the conflict detection result. Conflict detection is performed when the LRB chain is traversed to look up the LRB of the requester.

Looking back at our discussion in Section 2.1, we note that the basis for many extended transactions is the ability to use a synchronization algorithm that exploits the semantics of the operations, data and application to increase the number of transactions that can execute concurrently. However, the current Lock Manager services do not provide the necessary support for semantic transaction synchronization or for dynamic transaction restructuring. There are three main shortcomings. First, the interface to the Lock Manager does not allow an application to specify the conditions under which it should relax the definition of lock conflict. That is, an application cannot identify update operations that are compatible or declare that two transactions are members of a *cooperative group* and that the application will coordinate data access. The interface is *closed* in that the only information the Lock Manager will consider is the transaction identifier, the identifier for the lock being requested, and the mode in which the lock is being requested. Moreover, the only information the Lock Manager provides in response to a lock request is an acknowledgment that the lock has been granted or a message indicating that a conflict exists. There is no way for the application to identify the operation or transaction that is holding the lock, and thus it cannot determine the consequence of relaxing the conflict.

Second, the Lock Manager does not export its base interface, which typically consists of requests to *lock*, *unlock* and *unlock_all* data objects, outside of the TP monitor. Thus it is not possible for an application to access these operations and participate in managing lock resources on its own behalf. The lock services are presented to the application as a *black box* and are effectively hidden from the application.

Third, even if the application could access the interface of the Lock Manager in an attempt to manage data resources, the visible aspects of the underlying implementation are not sufficient to gain control over the abstractions. Consider an application that wishes to transfer ownership of a data object from a transaction t_i to another transaction t_j using the available commands in the interface. The application might first *unlock* the data object from transaction t_i and then *lock* it for transaction t_j. However, if there were another unrelated transaction t_k already waiting for a lock on the object, then as soon as the application unlocked the data object from t_i the Lock Manager would proceed to pass the lock to t_k. Clearly, the implementation constrains the way in which the lock service

abstraction behaves, and the original design decisions limit the Lock Manager applicability by committing to a particular approach to transaction support.

2.2.3 Building on Existing TP Monitor Functionality

One seemingly straightforward way to implement extended transactions would be to use the services provided by the functional components of an TP monitor directly. Two major impediments complicate this proposition. The first is the lack of an *interface for customization* of the TP monitor. The application interface to an TP monitor is *fixed*, as are the services provided by commands in this interface. Application programmers access transaction services through ACID transaction control operations, such as `Begin_Transaction`, `Commit_Transaction`, and `Abort_Transaction`. Ideally, application programmers would be able to define and then use similar transaction control operations for extended transactions, such as `Split_Transaction` or `Join_Transaction` introduced in the split/join transaction model. However, the single, fixed application interface does not provide access to the underlying transaction services of the TP monitor and does not permit extensions. Though the individual functional components of the TP monitor provide a rich set of transaction services, the application programmer would have to learn intricate details of the component-level API and run-time system. The size and complexity of the API alone presents a formidable barrier to even the most accomplished application programmer. The second impediment is the *level of customization.* TP monitor system-level code functions "underneath" the code of a transactional application, and is not subject to the same programming abstractions. This requires the TP monitor to be customized *outside* of the application, rather than *within* it, making it impossible for an application to specify its requirements for extended transaction behaviors at runtime. At best, a transaction system programmer could adjust TP monitor functionality through the API to implement a selected extended transaction model *a priori.* Unfortunately, such a customization to the run-time system could alter the entire system and, consequently, come at the expense of reusability; i.e., it is hard to localize the customization.

These issues, among others, combine to give users no convenient way to use TP monitor software directly to define new application interfaces and leverage existing transaction services to implement extended transaction functionality. It is for exactly these reasons that efforts to provide implementation support for extended transactions have gravitated towards the construction of entirely new transaction facilities. These efforts have proven to be expensive, and have limited practicality. What is required, from our perspective, is a framework that will carefully expose TP monitor functionality and provide the means to define new extended transaction services and application interfaces. The new services

defined by the framework must be separate from the TP monitor runtime, so that existing transactional applications will function properly, and be presented to application programmers through familiar transaction control operations so they do not have to "step down" to an operational description of extended transactions.

2.3 Reflection and Open Implementation

In this section, we discuss the problem of extending the functionality of software with reference to very general notions of abstraction in software design, and the Open Implementation approach. Open Implementations reveal aspects of system structure and behavior, providing applications (clients of the abstractions) with a principled means for examining and manipulating the internal operation of the abstractions. As a result, clients can become involved in how the infrastructure supports their operation, and can tailor the behavior of system abstractions to their own particular needs. Along with the Open Implementation approach, we shall also review the design principle on which it is based (*computational reflection*).

2.3.1 The Myth of "Abstraction"

Abstraction is one of the fundamental tools of computer science and system design. It is the means by which we can break down large problems into small ones and, conversely, combine small solutions to create large systems. Abstraction allows us to isolate one part of a system from another and consider the two separately. It is the key to analysis, modularity, and reuse; it is also, potentially, the source of a range of problems throughout systems design practice [CFN96].

The traditional form of abstraction in systems design relies on three basic components – *black boxes*, *clients*, and the *abstraction barrier*, as illustrated in Figure 2.4. The black box implements some abstraction which is offered to clients at an abstraction barrier. The abstraction barrier is a point of separation between client and implementation; the concepts, terms and structures in which the abstraction is phrased at the barrier are the only ones that clients can use to manipulate and control the abstraction. In an TP monitor, the abstraction barrier is typically presented as an application programming interface. The term "barrier" refers to the way in which the abstraction hides aspects of the transaction system implementation from the client. Behind the abstraction barrier, the internal details of ACID transaction processing functionality are not revealed to the client application.

There are two important features of abstraction being employed here. First, *separation* divorces the use of the abstraction from the details of its implementation, allowing a client

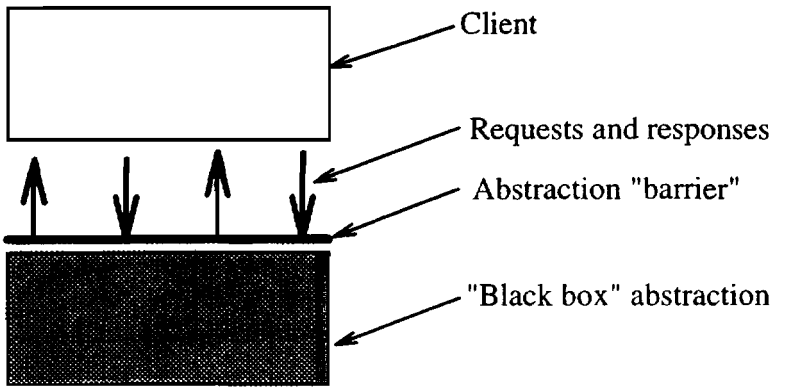

Figure 2.4: A traditional black box abstraction locks implementation details away behind an abstraction barrier.

to use an abstraction without understanding all the details that lie behind it. Second, *generalization* divorces the abstraction itself from any particular implementation, so the implementation may be changed without changing the abstraction (and hence, without forcing changes in its clients). By using separation and generalization in this way, systems can be modularized and their components reused. This model of abstraction runs throughout system design. The most basic elements of software systems, such as programming languages and instruction sets, are built upon it.

This notion of abstraction, as used in software development, is derived from the mathematical use of abstraction [Kru92]. However, software entities differ from mathematical entities. In software, the abstractions are not truly "abstract." Instead, they are the visible aspects of underlying implementations, and the implementation constrains the way in which the abstraction behaves. While any (correct) implementation of the abstraction will agree with the abstraction's specification, and hence operate in the "same" way, different implementation strategies will result in different performance characteristics, memory usage patterns, and so on. I shall use the term "behavior" to refer to the *manifestation* of these properties — that is, not just the semantics of the implementation, but also the details of its acceptable patterns of operation and performance.

Lists and arrays, for instance, are different implementations of a collection abstraction. Although they might share an interface, they exhibit different performance characteristics (different behavior). These particular examples happen to be so endemic to the problems we solve that we think of them as different abstractions; but the differences in their behavior are *not expressed* in the abstraction. This variability is something we often depend upon in implementation; for instance, caching in memory systems and memoization in programming language implementation are both techniques which *change* the performance characteristics while maintaining the *original* abstraction.

The same variability, though, can also introduce significant problems. To illustrate these problems, consider a situation opposite to that described above. Rather than one client and multiple possible implementations, consider a single implementation and multiple clients. This is a common arrangement: an operating system supports a text editor; an email reader and a database system; a window system supports a word processor, a spreadsheet and a game; the Lock Manager of an TP monitor supporting ACID transactions and extended transactions. The clients all make use of the same implementation, accessed through the same abstraction, in service of whatever functionality they themselves provide to their own clients. However, the clients have different needs. Consider the challenges that arise when we attempt to use the existing services of the Lock Manager for different transaction models. Some applications will need conventional read-write conflict behavior; some will require a *relaxed* definition of conflict using *operation semantics*; some will redefine conflict based on *transaction* or *application* semantics; others might require the ability to give up ownership of locks to restructure dynamically. In fact, the more clients there are, the more likely it is that there are going to be conflicts with their requirements for the behavior of the implementation. However, as observed above, the abstraction does not express the difference in behavior. In fact, those aspects of the implementation that would cause a programmer to choose one over another are systematically hidden by the abstraction barrier.

In this case, it's not the abstraction that is at fault. The simple specification of the abstraction (the transaction synchronization abstraction, defined in terms of the acquisition of locks and conflict detection) can be used effectively by all the clients. The problem lies, first, in the fact that the "abstraction" is not abstract at all, but is the interface to an implementation; and second, in the way in which a single implementation must serve multiple purposes. But this isn't some unusual special case; it's simply everyday reuse.

2.3.2 Mapping Dilemmas

The root of these problems can be explained in terms of mapping decisions, mapping conflicts and mapping dilemmas [KPng]. A mapping *decision* occurs when the implementor of an abstraction must choose between a number of possible strategies for implementing some internal mechanism. A mapping *conflict* occurs when some implementor makes the decision one way, but the needs of a client would be better met if the decision had been made another way. A mapping *dilemma* occurs when two clients of the same implementation require different mapping choices: whatever choice is made, a conflict results.

Mapping decisions arise not from the structure of the abstraction itself, but from the way in which it is implemented. Thus, since mapping decisions are not part of the abstraction, they are not visible through the abstraction barrier. While it is clear that

the incidence of mapping conflicts can be exacerbated by poor mapping decisions, it is important to recognize mapping dilemmas are not the result of particular implementations or abstractions, but are *inherent in the model of abstraction itself.* As such, software developers encounter them every day, and must employ some strategy to deal with them.

2.3.3 Gaining Control over Abstractions

As systems have become larger and more complex, and as hardware has improved and exposed more performance problems in software, strategies for overcoming these abstraction problems have become more common. One solution is to offer a number of different implementations to choose from (compiler optimization strategies often operate this way). Another is to provide switches that allow the application to select a particular strategy. For instance, the UNIX system call `madvise` allows application programmers to specify the style of memory access particular memory regions will experience, so that an effective paging strategy can be employed.

Recently, more radical solutions have been adopted, in various areas of system design. For example, the Mach operating system provides facilities for virtual memory behavior to be controlled directly by application programs — "external pagers" [RJY+88]. Scheduler activations [ABLL91] allow application control over thread facilities, addressing the design trade-offs involved in locating thread information and control in user space or kernel space. More generally, flexible object-oriented operating systems such as Spring [HK93] have allowed applications (or user-space code) a great deal of control over the implementation details of "lower-level" operating system abstractions [KN93, NKM93].

2.3.4 Open Implementation

Open Implementation (OI) is an approach to system architecture that "opens up" abstractions and provides clients with principled access to examine and control aspects of the implementation. The most important foundational principle behind Open Implementation is *computational reflection* [Smi82]. The reflection principle states that a system can embody a *causally-connected representation* of its own behavior, amenable to examination and change from within the system itself. The causal connection is a two-way relationship between the representation and the behavior it describes; this representation is maintained in correspondence with the system's behavior, and the behavior itself is controlled through manipulation of the representation. So, a reflective system can use the model to reason about its own behavior (*introspection*); and it can make changes to the model to effect changes in its behavior (*explicit control*). This causally connected self-representation creates a link between two "levels" of processing — the "base" level, which

is the traditional domain of computation for the given system, and the "metalevel" where the domain of computation is the system itself.

At the same time these new capabilities are introduced into a system, it is important to retain useful properties of the existing notion of abstraction, principally the conceptual simplification that it provides. There are two ways in which this is achieved in an Open Implementation, illustrated in Figure 2.5. First, a standard or default interface is available to access base services of the system, and a new *metalevel interface* is provided to access the causally-connected representation: the interface to the metalevel *augments* the traditional abstraction barrier, rather than replacing it. Second, the view into the implementation reveals its *inherent* structure and function, rather than the details of the specific implementation. It does not simply provide a set of "hooks" directly into the implementation; that would both constrain the implementor of the abstraction and require too much of the implementor of a client. Instead, it provides a rationalized model of the inherent behavior of the system offering its particular functionality.

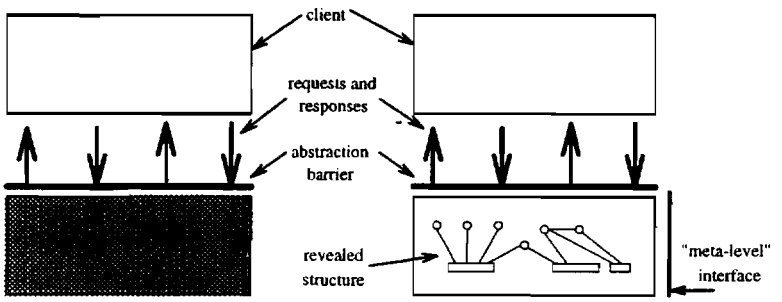

Figure 2.5: Black box abstraction contrasted with open implementation.

2.3.5 Designing an Open Implementation for an TP Monitor

We now consider how to apply the Open Implementation approach to design a framework for implementing extended transactions on a conventional TP monitor. To date, there have been few systems designed following the Open Implementation approach, and no one has applied it to extend an existing software system. There are, however, examples of Open Implementation-style concepts in otherwise traditional systems, such as operating systems [Yok92, ABLL91, PAB+95], composable microprotocols [BS95, BD95, EPT95], and external paging facilities [RJY+88]. On the basis of these experiences, what has emerged is not so much a process for Open Implementation design, but more a set of design principles. Emerging work on Open Implementation Analysis and Design (OIA/D) [KDLM95] represents an early attempt to draw out these principles.

In what follows, I shall describe a number of these principles, drawn from the Open Implementation literature, that shaped the design of the Reflective Transaction Framework. While each design principle is presented separately, it will become clear that they are strongly related to each other. First, however, a digression regarding reflective self-representations will provide some context.

Reflection and Self-Representation

When thinking about self-representations in reflective systems, it's important to bear in mind that they are just that — *representations.* The causal connection, in particular the computational effectiveness it supports, can lead to confusion between the representation and the mechanism that is represented. Similarly, the metaphorical relationship drawn between reflective systems and mechanical ones – in which mechanism is "exposed to view," and users can "reach in" to effect changes – can also contribute to this confusion. When thinking of the design of a reflective system, there are two important aspects of the representation *qua representation* to be considered: *maintenance* and *partiality.*

Maintenance refers to the way in which the representation is actively maintained by the reflective system. Elements of the representation can be created as needed, and/or maintained in correspondence with elements of the system itself, rather than being continually present. The lazily-created reflective interpreter layers of the 3-Lisp implementation [dRS84] illustrates this. While the 3-Lisp model guarantees the representation is available when requested, it may not actually exist *until* requested. At the point it is created, the elements of the representation (or rather, an instance of the representation) are a rationalization of the system's state according to an idealized model. So, when designing the Reflective Transaction Framework and considering the terms in which the metalevel interface is cast, it is important to remember the distinction between "exposed structure" and *actual* implementation mechanisms, a distinction the system must actively maintain.

Another design principle that follows from the maintenance of the representation is its inherent *partiality.* The purpose of the representation is not to provide an absolute, decontextualized or impartial description of the system's activity. Rather, the representation describes *selected* aspects of the system's behavior for the purposes of some domain of expected behavior [Kic92]. It reveals certain aspects of behavior, and hides others; similarly, it supports certain forms of tailoring and modification, but not others. The representation is a *designed* artifact; and, in line with perceived needs and expectations, we, as the designers, set the bounds on the flexibility it embodies. The representation, then, is guided more by expectations of use than it is by the structure of the implementation.

Scope Control

A critical design property is *scope control*, the ability to restrict attention (and changes) to a particular set of objects. The ability to maintain and manipulate different scopes not only sets up protection boundaries, but also allows for different behaviors to be mixed together in a single system [Yok92].

In CLOS, the Common Lisp Object System [BGW93], scope control is achieved through the class/metaclass mechanism. Since class behaviors are encapsulated by metaclasses, new behaviors are introduced into only those classes that specify a modified metaclass. Introducing a change to slot access or method dispatch in CLOS will not affect every class in the system. The metaclass mechanism bounds the effect of the change, restricting its scope. At the same time, it allows multiple behaviors to coexist. While a change in the slot access mechanism can be introduced for a new metaclass, the default behavior exists alongside it, associated with the original metaclass. Indeed, any number of new behaviors might be introduced, and the scope control introduced by the metaclass namespace allows them to co-exist without interference. A similar approach is used in Silica [Rao91], through the use of specific "contracts" between types of windows and their subwindows.

The ability to name and distinguish between sets of alternative behaviors is an important factor in maintaining scope control. It is also critical that the groupings and categories to which these behaviors can be applied are at an appropriate level of granularity. For example, in CLOS, it would be unwieldy to have to discuss metaclass-level behaviors individually for each object, or to have to talk about all classes at once. CLOS associates these behaviors with classes, which are a convenient unit of scope for the flexibility that CLOS provides. In the design of the Reflective Transaction Framework, the convenient unit of scope is likely to be individual transactions within an application. Scope control would then establish boundaries between different extensions to transaction services, and would also provide a mechanism for bounding the effects of changes applied to specific extended transactions within an application.

Conceptual Separation

Another design property is the separation of *conceptual concerns* expressed by the metalevel interface. Again, this is essentially a scoping issue, but of a different sort: scope control addresses *which application objects* will be affected by a particular change, while conceptual separation is concerned with the *extent of the behaviors* that are affected.

A metalevel interface can express a range of different behaviors and present many aspects of the system's internals; the principle of conceptual separation states that the separation between different aspects of internal behavior should be expressed in a similar

separation between those aspects of the interface used to control them. So, it should be possible to introduce a change in one aspect of the system's behavior, relatively independent of the other aspects that the metalevel interface may control. Similarly, it should be possible to do this using only specific aspects of the metalevel interface relevant for that concern, without having to bring in (or even understand) the other areas. Simple changes or extensions to the base transaction system should be simple to introduce.

Conceptual separation, perhaps more than the other design principles, highlights the fact that the metalevel interface is designed to support a particular range of behaviors, based on the designer's expectations. The separation of concerns in the metalevel interface provided by the Reflective Transaction Framework will reflect our assumptions and expectations about the transaction behaviors that will be tailored independently.

Incrementality

Another design property often discussed in the Open Implementation literature is *incrementality*, which deals with the ways in which changes introduced into the system relate to, and build upon, existing or default behaviors. The provision of a metalevel interface, and thereby a means to change the system and adapt it to particular needs and circumstances, does not relieve the system designer of the burden of designing a good base-level system. Open Implementations are intended to be usable; the metalevel interface is an added facility that many clients will not use.

The default behavior serves two ends. First, it provides the standard functionality of the system. It should be usable in a normal range of circumstances, without any appeal to the metalevel interface. Second, when the metalevel interface is used to introduce changes, the default behavior should be the basis for reuse. Incrementality concerns this second use of default behaviors. It states that it should be possible to introduce new behaviors by incrementally extending old ones, specifying what is new and different relative to the original behavior. Thus a programmer using the Reflective Transaction Framework should not have to recraft transaction behavior from scratch, but rather use the default ACID transaction behavior as a baseline. So, default transaction behavior is provided not only as a usable system in its own right, but also as the basis for redefinition and extension to implement extended transactions; that is, ACID transactions are both the default and basis for changes made at the metalevel.

2.4 Summary

This chapter presented the technical background for our work on the Reflective Transaction Framework. First, to understand the functional requirements of implementing extended transactions, a number of advanced transaction models and semantics-based concurrency control protocols were presented, and we identified three common extended services. Following this, we reviewed related implementation efforts and identified how they incorporated these extended services into their designs. We concluded the first section of the chapter with a discussion outlining our strategy for building the Reflective Transaction Framework on top of transaction services provided by conventional TP monitor software. Our approach is aimed at keeping the conventional TP monitor and ACID transactions running while incrementally adding extended transaction functionality.

In the second part of the chapter we presented an overview of the TP monitor architecture, along with a brief discussion of extending existing functionality to implement extended transactions. We argued that TP monitor software provides a useful substrate for implementing extended transactions, but gives programmers only limited control over the ways in which the transaction mechanisms will support their applications (and hence, limiting the range of transaction services that the system can support). In making a set of structures, behaviors, and mechanisms available to application programmers, TP monitor implementations also make a set of commitments to particular styles of application and interaction. So the traditional model of abstraction in modern TP monitor system design, which is meant to support the reuse of implementations, is actually getting in the way of reuse for implementing extended transactions.

Finally, in the third part of the chapter, we drew on ongoing work in the use of computational reflection and Open Implementation for guidance in designing the Reflective Transaction Framework. The Open Implementation approach provides a new way of thinking about the relationships between a client, the abstraction the client is using, and the implementation that realizes the abstraction. Drawing on computational reflection as a way of relating the abstraction and the implementation, Open Implementations provide clients not only with abstractions that they can use, but also with the means to examine and manipulate those abstractions. Using these facilities, applications can become involved in how the infrastructure supports their operation, and so can tailor the services of transaction system abstractions to their own particular needs.

Chapter 3

Reflective Transaction Framework

In this chapter we present the Reflective Transaction Framework. In Section 3.1 we first outline our design objectives and put forth the specific extensions provided by the Reflective Transaction Framework to implement extended transactions. In Section 3.2 we present the framework architecture, which constructs extended transaction services as a collection of transparent extensions to an existing TP monitor, and we discuss the computational model of the framework, which provides an open implementation of the underlying TP monitor. Finally, in Section 3.3 we present the detailed design of extended transaction services supported by the Reflective Transaction Framework.

3.1 Framework Design

We commence our design description of the Reflective Transaction Framework with a brief discussion of the main objectives. We present these objectives before describing the Reflective Transaction Framework because this discussion clarifies key rationales and justifies important design features.

3.1.1 Objectives

The primary objectives of this research were to define a software framework to support extended transactions and develop a practical implementation of the framework. Practicality was an overriding constraint in the definition of the framework, and it translated into the following specific design objectives: support for key extended transaction services, ease of implementation, compatibility with legacy transactional applications, ease of use, and acceptable overall performance. We elaborate on each of these as follows.

Key Transaction Functionality The ultimate goal of the Reflective Transaction Framework is to support the implementation of extended transactions. Therefore, the framework must provide extended transaction services sufficient to implement a wide range of advanced transaction models and semantics-based concurrency control protocols from the literature.

Ease of Implementation An important goal is that the Reflective Transaction Framework be designed for ease of implementation. We recognize that conventional transaction processing systems, in particular TP monitors, have already accumulated large amounts of implementation technology. We don't think that is clever to throw it away and attempt to build an extended transaction facility from scratch. TP monitors provide basic mechanisms such as lock-based concurrency control, logging and recovery services, and transaction management services. Therefore, we decided to leverage existing transaction processing functionality and structures in constructing the extended transaction services. This not only eliminates unnecessary infrastructure development but provides efficient, robust *base processing* for extended transactions.

Compatibility with Legacy Applications Maintaining compatibility with ACID transactions is a major priority. Legacy applications are here to stay; we must ensure that the behavior of ACID transactional applications remain unchanged when the services of the Reflective Transaction Framework are not involved. In addition, existing ACID applications should be able to exploit the Reflective Transaction Framework services with little change.

Ease of Use For programmers, we pursue two complementary goals of conceptual simplicity and access flexibility. The framework functionality must be presented to both transaction system programmers and application programmers through a simple abstraction that is easily understood and fully compatible with the traditional ACID application paradigm; programmers should not have to bend over backwards to achieve desired effects. Moreover, the mechanisms through which the framework services are accessed must be flexible and easy to use.

Acceptable Overall Performance Finally, it is widely recognized that good performance is an intrinsic aspect of system usability. Excessive application performance degradation would seriously undermine the usability of the Reflective Transaction Framework. While we accept that the extended transaction services and mechanisms will incur a certain amount of overhead, it is imperative that we seek good overall system performance in our design and implementation.

3.1.2 Focus on Specific Extensions

We limit the scope of this research by focusing on three specific extensions for implementing extended transactions: *dynamic transaction restructuring*, *semantic transaction synchronization*, and *transaction execution control*. The detailed design of each extended service is presented later in this chapter, in Section 3.3. There are two reasons for this narrowing of focus. The first is purely practical – any effort must start somewhere. These extensions provide a starting point to illustrate the application of the Open Implementation approach to a conventional TP monitor and demonstrate the gains that result, without having to open up absolutely everything in the TP monitor. The second reason is that these extensions offer the greatest leverage. As we saw in our background discussion on extended transactions in Chapter 2, these extensions are the common *dimensions of change* found in most extended transactions in the literature. Addressing the requirements of dynamic transaction restructuring, semantic transaction synchronization, and execution control, provides a base for implementing most extended transaction behaviors. While it may be tempting to design a facility that is richer in functionality, we feel such sophistication would come at the cost of runtime efficiency, ease of use, and more onerous programming constraints. Thus, one can view this decision as an exercise in minimalism. Instead of conjuring up all the extended transaction features we would like to include in the framework, we have determined what can be omitted while still being able to implement a number of extended transactions.

3.1.3 Design Summary

In this section we have presented the main considerations for the framework design. As subsequent design and implementation trade-off analysis will demonstrate, the specific objectives of extended transaction services, ease of implementation, ACID compatibility, ease of use, and reasonable overall performance often create competing demands at both design and implementation levels. Our approach is to balance these concerns and make the necessary compromises that best serve the ultimate purpose of practical usability.

3.2 Architecture

This section presents the architecture of the Reflective Transaction Framework. We first discuss the system architecture, which is decomposed into a collection of software modules called *transaction adapters.* These adapters expose selected functions and data structures of the underlying TP monitor and implement specific extended transaction services. We then describe the computational model, which builds extended transaction services as an extensible collection of transparent extensions to existing TP monitor functionality. In particular, we describe the role transaction adapters play in constructing an effective Open Implementation for the underlying TP monitor, and explain how framework extensions are coupled to the underlying TP monitor through *transaction significant events.* Throughout this section we identify framework interfaces that enable programmers both to implement extended transactions and develop applications using extended transactions, and we try to explain relevant mechanisms from the user's perspective as much as possible.

3.2.1 System Components

Figure 3.1 illustrates the major components and interfaces defined by the Reflective Transaction Framework. The framework is a layered architecture, designed to be implemented over an existing TP monitor. Reflective software modules, called *transaction adapters*, correspond to a particular functional aspect of the TP monitor, such as transaction execution, lock management and transaction conflict. Transaction adapters invoke services of the underlying TP monitor through "down" calls using the TP monitor service API, while functional components of the TP monitor pass state information and request extended transaction services from the transaction adapter layer through "up" calls or callbacks.

A transactional application program is linked to one or more libraries, labelled *RTF Library* in Figure 3.1, which provide a collection of extended transaction functions. Each function in this library is model-specific, implemented by a transaction system programmer familiar with the semantics of the advanced transaction model. Applications invoke functions in this library to access extended transaction functionality provided by the Reflective Transaction Framework in an TP monitor-independent manner. As we shall describe later in this section, the Reflective Transaction Framework manages communication between a transactional application and these RTF Libraries both to simplify the calling of functions and to guard against improper usage.

The layered architecture in Figure 3.1 does not specify how the transaction adapters in the framework should be connected to the underlying TP monitor to produce a working system. If the software modules that implement the transaction adapters were each in their own operating system process, then the inter-layer calls might require an RPC

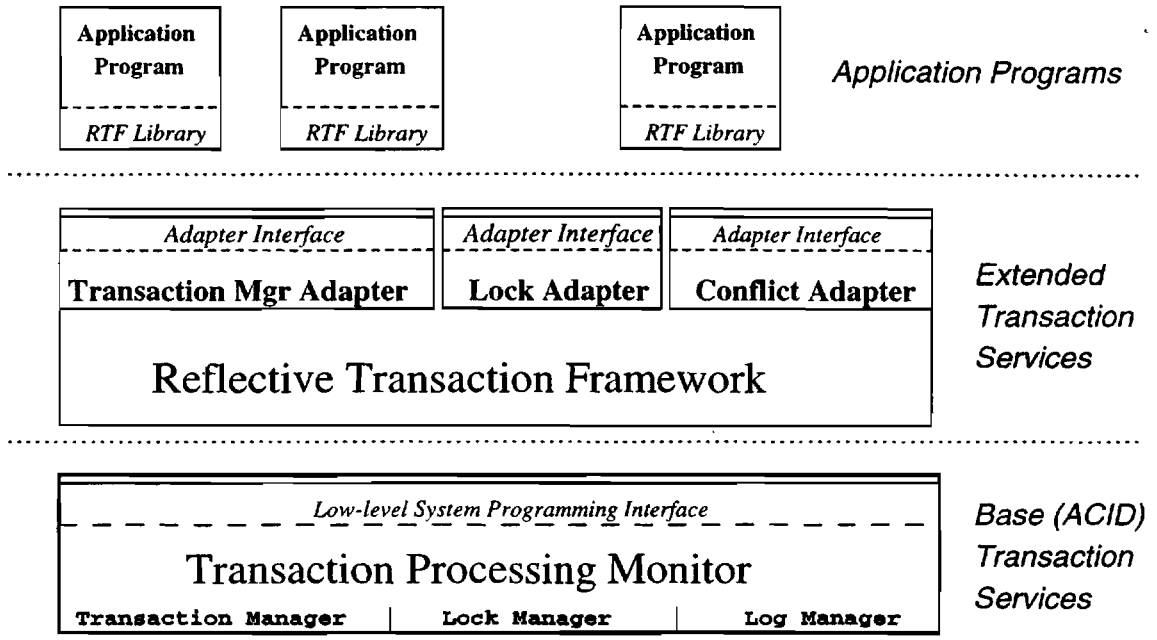

Figure 3.1: Major components and interfaces of Reflective Transaction Framework.

(Remote Procedure Call) mechanism, or perhaps a specially designed IPC (Inter-Process Communication) layer. If the transaction adapters that make up the framework were each built into the same operating system executable as the application program, then the inter-layer function calls between an application program and the framework services would be more efficient. Alternatively, the transaction adapters in the Reflective Transaction Framework could be integrated with the operating system process that executes the TP monitor. We shall revisit these options later in Chapter 5, when we present the Encina implementation of the Reflective Transaction Framework.

Transaction adapters are designed to provide principled access to selected functions and data structures of a particular functional component of the underlying TP monitor, and *augment* the basic transaction services it provides with a set of *extended transaction services.* Table 3.1 summarizes the Reflective Transaction Framework's initial set of transaction adapters, identifying state and extended transaction services that each provides. Other transaction adapters for extended transaction recovery, workflow management, and distributed extended transaction management are possible in the future.

Table 3.1: Mapping extended transaction services to transaction adapters.

	ADAPTER EXPOSES	EXTENDED SERVICES
TRANSACTION ADAPTER	Transaction State Execution control	Extended Transaction State Transaction Significant Events Intra-Transaction Dependencies
LOCK ADAPTER	Explicit Lock Control Lock Table Information	Lock Sharing Lock Delegation
CONFLICT ADAPTER	Lock Table State Lock Conflicts	Semantic Conflict Definition Explicit Cooperation

Three major arguments justify the functional partitioning of the Reflective Transaction Framework into separate transaction adapters. The first argument is *scope control.* Each adapter encapsulates a set of extended transaction services that augment the base services of a particular functional component in the TP monitor. By factoring extended transaction functionality into separate transaction adapters, we can isolate these functional extensions of the TP monitor. The second argument is *conceptual separation.* This is a scoping issue, but of a different sort. Conceptual separation is concerned with the *extent of the behaviors* that are affected. Each adapter implements a specific extended transaction service and provides an interface that expresses the range of different behaviors that it can support. By selecting particular adapters and invoking their interfaces to customize their behavior, adapters can be used in combination to create a different Reflective Transaction Framework configuration. The third argument is *incrementality.* It is essential to design the Reflective Transaction Framework for incremental extension. One aspect of extended transaction semantics can be modified in transaction adapter without affecting other services (transaction adapters) in the framework. For example, we could extend the TRANSACTION MANAGEMENT ADAPTER to support a richer set of transaction dependencies without having to modify the code of other adapters in the framework.

3.2.2 A Separation of Programming Interfaces

The Reflective Transaction Framework defines two new interfaces (sets of APIs) corresponding to two levels of understanding of transaction management. The purpose is to support two categories of programmers: transaction system programmers with skills in transaction model specification who implement primitives for new extended transactions, and application developers who program transactional applications using the available extended transaction primitives.

Application developers program transactional applications using a set of transaction model-specific verbs, or *transaction control operations.* For example, ACID transactions

are typically initiated by the operation `Begin-Transaction` and terminated by either a `Commit-Transaction` or `Abort-Transaction` operation. Extended transactions often introduce additional operations to control their execution, such as the operations `Split` and `Join` introduced in the split/join transaction model, or the operation `Join-Group` introduced in the cooperative group model. A transaction model defines not only the control operations available to a transaction, but the semantics of these operations. For example, whereas the `Commit-Transaction` operation of the atomic transaction model implies that the transaction is terminating successfully and that its effects on data objects should be made permanent in the database, the `Commit-Transaction` operation for a member transaction in a cooperative transaction group merely implies that its effects on data objects be made persistent and visible to transactions that belong to the same cooperative group.

To accommodate this diversity of interface and operation semantics between different advanced transaction models, we introduce a separation of programming interfaces, presented figuratively in Figure 3.2. Both the *base interface* and *extended transaction interface* are used for application-level programming, subdivided for clarity only, while the *metalevel interface* is used to introduce new extended transaction control operations and to define their semantics (implementation).

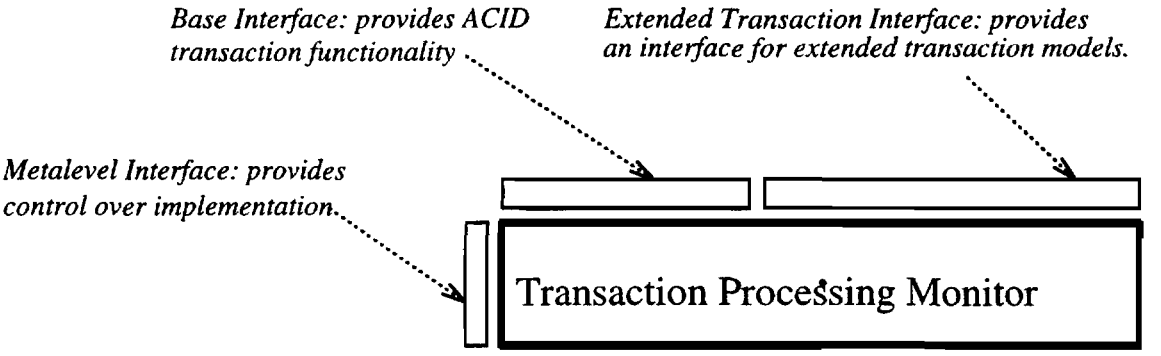

Figure 3.2: Separation of interfaces to Reflective Transaction Framework.

The *extended transaction interface* provides application programmers with a functional view of extended transaction management. It is intended for programmers who understand how to use the control operations of the extended transaction model(s) best suited for their application. They are responsible for the implementation of the transaction-aware portion of the application, which should account for only a small portion of the application code. This extended transaction-aware code will typically identify transactions that require extended services, select a specific model for the transaction, and then invoke control operations specific to the extended transaction model(s) selected (e.g., `split` and

`join` for the split/join transaction model). Similarly, the *base interface* provides conventional transaction control operations for ACID transactions that do not require extended services. These default control operations are implemented by the underlying TP monitor and typically include the operations: `Begin-Transaction`, `Commit-Transaction` and `Abort-Transaction`. Hence, with the exception of identifying transactions that require extended services and selecting the appropriate model, there is no discernible difference from ordinary transactional application development.

The *metalevel interface* provides an implementation view of extended transaction management. This interface concerns the transaction system programmer who wishes to augment the set of available transaction models to satisfy new application requirements. The metalevel interface consists of *building blocks* that may be used to implement a specialization of an existing control operation, such as `Commit-Transaction` operation for a member transaction in a cooperative group, or to introduce new extended control operations, such as `split` and `join`. The building blocks for implementing extended control operations are the extended transaction services provided by transaction adapters and functionality of the underlying TP monitor that the adapters expose.

When the need arises, new extended transaction behaviors can be defined using the metalevel interface and made available to application developers through the introduction of control operations in the extended transaction interface; the extended transaction interface *augments* the default transaction interface. This separation of programming interfaces provides the means of both introducing new extended transaction behaviors and interfaces, and developing transactional applications using these new extended transaction operations in a manner that does not deviate significantly from "normal" transactional application programming.

3.2.3 Open Implementation of an TP Monitor

From the Open Implementation perspective, transaction adapters present three kinds of opening to the underlying TP monitor on which to build extended transaction functionality. Each opening serves a different purpose and has its own set of operations. In this section we describe the purpose for each opening, how it is realized, and the operations that each provides.

Introspection

The first opening of the underlying TP monitor is through *introspection*. It involves the *reification* of selected aspects of an executing transaction's internal information, such as execution state, transaction dependencies, lock conflict, and transaction relationships into

a structure called an *extended transaction descriptor.* Every extended transaction has an extended transaction descriptor, which applications can use to examine reified state information. Table 3.2 lists attributes of the extended transaction descriptor.

Table 3.2: Attributes present in the descriptor for an extended transaction.

Attribute	*Description*
ETRID	Unique extended transaction identifier, assigned by the RTF.
TRID	Unique transaction identifier, assigned by the TP monitor.
NAME	Unique name for the extended transaction, assigned by the application.
STATE	Extended transaction state: one of *Initiated*, *Active*, *Pending*, *Committed*, *Aborted*, or *Terminated*.
TRANEVENTS	Extended transaction management events. Represented as a list of event descriptor structures.
EVENTHISTORY	Ordered list of management events that have been executed, recorded as a tuple in the form <event descriptor, timestamp>.
TYPE	Transaction type (optional), assigned by application program.
INTERNALSTATE	Internal transaction state (optional), assigned by application program.
PROPERTYLIST	Transaction properties (optional), assigned by application program.
DELEGATE_ENABLED	Indicates whether the transaction can delegate locks.
ACQUIRE_ENABLED	Indicates whether the transaction can acquire delegated locks.
DELEGATESETLIST	List of delegate sets owned by this transaction.
SBCC_ENABLED	Indicates whether the transaction can relax R/W conflicts.
SBCC_POLICY	Specifies the order in which to apply semantic compatibility definitions.
COMPATIBILITYTABLES	List of semantic compatibility tables loaded by the application.
COOPTRANSET	Table of active ignore-conflict relationships.
DEPENDENCY_ENABLED	Indicates whether the transaction is permitted to form dependencies.
DEPENDSET	Table of active transaction dependencies.

To *register* a transaction with the Reflective Transaction Framework and create an extended transaction descriptor for the transaction, an application uses the `instantiate` command. When *instantiated*, a descriptor is created and the transaction is assigned an *extended transaction identifier* (etrid) that uniquely identifies the extended transaction and can be used to access its corresponding descriptor.

There is an *external state* attached to each extended transaction. Typically, an extended transaction is in one of the states INITIATED, ACTIVE, PENDING, COMMITTED, ABORTED, or TERMINATED. The external state of an extended transaction is set to INITIATED when its descriptor is created. An extended transaction is ACTIVE if it has been initiated by an initiation event, such as a `Begin-Transaction`, and has not yet executed

one of the termination events associated with it. Eventually, the extended transaction will either abort and move to the ABORTED state, or move to the PENDING state by issuing a prepare operation. From the PENDING state, an extended transaction can either commit (i.e., make the COMMITTED transition), or abort (i.e., make the ABORTED transition). If the application wishes to express the fact that a transaction is no longer active, irrespective of whether it aborted or committed, we refer to the state as TERMINATED.

The descriptors for all *active* extended transactions are stored in an *extended transaction table*, complementary to the transaction table managed by the TP monitor. The transaction manager of the TP monitor creates an entry in the transaction table to record the TRID of an executing transaction along with other pertinent information, and to track the transaction through its execution. However, while every active transaction in the TP monitor will have an entry in the transaction table, only extended transactions have an entry in the extended transaction table. Data stored in both the transaction table entry and the extended transaction table entry permit bidirectional access to the information stored in these tables.

Operation Definition 3.1 (instantiate) *The operation* `instantiate(tran_name,` $\texttt{TRID}_{t1}$*)* *creates an extended transaction descriptor and a unique extended transaction identifier (etrid) for* t_1*. Both the transaction identifier (*$TRID_{t1}$*) and transaction name (tran_name) are stored in the descriptor, along with the etrid. The state of the extended transaction is set to* INITIATED *and the descriptor is entered into the extended transaction table. The* `instantiate` *operation returns either the etrid value indicating success or an error code.*

The reification of state information for an active extended transaction is implemented using *callbacks*. Callbacks support efficient cross-layer communications and enable the TP monitor to pass state information to the adapters in the Reflective Transaction Framework. Callbacks are associated with *significant events*, such as a transaction attempting to change state (e.g., the transaction begins, aborts or commits) or a transaction requesting a service from the TP monitor. For each transaction event there is an associated callback that can be called *before* and *after* the event. If a function is registered with a callback and the event is raised during transaction processing, execution control is passed to the function, along with all information relating to the event. For example, when a transaction attempts to commit, an event is raised and control passed to the TRANSACTION MANAGEMENT ADAPTER. The adapter can perform commit pre-processing functions, such as checking for termination dependencies that might exist with other extended transactions, then update the extended transaction descriptor. Once the extended transaction descriptor has been updated and processing for the event is complete, the adapter will then return execution control to the TP monitor for normal processing.

The most important decisions made in designing the introspective capability involve selecting aspects of the underlying TP monitor component that should be reified. We systematically identified the aspects required to implement extended transactions, by first identifying the state required for each extended transaction service and then defining a callback to pass this information on to the appropriate transaction adapter.

Introspection provides programmers with a principled way of examining selected implementation state. The interface is principled in the sense that it allows access to this state information without forcing the transaction processing system implementation to expose the internal data structures they actually use to represent it. An application can use this representation to reason about the transaction system and to implement utilities such as an application monitor or browser, a trigger facility, or to compile program statistics. However, a programmer cannot *yet* change how the underlying TP monitor behaves. The next opening begins to provide that additional power.

Table 3.3: Commands to inspect and modify an extended transaction descriptor.

Command	*Description*
instantiate(name, trid)	Generate an extended transaction identifier (etrid) and create an extended transaction descriptor for the transaction, storing both name and trid in the structure.
getetrid_using_name(name)	Returns the etrid value of the extended transaction descriptor identified by the string *name*.
getetrid_using_trid(trid)	Returns the etrid value of the extended transaction descriptor identified by the value *trid* assigned by the TP monitor.
getname_using_etrid(etrid)	Returns the name of the extended transaction descriptor identified by *etrid*.
getname_using_trid(trid)	Returns the name of extended transaction identified by trid.
gettrid_using_name(name)	Returns the trid of extended transaction identified by *name*.
gettrid_using_etrid(etrid)	Returns the trid of extended transaction identified by *etrid*.
transtate(etrid)	Returns the extended transaction state – *initiated*, *active*, *pending*, *committed*, *aborted*, or *terminated*.
set_type(name, val)	Sets the (optional) type of the extended transaction identified by name to the assigned value.
get_type(name)	Returns the type of the extended transaction identified by name.
set_state(name, val)	Sets the (optional) *state* value of the extended transaction identified by the input argument *name* to the value supplied as input.

continued on next page

continued from previous page	
Command	*Description*
get_state(name)	Returns the *state* of extended transaction identified by name.
set_etranprop(etrid, key, val)	Sets the value of the property list identified by *key* for extended transaction *etrid* to the supplied value *val.*
get_etranprop(etrid, key)	Returns the value of *key* for the extended transaction *etrid.*
setp(etrid, switch, val)	Sets the value of *switch* for extended transaction *etrid* to the supplied value *val.*
getp(etrid, switch)	Returns the value of *switch* for extended transaction *etrid.*
record_event(etrid, desc, tstamp)	Record that a transaction management event has been executed by appending the entry (description,timestamp) to the field eventHistory.
find_event(etrid, desc, tstamp)	Search the eventHistory field for an entry matching the input descriptor, beginning with the first entry after *tstamp.* If found, then return the value of the timestamp (found) or the value 0 (not found).

Explicit Invocation

A transaction system hides not just the *state* that would be useful for an application to have access to, but also pieces of *functionality* inherently present in every transaction system that would be useful if exposed. For example, the ability for an application to explicitly acquire a lock on a data object or to release a lock held by a transaction. The second kind of opening that transaction adapters provide is called *explicit invocation*, which is the ability of an application to invoke existing functions of the underlying transaction processing system directly, without going through the ordinary transaction system interface.

Explicit invocation is implemented by linking transaction adapters to the functional components of the underlying TP monitor. Applications can directly invoke TP monitor functions through the API presented by transaction adapters. For example, an application can query the LOCK ADAPTER for the list of locks held by an extended transaction, then release locks on selected data objects. The most challenging issue in implementing explicit invocation is to identify the appropriate interface to expose these new capabilities. Extra care may be required to avoid introducing new failure modes, but runtime usage checking can be performed to avoid such failures. Ideally, this task of identifying the appropriate API calls and linking the transaction adapters to the TP monitor is performed only once during the implementation of the framework, by someone familiar with the underlying TP monitor. Once complete, each transaction adapter in the framework not only reifies selected aspects of the underlying transaction system, enabling introspection, but now provides the means to affect the state and control behavior of active transactions.

So far, the cost to transaction system implementors has been modest. They have been asked only to expose information and functionality that is inherently part of any transaction processing system. In a sense, the new functionality that introspection and explicit invocation offer has *"been there all along."* We now consider a new challenge, in which application programmers want more than enhanced access to what is already there. Instead, they require some *additional* or *extended* transaction functionality for their application. For example, an application may wish to use SPLIT or JOIN to restructure transactions dynamically, or to redefine the notion of operation conflict. This is where the decision to represent each extended transaction as a metalevel object, an extended transaction descriptor, will come into its own. An extended transaction descriptor makes it possible to ensure that when a programmer changes or extends a transaction's behavior, it will have an appropriately localized effect – *they provide scope control.*

Intercession – customizing transaction behavior

The third opening that transaction adapters provide is called *intercession*, and it allows programmers to introduce extensions into a transaction processing system. Intercession is qualitatively different from the two previous openings. Intercession builds on the introspection and explicit invocation capabilities of the framework to extend the processing of transaction significant events in a controlled manner.

Intercession is implemented, in part, through *transaction events*, which are "hooks" onto which applications can attach their extensions. Events are generally recognized as an effective technique for implementing loosely-coupled, flexible systems where relationships between code components can be dynamically established [SN92]. In the Reflective Transaction Framework, a transaction event can be passed to an *event handler*, which is code that is executed in response to a specific event. In the framework, every transaction management primitive, such as BEGIN, SPLIT, JOIN, COMMIT, ABORT, etc., represents an event, as does a transaction changing state (to ACTIVE, ABORTED, COMMITTED, etc.) or requesting a service (e.g., lock request) from the TP monitor. Consequently, all relationships between a transaction and TP monitor are subject to change simply by changing the handler associated with a given transaction event. The binding between a transaction event and corresponding handler is captured in an *event descriptor*, depicted in Figure 3.3. The event descriptor identifies the name of a transaction event, provides a function pointer to the handler that is to be invoked when the event is raised, and records other information, such as guards (predicates) that are to be evaluated prior to invoking the handler and properties for event execution control.

An extended transaction can *own* multiple event descriptors – there is a descriptor for each event that has been extended. Event descriptors for an extended transaction are

```
TYPE
  event_type: STRUCT;
    EVENTNAME: char*;
    GUARDS: list of char*;
    HANDLER: ptrfHandler;
    ATTRIBUTE: enumerated type, one of ''normal'', ''inevitable'' or ''immediate'';
    TRIGGERABLE: boolean;
  end; (* event_type *)
```

Figure 3.3: Basic structure for representing a transaction event.

stored in the *tranEvents* field of the associated extended transaction descriptor (refer to Table 3.2). Event descriptors enable the framework to bind an extension to an extended transaction seamlessly, so that applications see the original (or expected) behavior and interface, unless the handler requires the application to be informed about some exception (for example, an error message returned for lack of access rights). The actual invocation of the handler is hidden. This is accomplished by linking the application program to the transaction adapters in the framework, which will trap all control operations, such as BEGIN, COMMIT, SPLIT, etc., and transaction system events, such as lock requests, lock conflicts, transaction initiations and terminations, etc. After detecting an event, the framework first locates the corresponding event descriptor in the extended transaction descriptor, then passes the arguments to the specified event handler. In the next section we describe how transaction adapters actually bind a transaction significant event to the function that implements the handler, but first we identify measures that can be taken to ensure that these extensions do not corrupt the transaction processing system.

What we have implied throughout our discussion is that when applications use transaction adapters, their behavior must be moderated by "rules of behavior," as is customary for software engineering in general. The capabilities available to programmers through introspection, explicit invocation, and intercession potentially allow private transaction information to be accessed and system behavior to be altered inappropriately. To control the set of operations that an application can invoke through the metalevel interface, the Reflective Transaction Framework uses *guards*. Guards encode the rules of behavior for accessing metalevel interface operations and processing transaction events.

Each event in the Reflective Transaction Framework can have an associated guard that identifies a predicate to evaluate prior to invoking the handler. If the predicate is true when the event is raised, the handler is invoked; otherwise, the event will be delayed or

rejected. For example, when an application calls COMMIT for an extended transaction, an event is raised and the framework evaluates any guards that are in place before calling the handler assigned for commit processing. If, for example, the transaction had established a dynamic commit-dependency during execution, a guard could be written to verify that the dependent transaction has been committed and, if not, block the execution of the handler until the dependency is eliminated. In this manner, transaction adapters can guard events on a per-transaction basis, separating the specification of *what* should happen from *when* it should happen for each extended transaction. Thus, while the extensions in the Reflective Transaction Framework define the function that is to occur in response to a transaction event, guards ensure that this function is executed only at the proper time.

3.2.4 Binding Extensions to Transaction Significant Events

The current design of the Reflective Transaction Framework makes only one aspect of the underlying TP monitor reflective, namely *transaction significant event* processing. The basic idea is that transaction event invocation can be intercepted by the framework and passed to a corresponding handler. In this way, transaction systems programmers can make significant events behave according to a particular extended transaction model through the implementation of a model-specific handler. The extended transaction descriptor, specifically the *event descriptor* field, describes how to deal with the invocation – it identifies the event and handler that is to be invoked when the event is raised.

Handlers for significant events are implemented as functions in an RTF Library (refer to Figure 3.1). Each function is specific to an extended transaction model; for example, there might be a handler for the control operation `join` of a member transaction in a cooperative group, as well as handlers for the control operations `split` and `join` of a Split/Join transaction. This immediately raises a technical difficulty: how can an application call the same function for different extended transactions but have it execute different code? Any developer knows that if an application defines a function twice, the linker will generate an error saying something like, "Duplicate symbol defined: *function name.*"

The framework could solve this problem by simply supplying the handlers in RTF Libraries to application programmers, but consider what would happen if an application attempted to call the handlers (functions) in an RTF Library directly. Unless the application was linked directly to a particular RTF Library, it would have to build a table of pointers to the handlers in that RTF Library and call those handlers by pointer. Using the same code for more than one RTF Library at a time would add yet another level of complexity. The application would first have to set a function pointer to point to the correct handler in the correct RTF Library, and then call the handler through that pointer. Exposing RTF Libraries clearly introduces new complexities for application development.

It forces application programmers to be aware of the contents and organization of the RTF Libraries, and to understand the functionality and differences of available handlers, as well as creating and managing function pointers to the required handlers.

The Reflective Transaction Framework solves this problem by providing a single place for an application to call each transaction control operation – the *extended transaction interface.* The application is linked to the transaction adapters in the framework and calls extended transaction control functions exported by the extended transaction interface, not the functions implemented in the RTF Library. The application identifies the extended transaction for which it is making the call, either *explicitly* by passing the name of the extended transaction with the call or *implicitly* by virtue of a call attribute. For example, the framework can identify the extended transaction using a common TP monitor function that performs a thread-to-trid mapping; and using the trid, the framework can retrieve the extended transaction identifier. The framework can use the extended transaction identifier to retrieve the associated extended transaction descriptor, then locate the address of the function for the event handler and, finally, call that function by address. For the most part, the framework just passes function calls from the application to the correct handler (function) in an RTF Library, but it can also evaluate any guards placed on the event and perform basic error checking. Thus, the application program calls extended transaction control operations by name in the extended transaction interface, rather than by pointer in an RTF Library.

Relieving application developers from the burden of invoking the appropriate handler when a transaction significant event is raised is only one role that adapters play in the framework. Another is to bind the set of events an extended transaction can invoke, hence defining the interface to extended services available to an application. The Reflective Transaction Framework does not *a priori* assume a specific transaction model for an extended transaction. Instead, it provides the means for an application programmer to *select* a model for each extended transaction using the `select` command from the extended transaction interface.

Operation Definition 3.2 (select) - `select(transaction_name, model_name)` *binds a set of transaction significant events associated with the specified transaction model to the extended transaction descriptor for* transaction_name.

The select command *fixes the interface* of an extended transaction. The framework, however, does not assume a fixed set of events for a given transaction model, nor does it associate a handler with each event. Instead, the framework provides the means to specify the transaction significant events for a named transaction model, and bind a handler and guards to each event using the following commands from the *metalevel interface.*

Operation Definition 3.3 (register_event) - `register_event(etrid, event_name)` *creates an event descriptor structure (see Figure 3.3) for the named event* event_name *and initializes all fields in the structure. The event descriptor is stored in the* TRANEVENTS *entry of the extended transaction descriptor for the transaction identified by* etrid.

To bind a transaction significant event to a handler function in an RTF Library, the transaction system programmer calls `bind_handler`, providing both the name of the transaction event and the name of the function that will act as the handler. Normally, binding occurs only when an extended transaction is initialized, but it can also be used when runtime conditions are altered and an alternate handler is required to process the event.

Operation Definition 3.4 (bind_handler) - `bind_handler (etrid, event_name, handler_function_name)` *sets the handler for an event to the named handler function by storing a pointer to handler_function_name in the* HANDLER *field of the event descriptor. When the application raises the event an indirect function call will be made by referencing a pointer to the* HANDLER *field.*

Each event can be associated with one or more guards that identify a predicate to evaluate prior to invoking the handler. Guards are implemented as functions in an RTF Library. If all predicates (guards) evaluate to TRUE when the event is raised, the handler is invoked. Otherwise, the event may be delayed or rejected, depending on scheduling properties of the event.

Operation Definition 3.5 (assign_guard) - `assign_guard(etrid`$_{t1}$`, event_name, guard_name)` *appends the string guard_name to the* GUARDS *field of the event descriptor for event_name. If the keyword* NULL *is supplied as input for guard_name, all values recorded in the* GUARDS *field are removed and the field is set to null.*

Each event is associated with properties that specify what actions the framework can take in scheduling the execution of the associated handler. The possible actions include variously *allowing*, *delaying* or *rejecting* the execution of the handler, or possibly *triggering* another event to satisfy a dependency or runtime correctness constraint. We describe the use of event properties later in Section 3.3.3, but note here that event properties are recorded in two fields: the ATTRIBUTE field and the TRIGGERABLE field.

When an event descriptor is created, the ATTRIBUTE field is initialized to "normal", indicating that, if necessary, the execution of the handler can be delayed or rejected. The ATTRIBUTE field can be redefined using the command `event_property`.

Operation Definition 3.6 (event_property) - `event_property(etrid, event_name, event_type)` *sets the attribute field of the event descriptor for event_name to the value event_type, where event_type is either* ``normal'', ``inevitable'' *or* ``immediate''.

The *triggerable* field of the event descriptor indicates whether the framework can initiate the event, a property orthogonal to the event properties *"normal"*, *"inevitable"* or *"immediate"*. When an event descriptor is created, the *triggerable* field is initialized to FALSE, indicating that event *event_name* cannot be triggered. The field can be reset using `can_trigger`, providing the name of the event and boolean value as input.

Operation Definition 3.7 (can_trigger) `-can_trigger(etrid, event_name, bool)` *sets the* triggerable *field of the event descriptor for* event_name *to the boolean value.*

To summarize, when a transactional application needs to run an extended transaction, it first creates an extended transaction descriptor, then selects a specific extended transaction model. In response, the framework creates and initializes an extended transaction descriptor, along with an event descriptor for each transaction significant event the model supports, storing the address of the event handler function in the event descriptor. To invoke a control operation in an RTF Library, an application calls that function in the extended transaction interface and passes the identifier of the extended transaction. The framework retrieves the extended transaction descriptor, then calls the handler function using the address stored in the event descriptor.

We close our discussion on the architecture of the Reflective Transaction Framework with a summary of commands in the *metalevel interface*, presented in Table 3.4. These commands are made possible via various openings presented by the open implementation of the TP monitor — *introspection*, which enables an application to reify selected state for a transaction in an extended transaction descriptor; *explicit invocation*, which enables an application to directly invoke existing functions provided by functional components of the TP monitor; and, *intercession*, which builds on introspective and explicit invocation capabilities to define new extended transaction control operations and link these operations with a specific extended transaction. Some of the commands listed in Table 3.2 have already been introduced; the balance of the metalevel interface will be presented in Section 3.3 as we describe the extended transaction services provided by the framework.

Table 3.4: Summary of the commands in the metalevel interface.

Interface Exported By	*Responsibility*	*Transaction Adapter Command*
TRANSACTION MANAGEMENT ADAPTER	Initialization	`instantiate`
		`register_event`
		`bind_handler`
		`assign_guard`
		`event_property`
		`event_trigger`
	Extended Transaction Descriptor	`getetrid_using_name`
		`getetrid_using_trid`
		`getname_using_etrid`
		`getname_using_trid`
		`gettrid_using_name`
		`gettrid_using_etrid`
		`transtate`
		`set_type`
		`get_type`
		`set_state`
		`get_state`
		`set_etranprop`
		`get_etranprop`
		`record_event`
		`find_event`
		`setp`
		`getp`
	Transaction Management	`begin_tran`
		`commit_tran`
		`abort_tran`
		`thread_to_trid`
	Execution	`define_dependency`

continued on next page

continued from previous page		
Interface Exported By	*Responsibility*	*Transaction Adapter Command*

	Control	`form_dependency`
		`delete_dependency`
		`enable_dependency`
		`disable_dependency`
		`list_dependency`

LOCK ADAPTER	Lock Management	`lock`
		`unlock`
		`unlock_all`
		`locks_held`
		`locks_waitfor`
		`lock_list`

	Transaction Restructuring	`create`
		`delete`
		`insert`
		`remove`
		`delegate`
		`acquire`

CONFLICT ADAPTER	Semantic Conflict	`load_table`
		`remove_table`
		`ignore_conflict`
		`remove_icrecord`
		`clear_icset`
		`select_table`
		`clear_policy`

3.3 Extended Transaction Services

This section presents the detailed design of the extended transaction services provided by the Reflective Transaction Framework. Specifically, we present the design of *dynamic transaction restructuring*, *semantic transaction synchronization* and *transaction execution control*. These extensions were selected because they provide a base for expressing a wide range of extended transaction behaviors and, consequently, provide the greatest leverage to implement advanced transaction models and semantics-based concurrency control protocols. In this section we consider each extended transaction service in turn, first presenting an overview of the extension as supported by our design, then considering the implications of adding this extension and identifying assurances that must be made to guarantee transaction correctness and, finally, listing commands provided to utilize this new extended transaction service.

It should be emphasized that we do not intend that an application programmer use these extended transaction services directly. Rather, we expect these services and associated commands to be used by systems programmers to implement higher-level primitives for extended transactions. In terms of our *separation of programming interfaces*, described in Section 3.2.2, commands for these extended transaction services make up the *metalevel interface* that transaction system programmers will use to implement extended transaction control operations in the *extended transaction interface*.

3.3.1 Dynamic Transaction Restructuring

An essential requirement of many advanced transaction models is the ability for member transactions to *dynamically restructure*. From a transaction execution point of view, dynamic restructuring is the ability of an extended transaction to transfer ownership of data objects to another extended transaction explicitly. Dynamic restructuring allows an extended transaction to selectively make tentative and partial results, as well as give hints, such as coordination information, accessible to other extended transactions. Dynamic restructuring also makes it possible to decouple the fate of updates to data objects from that of the extended transaction that performed the operation(s); for instance, an extended transaction can transfer selected data objects that will remain uncommitted but alive after it aborts. Examples of advanced transaction models that can be synthesized using transaction restructuring by resource delegation include Reporting Transactions [CR91a], Chained Transactions [Chr91], SAGAs [GMS87, CR92], Nested Transactions [Mos85], and both Split and Join Transaction models [PKH88, KP92].

In our design, dynamic transaction restructuring is realized through the *delegation* of locks held on data objects from one extended transaction to another. After the delegation of a lock is complete, the scope and fate of the data object that it protects, i.e., its visibility and conflicts with the operations of other transactions, are dictated by the scope and fate of the delegatee transaction.

Definition 3.1 (Delegation) *The operation Delegate(t_1, t_2, obName) transfers ownership of the lock extended transaction t_1 holds on obName to extended transaction t_2. More generally, Delegate(t_1, t_2, DelegateSet) delegates the lock held by t_1 on each data object in DelegateSet to t_2.*

To perform dynamic restructuring operations, an extended transaction must have the appropriate permissions set. Specifically, to delegate a lock, the property *Delegate_Enabled* must be set to TRUE for the delegator; and similarly, to acquire a lock the property *Acquire_Enabled* must be set to TRUE for the delegatee. These properties, recorded in the descriptor for an extended transaction, are set using the command **setp**. Thus, in preparation for extended transaction t_1 to delegate locks on data objects to extended transaction t_2, the application must first set permissions **setp(t_1, delegate_enabled,** TRUE) and **setp(t_1, acquire_enabled,** TRUE), respectively.

In what we have discussed so far, a transaction delegates the lock for a single data object with each invocation of delegate. Delegation of a *set* of locks in a single invocation can be regarded as the atomic invocation of multiple delegations, one for each lock in the set. We speak of *global delegation* when a transaction transfers the responsibility for all its locks at once, and *partial delegation* when a transaction transfers the responsibility for only a subset of its locks. Global delegation is best suited for transaction models where the set of data objects that will be delegated at the termination of the transaction is known in advance. The Nested Transaction model [Mos85] is a well-known example of global delegation: upon commit, a sub-transaction does a global delegation of all locks that it holds on data objects to its parent transaction. Other advanced transaction models that use global delegation include the Chained Transaction model [Chr91], the Join transaction model [KP92] and SAGAs [GMS87, CR92]. Partial delegation is best suited for transaction models that make partial results, such as hints and coordination information, accessible to other extended transactions, and for transaction models that support open-ended activities where processing is unpredictable and the set of data objects that must be transferred is only known at the time restructuring actually occurs. The Split transaction model [PKH88] is a straightforward example of the use of partial delegation: an application can select a set of objects that an extended transaction holds and delegate

locks on these objects to another extended transaction. Other advanced transaction models that use partial delegation include the Co-Transaction model [CR91b] and Reporting Transactions [CR91a].

To perform a `delegate` operation, an extended transaction must provide the name of a structure that lists the data objects to be delegated. This structure is referred to as the *delegate set.*

Definition 3.2 (Delegate Set) *A delegate set is a named container of logical lock names, where each name is associated with a data object that an extended transaction wishes to delegate. To create a delegate set the transaction must provide a unique name for the delegate set and identify which transaction (delegator or delegatee) is responsible for the delegate set (the purpose of declaring responsibility will be described later in this section). After creating a delegate set, an extended transaction can then insert and remove data objects for which it holds a lock.*

The LOCK ADAPTER provides a command to `create` a named (empty) delegate set, along with commands to `insert` and `remove` the names of data objects that it wishes to delegate. Thus, to perform partial delegation, an extended transaction first creates a named delegate set, then inserts the names of selected data objects. Similarly, to perform global delegation, an extended transaction first creates a named delegate set and then issues the insert command, using the keyword ALL to insert the names of all data objects that it currently has locked at that point in time.

Operation Definition 3.8 (Delegate) `delegate(`t_1`, `t_2`, DelegateSet, dtype)` *directs the* LOCK ADAPTER *to transfer ownership of the lock on each data object specified in the named DelegateSet from extended transaction* t_1 *to extended transaction* t_2*. The parameter* `dtype` *specifies when the transfer of locks is to take place – the keyword* IMMEDIATE *indicates that the transfer is to take place at once, while the keyword* DEFERRED *specifies that the transfer will be deferred until the delegatee requests the locks.*

Requirements for performing delegation are that the transaction have permission set to delegate and that it hold a lock on each data object it is attempting to delegate. Requirements for receiving the delegated locks are that the transaction have permission set to acquire delegated data objects. In addition, both delegator and delegatee must currently be *active* (i.e., *initiated* but not *terminated*). Thus, we have the following guard for well-formed delegation.

Guard 3.1 (Well-Formed Delegation) *For the delegate(t_1, t_2, DelegateSet) operation:*
Preconditions

- *State(t_1, Active) = True* AND
- *State(t_2, Active) = True* AND
- *Delegate_Enabled(t_1) = True* AND
- *Acquire_Enabled(t_2) = True* AND
- *For each obname in the DelegateSet, Holds_Lock(t_1,obname) = True*

Postconditions

- *For each obname in the DelegateSet, Holds_Lock(t_1,obname) = False* AND
- *For each obname in the DelegateSet, Holds_Lock(t_2,obname) = True*

The transfer of locks from the delegator to the delegatee occurs immediately after the delegator issues the **delegate** command. An alternative is to defer the transfer of the locks on the delegated data objects until the delegatee indicates it is ready to *acquire* the locks. This is referred to as *deferred delegation.* To perform a deferred delegation the delegator must specify which transaction, the delegator or delegatee, is *responsible* for the delegate set. This value is set when the delegate set is created. Intuitively, the *responsible transaction* is obligated to *eventually* acquire the locks on the data objects in the delegate set. A brief example is presented to clarify this.

```
create(t1, mydelset, DTEE)                          (1)
insert(t1, mydelset, account003)                    (2)
insert(t1, mydelset, account007)                    (3)
delegate(t1, t2, mydelset, DEFERRED)                (4)
```

In Line 1 extended transaction t_1 creates a named delegate set and identifies the delegatee as the *responsible transaction.* In Line 2 and Line 3, t_1 inserts named data objects into the delegate set. Finally, in Line 4, t_1 delegates the locks on the data objects in the named delegate set to t_2, specifying that the actual transfer is to be *deferred* until t_2 is prepared to acquire the delegate set. After the delegate operation has successfully completed, t_1 will no longer hold locks on the data objects specified in delegateSet. However, since the actual transfer was deferred, t_2 does not yet own the locks. Until t_2 requests the locks, they will be held by a intermediary transaction managed by the TRANSACTION MANAGEMENT ADAPTER.

To realize the deferred delegation of data objects we introduce the operation **acquire**. This operation indicates that the intended recipient of a deferred delegation (e.g., the delegatee) is prepared to receive the locks on the delegated data objects and directs the LOCK ADAPTER to complete the transfer.

Operation Definition 3.9 (Acquire) *The operation* `acquire(t2, delSet)` *indicates that extended transaction t_2 is prepared to acquire and directs the* LOCK ADAPTER *to perform the transfer.*

For a transaction to acquire a delegate set, it must be permitted to acquire delegated data objects and be the intended recipient of the named delegate set. Thus, we have the following guard for well-formed acquire.

Guard 3.2 (Well-Formed Acquire) *For the acquire operation:*
Preconditions:

- *State(t_2,Active) = True* AND
- *Acquire_Enabled(t_2) = True* AND
- *Delegatee(DelegateSet) = t_2*

Postconditions:

- *For each obname in the DelegateSet, Holds_Lock(t_2,obname) = True*

Adding Dynamic Transaction Restructuring

We now discuss the issues that arise from adding the capability for extended transactions to restructure dynamically through delegation, and discuss how these issues are handled. Specifically, we first identify properties that the Reflective Transaction Framework must preserve during transaction restructuring for key transaction correctness requirements to be satisfied. Next, we present the application programming interface commands the LOCK ADAPTER provides to support dynamic transaction restructuring. Finally, we identify transaction services required from the lock management services the TP monitor provides, and any underlying assumptions in our design.

Bypassing the Lock Scheduler The delegation of data objects involves explicitly passing ownership of the lock on delegated data objects from one transaction to another. However, the lock service of the underlying TP monitor is responsible for servicing lock requests, typically in a *first come first served* manner, queuing lock requests that cannot be immediately granted following a FIFO queuing policy. If the delegator were to release its lock on a data object that it wished to delegate, the lock service of the underlying TP monitor would grant the lock to the first transaction in the lock queue – not necessarily the delegatee transaction. Thus, to realize lock delegation, the LOCK ADAPTER must effectively bypass the lock request scheduler of the underlying TP monitor.

To accomplish this, the LOCK ADAPTER utilizes the services of the CONFLICT ADAPTER, to lock and unlock data objects explicitly, and to relax conflicts between incompatible

lock requests. For each data object being delegated, the LOCK ADAPTER first notifies the CONFLICT ADAPTER that a single instance of a lock conflict between the delegator and delegatee transaction on this data object should be *relaxed*. The LOCK ADAPTER then issues a `lock` command to obtain a lock on the data object on behalf of the delegatee. The lock service of the TP monitor will detect a lock request conflict, due to the fact that a lock on the data object is already held by the delegator, raising a conflict event to the CONFLICT ADAPTER. The CONFLICT ADAPTER relaxes the lock conflict, allowing the delegatee transaction to obtain the lock on the data object (see the semantic conflict discussion in Section 3.3.2 for more details). At this point, both delegator and delegatee hold a lock on the data object. Finally, the lock adapter issues an `unlock` command to release the lock on the data object on behalf of the delegator.

Preventing Transaction Deadlock One consequence of the fact that delegation bypasses the lock request scheduler is the potential for transaction deadlock; namely, the potential for deadlock between the delegatee and another transaction waiting for a lock on one of the data objects being delegated. There are two approaches to dealing with deadlocks: *detection* and *avoidance*. The first approach, detection, assumes deadlocks are rare and allows delegation to proceed *unchecked*, then relies on the TP monitor to detect deadlocks. The second approach, avoidance, explicitly checks whether the call to delegate a lock would result in a deadlock, returning a status code to disallow the delegation.

The latter mechanism was chosen to prevent transaction deadlocks from occurring after delegation for two reasons. First, while deadlocks might be rare in correctly written application code, the added flexibility of transaction restructuring can introduce programming errors, increasing the chance for deadlocks to occur. Second, we do not want the underlying transaction system to resolve deadlocks, as it would likely terminate the waiting transaction, which in all likelihood would be a conventional ACID transaction. Instead, the computational cost (e.g., CPU cycles) and risk of blocking or possible termination should be the responsibility of the extended transaction attempting to perform the delegation. The current LOCK ADAPTER design uses a simple procedure for detecting deadlocks during delegation. For the delegatee transaction, the implementation simply examines the list of locks that it is waiting for. If lock waits are rare, which is common in most application environments, the procedure can immediately conclude that no deadlocks exist. Otherwise, for each transaction holding the lock, the list of locks that transaction is waiting for is examined and so on, until a cycle is detected or all locks are examined.

Guard 3.3 (Deadlock Prevention) *If the delegation of any data object would result in deadlock the* `delegate` *operation will not proceed and an error will be reported.*

Preventing Orphaned Data Objects When performing a deferred delegation it is necessary to protect against orphaned data objects. This can occur if both delegator and delegatee transactions were to terminate before the delegatee executes the `acquire` command, leaving the locks on the data objects *"unclaimed"*. To prevent this, the LOCK ADAPTER requires the transaction performing a deferred delegation to indicate which transaction is responsible for the delegate set, the delegator (itself) or the intended delegatee. The transaction responsible for the delegate set will not be allowed to commit until it has acquired the delegate set; if the responsible transaction is preparing to abort, the locks on the data objects in the delegate set must first be acquired. Thus, the fate of the data objects in the delegate set lie with the responsible transaction, in the sense of visibility and committing or aborting the updates that have been made to the data objects.

To accomplish this, the LOCK ADAPTER notifies the TRANSACTION MANAGER ADAPTER to record a termination dependency between the responsible transaction and the named delegate set. If the dependency cannot be created, the deferred delegation is not allowed to succeed. This dependency is removed only when the intended delegatee issues the `acquire` operation, or the responsible transaction terminates. Therefore, the termination of the responsible transaction dictates the fate of the locks on the data objects in the delegate set, eliminating the possibility of orphaned data objects due to delegation.

Guard 3.4 (Preventing Orphaned Resources) *The transaction responsible for a deferred delegation is not allowed to commit or abort until it has acquired the delegate set.*

Preserving Transaction Dependencies Delegating data objects not only means transferring ownership of the delegated locks, but also transferring the transaction dependencies that were created by acquiring these locks. To illustrate, if transaction t_1 delegates an exclusive write lock on a data object *ob* to transaction t_2, t_1 is no longer able to access *ob* after the delegation until t_2 either releases its lock on *ob* or delegates the lock back to t_1. Moreover, if t_1 acquired the lock on *ob* by ignoring a conflict with transaction t_3, forming a dependency between t_1 and t_3, then t_1's dependency on transaction t_3 is also transferred, such that after delegation, t_2 now depends on t_3.

A prerequisite for the successful delegation of data objects, then, is the successful delegation of the dependencies associated with these data objects. This implies that the transfer of the dependency does not introduce a cycle in the dependency graph of the delegatee transaction. This gives rise to the following guard:

Guard 3.5 (Preserving Transaction Dependencies) *If the transfer of a dependency associated with a data object being delegated would introduce a cycle in the dependency graph of the delegatee transaction then the* `delegate` *operation will fail.*

To support the specification and implementation of dynamic transaction restructuring, the LOCK ADAPTER provides the following operations that permit an extended transaction to create and manipulate named delegate sets during transaction execution. Where appropriate, selected status codes for each operation are provided.

- `create(`t_1`:etrid, delegateSet:string, resp:string)`: creates a named (empty) container for transferring access to and responsibility for data object resources from one transaction to another, referred to as the *delegate set.* The owner of the named delegate set is set to the identifier (etrid) of extended transaction t_1. The parameter resp identifies the transaction responsible for the delegate set (`dtor` = delegator and `dtee` = delegatee). Return codes for this command include:
  - `success`
  - `delegate set name not unique`
  - `responsible transaction not specified`
- `delete(`t_1`:etrid, delegateSet:string)`: deletes the named delegate set. Transaction t_1 must be the owner of the delegate set. The set does not have to be empty (it can contain lock names), but it must not have already been delegated in a deferred delegation. Return codes for this command include:
  - `success`
  - `delegate set not found`
  - `not owner of delegate set`
  - `deferred delegation in progress`
- `insert(`t_1`:etrid, delegateSet:string, dataObject:string)`: inserts the name of the data object into the specified delegate set. If the keyword ALL is specified, in place of a data object name, the name of all locks that t_1 currently holds will be inserted into the specified delegate set. Return codes for this command include:
  - `success`
  - `delegate set not found`
  - `not owner of delegate set`
- `remove(`t_1`:etrid, delegateSet:string,dataObject:string)`: removes the name of the data object from the specified delegate set. If the keyword ALL is specified in place of data object name, the names of all locks currently in the specified delegate set will be removed. Return codes for this command include:
  - `success`
  - `delegate set not found`
  - `not owner of delegate set`
  - `deferred delegation in progress`
  - `data object not found`

- `delegate(dtor:etrid, dtee:etrid, delegateSet:string, dtype:string)`: transfers locks on the data object specified in the named delegate set from extended transaction t_1. If the parameter dtype is set to `immediate`, the transfer of locks is attempted immediately. However, if the parameter dtype is set to `deferred`, the locks are transferred to an intermediary transaction managed by the TRANSACTION MANAGEMENT ADAPTER. Return codes for this command include:
  - `success`
  - `delegation not enabled for delegator`
  - `delegatee is not active`
  - `acquire not enabled for delegatee`
  - `transaction deadlock detected`
  - `transaction dependency cycle detected`
- `acquire(t2:etrid, delegateSet:string)`: transfers locks on data objects specified in the named delegate set to transaction t_2. Extended transaction t_2 must be the intended recipient of the delegate set and must have the property acquire_enabled set to true. Return codes for this command include:
  - `success`
  - `delegate set not found`
  - `not specified delegatee`
  - `acquire not enabled`
  - `dependency cycle detected`
  - `transaction deadlock detected`

Finally, we identify the support that the LOCK ADAPTER requires from the transaction services of the underlying TP monitor, primarily from the Lock Manager, to implement dynamic transaction restructuring.

- To interface with the Lock Manager of the underlying TP monitor, the LOCK ADAPTER requires access to the data type for lock names (`lock_name_t`). This type will be used in constructing the delegate set, and for explicitly locking and unlocking data objects during lock delegation.
- To implement the `delegate` and `acquire` operations, the LOCK ADAPTER requires that the Lock Manager export lock service interface functions to `lock` and `unlock` individual data objects explicitly on behalf of an extended transaction.
- To perform global delegation, the LOCK ADAPTER requires that the Lock Manager export a function that returns a list of all locks held by a transaction, referred to as the *transaction lock list.*
- To perform efficient deadlock detection, the LOCK ADAPTER also requires an access function that returns a list of the transactions waiting for a lock on a specific data object, referred to as the *lock request list.*

- Finally, the LOCK ADAPTER requires that the Lock Manager allow multiple transactions to possess a lock on a data object in the same mode simultaneously. A typical lock manager already allows multiple transactions to hold Read (Shared) locks on a data object, so that multiple possession is common. However, the LOCK ADAPTER requires the ability for multiple transactions to hold Write (Exclusive) locks as well. Specifically, the data structure used in the lock table to count the number of times a lock is held in a particular mode, referred to as a possession vector, must permit multiple transactions to hold the lock in exclusive mode.

3.3.2 Semantic Transaction Synchronization

The purpose of transaction synchronization, or concurrency control, is to mediate access to data objects so that the consistency of the data is not compromised when accessed by concurrently executing transactions. Fundamental to all transaction synchronization is the notion of *conflict* — incompatibility between operations or transactions. Traditional concurrency control used in most commercial database systems and transaction processing systems defines conflict in terms of `read` and `write` operations [BHG87] (abbreviated as R/W) — two operations conflict if one is a `write` operation.

Definition 3.3 (R/W **Conflict**) *An operation P in transaction t_1 is in conflict with another operation Q in transaction t_2, if both operations access the same data object O and at least one of them is a* `write` *operation. Operations P and Q are said to be conflicting operations and, similarly, transactions t_1 and t_2 are said to be conflicting transactions.*

The Lock Manager component of a TP monitor detects R/W conflicts when a transaction requests a lock in order to perform an operation on a data object. In our design, the lock acquisition mechanism of the Lock Manager must also raise a *conflict event* when a R/W conflict is detected. To perform semantic transaction synchronization, an application must first set the property *sbcc_enabled* to TRUE using the command `setp`$_{t1}$`(sbcc_enabled,` TRUE). This *registers* the CONFLICT ADAPTER to receive a *conflict event* when the Lock Manager detects a R/W conflict involving extended transaction t_1. It is this conflict event that enables the CONFLICT ADAPTER to participate in resolving R/W conflicts.

Definition 3.4 (Conflict Event) *The Lock Manager raises a conflict event when a* R/W *conflict is detected for an extended transaction lock request. Each conflict event returns the following information:* hold$_{TRID}$ *- identifier of the transaction holding the lock,* hold$_{op}$ *- operation currently active,* hold$_{mode}$ *- mode the lock is being held,* lockName *- logical lock name,*

request$_{TRID}$ - *identifier of the transaction requesting the lock,* request$_{op}$ - *operation pending, and* request$_{mode}$ - *mode the lock is being requested.*

The basis for transaction synchronization in semantics-based concurrency control and many advanced transaction models is the introduction of their own notion of conflict that uses available semantic information to relax R/W conflicts, referred to as *semantic compatibility.* Two operations are semantic compatible if their relative order of execution is insignificant from the point of view of the application. Semantic compatibility is typically weaker than traditional R/W compatibility and permits a higher degree of transaction concurrency [GM83, FO89, BR91, RC92].

There are several sources of information available to an application to define semantic compatibility. One source is *operation level semantics*, where data access semantics beyond `read` and `write` are considered. For example, in the case of credit and debit on a bank account data object — the commutativity property of `Deposit` and `Withdraw` operations allows the transaction system to achieve higher concurrency by allowing these operations to run concurrently, where `read` and `write` operations could not. Another source is *transaction level semantics*, where information on structured interactions between transactions can be used to specify semantic compatibility. For example, cooperative serializability [MP92, RC92] uses semantic information to permit conflicting operations to run concurrently, as long as the transactions that issued the conflicting operations are in the same cooperative transaction group. This supports collaborative and cooperative work, where the exchange of intermediate information is desirable and necessary. A bank customer waiting for an account balance or activity summary at an automatic teller machine would not be delayed if the request were issued as a cooperative transaction to other transactions posting interest or auditing accounts. Yet another source of information is *application level semantics*, where information on the application that issued the transaction can be used to define semantic compatibility. For example, if the application can tolerate a limited amount of inconsistency in a query result, this information can be used to allow conflicting operations to execute concurrently as long as the total inconsistency is below the specified limit. A bank officer requiring branch balance information accurate to within ± $10,000 could issue such a transaction during times of peak customer activity.

Generally speaking, the more semantic information available for transaction synchronization, the greater the degree of concurrency that can be achieved. However, representing and using these various forms of semantic information can be problematic. Because an overriding goal of our design is practical usability, we must strike a balance between efficiency and the ability to represent semantic information.

The CONFLICT ADAPTER provides a *semantic transaction synchronization service* that allows an application to define and select semantic compatibility for individual extended

transactions. The only restriction it imposes is that semantic compatibility be expressible in terms of either a *compatibility relation* or an *explicit cooperation relation* between extended transactions. These representations have the advantage of being simple to create. They can be efficiently tested at runtime and, as we shall demonstrate, they facilitate the implementation of a wide range of semantic transaction synchronization methods.

Compatibility Relation

A number of semantic transaction synchronization methods in the literature can be expressed as a *compatibility relation* between pairs of semantically rich operations. The compatibility relation specifies whether two conflicting operations can be allowed to execute concurrently, or indicates actions that may be taken to guide the resolution of the conflict.

Typically, a compatibility relation reflects the general (i.e., state-independent) commutativity of operations and considers only operation name or transaction type [Kor83, SS84]. In addition, state-dependent commutativity can be exploited, for example, by considering the return values of the operations [O'N86, Wei88], and this can be further refined by considering one-sided commutativity [BR91]. In all of these cases, the allowable interleaving of transactions can be expressed by a compatibility relation as commutative pairs of operations that can be freely reordered. However, depending on the semantics of an operation and its relationship to other active operations, this reordering may produce transaction dependencies or serialization orderings. In the recoverability protocol [BR91], for example, if a conflicting lock request is granted because the operation is *recoverable* with respect to all uncommitted operations, a commit-dependency $t_i \rightarrow t_j$ must be created for each transaction t_j that owns a lock in a mode incompatible but *recoverable* to t_i.

One may even go one step further by declaring two non-commutative operations *semantically compatible* if the different effects of the two possible execution orders are considered negligible from an application point of view (e.g., a pair of `deposit` and `withdraw` operations on bank account without overdraft protection, but with a penalty of, say, \$10.00 charged at the end of the business day if a `withdraw` operation results in a negative balance). A compatibility relation may also be derived from a specification of the precondition of the operations [AAS93]. Further, one can define semantic compatibility relations that enforce an upper limit on the number of semantically compatible but non-commutative operations that are out of order with respect to the serialization order of the transactions. This latter type of "bounded inconsistency" guarantee was introduced in epsilon-serializability [RP95], and used in the proclamation [JS92] method as well.

Representing semantic information as a compatibility relation between pairs of semantically rich operations can support a wide range of transaction synchronization methods

from the literature. Moreover, it lends itself to a practical implementation, as it is essentially context-free — the method considers only pairs of operations or pairs of transactions, rather than operation sequences, combinations of interleaving transactions, future database states, etc. As such, these various compatibility relations can be represented using simple *semantic compatibility tables* to guide the resolution of conflicts.

Definition 3.5 (Semantic Compatibility Table) *The semantic compatibility of operations performed on a data object is defined by a two dimensional compatibility table: one dimension corresponds to the operation type currently active and holding a lock on the data object, the other corresponds to the operation requesting a lock. Each entry in a compatibility table is of the form* `[Action, Dependency]`, *where* `Action` *is one of:* SOK – *the operations are semantically compatible and the conflict can be relaxed,* NOK – *the operations conflict, or* event – *a named event (predicate) that is evaluated to determine semantic compatibility, and where* `Dependency` *is a named transaction dependency that is to be recorded between the two corresponding transactions if the conflict is relaxed.*

A semantic compatibility table specifies the name of the data object to which it applies, using the keyword ALL if the table can be used for any data object. Each table also specifies how its entries are indexed. In our current design, the options for indexing entries in the table are OPNAME — entries are accessed using the name of the active operation and the name of the operation requesting a lock, and LOCKMODE — entries are accessed using the mode in which the lock is currently being held and the mode in which the lock is being requested. Both OPNAME and LOCKMODE are supplied in the conflict event. This design can be easily extended to include other values for indexing semantic compatibility tables; for example, the transaction identifiers (TRIDs) provided by the conflict event could be used to look up information in the corresponding extended transaction identifiers, such as such as transaction type or transaction name.

To illustrate, consider the semantic compatibility table for an *Account* data object based on operation commutativity [Wei88]. Semantic compatibility between the operations `Deposit`, `Withdraw`, and `Balance` is reflected in the entries of the table, in which columns represent operations currently holding a lock, and rows represent operations requesting a lock. Entries marked SOK indicate that the requested operation is semantically compatible (commutes) with the concurrently executing operation, while an entry marked NOK indicates the requested operation conflicts. Reordering commutative operations does not produce a transaction dependency, so no dependency (ND) is recorded.

Example 3.1 (Compatibility relation based on commutativity)

ACCOUNT:OPNAME	Balance	Deposit	Withdraw
Balance	SOK;ND	NOK;ND	NOK;ND
Deposit	NOK;ND	SOK;ND	NOK;ND
Withdraw	NOK;ND	SOK;ND	NOK;ND

To perform an operation on the Account data object, an extended transaction would first request a lock. If the Lock Manager detects a R/W conflict, it raises a *conflict event* that the CONFLICT ADAPTER attempts to resolve using the semantic compatibility table. In this case, a simple table lookup indexed by the name of the operation currently active and the name of the operation requesting the lock determine if the conflict can be relaxed.

As another example, consider epsilon serializability (ESR). In ESR, a precondition for allowing semantically compatible but non-commuting operations is that a predicate, ESR, must first be evaluated to determine if a "bounded inconsistency" guarantee still holds.

Example 3.2 (Compatibility relation based on epsilon serializability)

ALL:LOCKMODE	Share(s)	Exclusive(x)
Share(s)	SOK;ND	ESR;ND
Exclusive(x)	ESR;ND	NOK;ND

As in our previous example, when the Lock Manager detects a R/W conflict, it raises a *conflict event* and passes the event to the CONFLICT ADAPTER. The CONFLICT ADAPTER performs a table lookup using the mode the lock is held and mode the lock is being requested. If the lookup returns ESR, the CONFLICT ADAPTER evaluates the predicate ESR using the conflict event data — if the predicate returns TRUE, the conflict is relaxed. Later, in Chapter 4 we shall revisit these examples and present their implementation.

To use a compatibility relation for semantic transaction synchronization, an application must first load the semantic compatibility table using the command `load_table`. This command directs the CONFLICT ADAPTER to load the table from the specified pathname, and assigns a name and class type to the compatibility table.

Operation Definition 3.10 (load_table) `-load_table(`t_1`, pathname, name, class)` *directs the* CONFLICT ADAPTER *to load a compatibility table from the pathname, and assign this table the specified name and class name.*

Using `load_table`, an application can load multiple semantic compatibility tables. These tables are stored in a compatibility table set, referred to as *CompTblSet*. An application can specify which semantic compatibility table(s) should be used for semantic transaction synchronization for an extended transaction using the `select_table(`t_1`,`

`class)` command. This command appends the class name to the *sbcc_policy* field of the extended transaction descriptor.

Operation Definition 3.11 (select_table) *The command* `select_table(t1, class)` *appends the class name to the* sbcc_policy *field of* t_1*'s extended transaction descriptor.*

At runtime, in an attempt to relax a R/W conflict for an extended transaction, the CONFLICT ADAPTER will use the semantic compatibility table(s) found in the *CompTblSet* that match the class names listed in the *sbcc_policy* field.

Explicit Cooperation

The CONFLICT ADAPTER provides support for explicit transaction cooperation by enabling an application to establish *ignore-conflict* relationships between extended transactions. An ignore-conflict relationship specifies conditions under which a transaction will allow other transactions access to data objects on which it currently holds a lock. The ignore-conflict relationship is an essential component in constructing extended transactions that require explicit cooperation, such as cooperative transaction groups [NZ90, MP92, RKT+95], nested transactions [Mos85], as well as semantics-based concurrency control based on structured cooperation [GM83, BK91, FO89, SGMS94].

Definition 3.6 (ignore-conflict relationship) *An ignore-conflict relationship specifies that an extended transaction wishes to ignore conflicting lock requests from a specific extended transaction. For example, if an application declares an ignore-conflict relationship on extended transaction* t_1 *with transaction* t_2*, then subsequent lock requests from* t_2 *that conflict with locks held by* t_1 *will be permitted. The ignore-conflict relationship is not commutative.*

The ignore-conflict relationships established for an extended transaction t_1 are recorded in a table, referred to as the cooperating transaction set for t_1.

Definition 3.7 (Cooperating Transaction Set) *The cooperating transaction set for an extended transaction* t_1*, denoted* $CoopTr\_Set_{t1}$*, specifies all ignore-conflict relationships established by* t_1 *at that point in time. Each element of* $CoopTr\_Set_{t1}$ *is a unique ignore-conflict relationship. Thus, if no explicit cooperation occurs between* t_1 *and any other extended transaction, the cooperating transaction set* $CoopTr\_Set_{t1}$ *is empty.*

An application can qualify ignore-conflict relationships for an extended transaction by specifying data objects and operations, thereby restricting the set of conflicts that are to be relaxed. In addition, the application can specify a named event (predicate) that must be

evaluated to determine if the conflict can be relaxed, and a transaction dependency that is to be recorded if the conflict is relaxed. Thus, each ignore-conflict record in $CoopTr\_Set_{t1}$ for extended transaction t_1 is of the form: `[CoopTran, ObjName (optional), OperName (optional), Event (optional), DepName (optional), Handle (optional)]`, where *CoopTran* is the name of the extended transaction that t_1 will allow conflicting lock requests, *ObjName* is an optional parameter specifying the data object that t_1 will allow CoopTran to access, *OperName* is an optional parameter specifying the operation t_1 will allow CoopTran to perform on ObjName, *Event* is an optional parameter specifying a predicate that is to be evaluated to determine semantic compatibility, *DepName* is an optional parameter specifying a named transaction dependency to be recorded if the conflict is relaxed, and *Handle* is an optional parameter specifying a unique name for the ignore-conflict record.

Operation Definition 3.12 (ignore_conflict) *When extended transaction t_1 issues the operation* `ignore_conflict(`t_1`, `t_2`, objName, operName, event, depName, handle)`, *an ignore-conflict record is created and placed in $CoopTr\_Set_{t1}$. This operation establishes an ignore-conflict relationship between extended transactions t_1 and t_2.*

To illustrate, consider *altruistic locking* [SGMS94], an extension to two-phase locking (2PL) that accommodates long-lived transactions. Under 2PL, short transactions can encounter serious delays, since a long-lived transaction ties up database resources for significant lengths of time. In altruistic locking, an application can *donate* a data object held by extended transaction t_1 that it will no longer access, thus allowing other transactions to access it (certain constraints apply, but will be omitted from this discussion for brevity). Donating a data object does not release the lock that t_1 holds on the data object, but simply allows other extended transactions to acquire a conflicting lock on the data object. Transaction t_1 must still explicitly unlock data items that it has donated – thus, t_1 is free to continue locking data items even after some have been *donated.*

```
ignore_conflict(t1, ALL, obName, NULL, NULL, AD, DONATE)      (1)
```

To realize lock donation, the application would simply create an ignore-conflict record for extended transaction t_1, as illustrated above in Line 1, specifying that any extended transaction can obtain a conflicting lock on the data object obName. The application has effectively *donated* the data object obName held by t_1 to any other extended transaction that requires access to it.

Another brief but illustrative example of the ignore-conflict operation is the formation of cooperative groups [NZ90, MP92, RKT+95]. In a cooperative group the member transactions collaborate over shared data objects while maintaining the consistency of the data objects. Consistency of the data objects can be maintained if other transactions that do

not belong to the group are serialized with respect to all the transactions in the group. Thus, conflicting operations are permitted as long as the conflicting transactions are in the same cooperative group:

```
ignore_conflict(t1, t2, ALL, NULL, NULL, ND, NULL)     (2)
ignore_conflict(t2, t1, ALL, NULL, NULL, ND, NULL)     (3)
```

In the example above, extended transactions t_1 and t_2 are members of a cooperative group. In Line 2, t_1 creates an ignore conflict record, specifying that t_2 can obtain a conflicting lock on any data object that t_1 holds. Similarly, in Line 3, t_2 creates an ignore-conflict record, specifying that t_1 can obtain a conflicting lock on any data object that t_2 holds. If other extended transactions, not members of this cooperative group, attempt to access a data object held by either t_1 or t_2, they will receive a lock conflict.

An application can then specify that it wishes to use explicit cooperation for semantic transaction synchronization for an extended transaction using the command `select_table`$_{t1}$ `(ignoreconflict)`. The command will append the keyword string "IGNORECONFLICT" to the *sbcc_policy* field of t_1's extended transaction descriptor. At runtime, the CONFLICT ADAPTER will use the ignore conflict records in CoopTr_Set_{t1} to relax any R/W conflict for t_1.

Adding Semantic Transaction Synchronization

In our design, the CONFLICT ADAPTER *extends* the fixed transaction synchronization mechanism of the underlying TP monitor to support semantic transaction synchronization. Operationally, the Lock Manager of the TP monitor and the CONFLICT ADAPTER of the Reflective Transaction Framework combine to form a two-step semantic conflict test. Step one, executed by the lock acquisition mechanism of the Lock Manager, performs the standard conflict test based on the type of the operation (e.g. read or write). If the Lock Manager detects a R/W conflict for an extended transaction, it will raise a conflict event to the CONFLICT ADAPTER. Step two, then, is executed by the CONFLICT ADAPTER, which will perform semantic compatibility testing using a *semantic conflict rule* to determine if the conflict can be relaxed.

The *semantic conflict rule* states that an extended transaction t_i may acquire a lock if R/W CONFLICTS with all other transactions owning the lock in a mode incompatible with t_i are relaxed by either a selected compatibility table(s) or an explicit cooperation agreement between the conflicting transactions. The generality of this relaxed conflict rule allows the CONFLICT ADAPTER to present and change the definition of conflict for one or more underlying data objects or transactions selectively.

Definition 3.8 (Semantic Conflict Rule) *A* R/W *conflict detected by the lock acquisition mechanism of the underlying TP monitor can be relaxed (i.e., is semantically compatible) if either of the following conditions are true:*

1. *The semantics of the data object indicate that the operation for which the lock is being requested is compatible with the uncommitted operation holding the lock in an incompatible mode.*

2. *The transaction holding the lock on the data object has explicitly indicated that the transaction requesting the lock has permission to perform the operation, regardless of the basic conflict.*

When operations conflict, the order of access to the data object may imply a dynamic dependency between the extended transactions that must be recorded and tracked. If a named transaction dependency is specified in either the semantic compatibility table or ignore-conflict record, this dependency will be recorded using the services of the TRANSACTION MANAGEMENT ADAPTER. Finally, if an event name is specified in either the semantic compatibility table or ignore-conflict record, the event will be raised to determine if the conflict can be relaxed.

In summary, an application that wishes to use semantic transaction synchronization must first *register* with the CONFLICT ADAPTER using the command **setp(t_1, sbcc_enabled,** TRUE). Next, the application will either load specified compatibility tables using the **load_table** command or establish ignore-conflict relationships between extended transactions using the **ignore_conflict** command. Finally, the application will select the semantic specifications for an extended transaction using the **select** command. During execution, if a R/W conflict is detected by the Lock Manager the transaction processing system will raise a conflict event to the CONFLICT ADAPTER which will then perform semantic conflict testing using the selected semantic specification.

To enable an application to define and select semantic compatibility definitions for individual extended transactions, the CONFLICT ADAPTER provides the following operations. Where appropriate, selected status codes for each operation are provided.

- **load_table(t_1:etrid, pathname:string, name:string, class:string)**: Loads the named compatibility table from the supplied pathname, assigns the unique name to the table and stores the class name.
  - **Could not load specified compatibility table.**
  - **Table name is not unique.**
  - **Object name not specified in table.**
  - **Index field not specified in table.**

- `remove_table(t1:etrid, name:string)`: Removes the named compatibility table.
  - `Table not found.`
- `ignore_conflict(t1:etrid, t2:etrid, objName:string, opName:string, event:string, depName:string, handle:string)`: Creates an ignore conflict record for extended transaction t_1 and places it in CoopTr_Set$_{t1}$. The result of this operation is that an ignore-conflict relationship is formed between t_1 and t_2. The parameters `objName`, `opName`, `event`, `depName` and `handle` are optional.
  - `Duplicate ignore conflict record.`
- `remove_icrecord(t1:etrid, handle:string)`: Removes the specified ignore-conflict record from the CoopTr_Set for extended transaction t_1.
  - `Could not find specified entry.`
- `clear_icset(t1:etrid)`: Removes all ignore-conflict records from the CoopTr_Set for extended transaction t_1.
- `select_table(t1:etrid, name:string)`: Appends the name to the sbcc_policy field in the extended transaction t_1's extended transaction descriptor. The *sbcc_policy* field lists the sources to be checked by the CONFLICT ADAPTER to relax a conflict. The keyword "ignoreconflict" indicates the ignore conflict records in CoopTr_Set$_{t1}$ to use, otherwise the name specifies the class name of semantic compatibility table(s).
- `clear_policy(t1:etrid)`: Clears the sbcc_policy field for extended transaction t_1, setting it to null.

The CONFLICT ADAPTER places two requirements on a TP monitor, in particular the Lock Manager, to support semantic transaction synchronization. First, the Lock Manager must generate a *conflict event* when R/W conflicts are detected, so the CONFLICT ADAPTER can perform semantic conflict testing. Second, the Lock Manager must allow the CONFLICT ADAPTER to affect the decision to ignore the R/W conflict raised by the conflict event. It is our observation, however, that these requirements are reasonable for modern transaction processing systems.

A conventional Lock Manager detects R/W conflicts by comparing the overall lock status and the mode in which the lock has been requested. When the lock acquisition mechanism detects a R/W conflict, it typically passes the conflicting request on for further analysis, to determine if the conflict is real or whether the lock request should be granted — a R/W conflict may be not be a real conflict if, for example, the requesting transaction already has a lock on the data item or is part of a nested transaction that is holding the lock. In these cases, the function informs the lock acquisition mechanism to grant the lock request. In a sense, then, the Lock Manager already generates a conflict event in an attempt to relax R/W conflicts, and we are simply generalizing the processing of this event

for semantic compatibility testing. In fact, commercial TP monitors, such as Transarc's Encina [Tra94a] and BEA's Tuxedo [Lab93], as well as research database systems, such as the Exodus extensible database system [CDG+90] and the Open OODB Project [WBT92], already allow application programs to register functions to relax detected R/W conflicts. For these systems the implementation of semantic transaction synchronization is rather straightforward.

3.3.3 Transaction Execution Control

Fundamental to many advanced transaction models is the ability to place constraints on the execution of individual transactions. Transaction *dependency rules*, expressed in terms of transaction significant events, provide a convenient way to control the execution of concurrent extended transactions. Simply put, dependency rules are constraints on the execution of the *significant events* associated with an extended transaction. We begin our discussion of the services the TRANSACTION MANAGEMENT ADAPTER provides for transaction execution control with a description of dependency rules.

Dependency Rules

The *dependency rules* used in the TRANSACTION MANAGEMENT ADAPTER are based on the work of Johannes Klein [Kle91] and the formalism introduced in the ACTA model [CR91a, CR94]. Following [Kle91] and [CR92], we specify dependencies as constraints on the occurrence and temporal order of certain transaction significant events. However, unlike ACTA, the TRANSACTION MANAGEMENT ADAPTER does not use dependency rules merely to *specify* the interactions between the transactions in an advanced transaction model, but as the basis for synchronizing and coordinating extended transactions at runtime. We build on the following two dependency primitives proposed by Klein [Kle91]:

1. $e_1 \rightarrow e_2$: If e_1 occurs, then e_2 must also occur. There is no implied ordering on the occurrences of e_1 and e_2. We refer to this as a *causal dependency.*
2. $e_3 \prec e_4$: If e_3 and e_4 both occur, then e_3 must precede e_4. We refer to this as an *ordering dependency.*

The first primitive defines a *causal dependency* between two events e_1 and e_2 — if event e_1 occurs, then e_2 must also occur. A causal dependency *does not* imply that event e_2 must have already occurred at the time e_1 occurs. Rather, it is sufficient to permit event e_1 to occur if there is reliable knowledge that e_2 will eventually occur, or if event e_2 can be *forced* to occur. The second primitive defines an *ordering dependency* between two events — event e_3 must occur before e_4, otherwise the dependency rule would be violated. To

demonstrate the use of Klein's primitives in the context of transaction execution, consider the two following well-known transaction dependency rules:

Abort Dependency [CR92]: If transaction t_1 is abort-dependent on transaction t_2, then if t_2 aborts then t_1 must also abort. Let the significant events are denoted as $abort_{t1}$ and $abort_{t2}$. Then the abort dependency between t_1 and t_2 can be expressed as $abort_{t2} \rightarrow abort_{t1}$.

Commit Dependency [CR92]: If transaction t_1 is commit-dependent on transaction t_2, then if both transactions commit, t_1 must commit before t_2 commits. Let the relevant significant events be denoted as $commit_{t1}$ and $commit_{t2}$. Then the commit dependency between t_1 and t_2 can be expressed as $commit_{t1} \prec commit_{t2}$.

Klein's primitives can capture most of the important semantic constraints encountered in practice. To illustrate, below is a list of transaction dependencies that have been defined by various advanced transaction model descriptions in the literature, taken from [CR92]. We have presented each as a dependency rule using the appropriate Klein primitive.

Begin Dependency: If transaction t_2 is begin-dependent on transaction t_1, then t_2 cannot begin executing until t_1 has begun. Let the relevant significant events be denoted as $begin_{t1}$ and $begin_{t2}$. Then the *begin dependency* between t_1 and t_2 can be expressed as $begin_{t1} \prec begin_{t2}$.

Begin-on-Abort Dependency: If transaction t_2 is begin-on-abort dependent on t_1, then t_2 cannot begin executing until t_1 has aborted. A *begin on abort dependency* between t_2 and t_1 can be expressed as $abort_{t2} \prec begin_{t1}$.

Begin-on-Commit Dependency: If transaction t_2 is begin-on-commit dependent on t_1, then t_2 cannot begin executing until t_1 has committed. The begin on commit dependency between t_2 and t_1 can be expressed as $commit_{t2} \prec begin_{t1}$.

Weak-Abort Dependency: If transaction t_2 is weak-abort dependent on t_1, then if t_1 aborts and t_2 has not yet committed, then t_2 aborts. Let the relevant significant events be denoted as $abort_{t1}$, $active_{t2}$ and $abort_{t2}$. The weak abort dependency between t_1 and t_2 can be expressed as $(abort_{t1}$ AND $active_{t2}) \rightarrow abort_{t2}$.

Strong Commit Dependency: If transaction t_2 is strong commit dependent on t_1, then if t_1 commits then t_2 must also commit. The strong commit dependency between t_1 and t_2 can be expressed as $commit_{t1} \rightarrow commit_{t2}$.

Termination Dependency: If transaction t_2 is termination dependent on transaction t_1, then t_2 cannot commit or abort until t_1 either commits or aborts. The termination dependency between t_1 and t_2 can be expressed as $(commit_{t1}$ OR $abort_{t1}) \prec (commit_{t2}$ OR $abort_{t2})$.

Exclusion Dependency: if t_1 commits and t_2 has begun executing, then t_2 aborts. The exclusion dependency between t_1 and t_2 can be expressed as $(commit_{t1}$ AND $active_{t2}) \rightarrow abort_{t2}$.

Serial Dependency: If transaction t_2 is serially dependent on transaction t_1, then t_2 cannot begin executing until t_1 has terminated (t_1 either commits or aborts). The serial dependency between t_1 and t_2 can be expressed as $(commit_{t1}$ OR $abort_{t1}) \prec begin_{t2}$.

Force Commit on Abort Dependency: If transaction t_2 is force-commit-on-abort dependent on transaction t_1, then t_2 must commit if t_1 has aborted. The force commit on abort dependency between t_1 and t_2 can be expressed as $abort_{t1} \rightarrow commit_{t2}$.

While this list includes most of the transaction dependencies found in advanced transaction model proposals in the literature, it is *not* exhaustive. Other dependencies that involve transaction significant events besides the BEGIN, COMMIT and ABORT events can be defined. Thus, the TRANSACTION MANAGEMENT ADAPTER provides a command to define new transaction dependencies, based on the constituent events and the Klein primitive that characterizes the type of the dependency.

Operation Definition 3.13 (Define_Dependency) *The command* `define_dependency (dependency_name, event_name`$_a$`, event_name`$_b$`, dtype)` *defines a named dependency between event*$_a$ *and event*$_b$. *The dependency type, dtype, is specified by the Klein primitive:* `causal` *or* `order`.

Thus, as new significant events are associated with extended transactions, the TRANSACTION MANAGEMENT ADAPTER can support the definition of the new transaction dependencies based on these events. First, we describe the command `form_dependency` that is used to form transaction dependencies and discuss sources of transaction dependencies. Then we discuss how the TRANSACTION MANAGEMENT ADAPTER determines whether the newly defined dependency can actually be enforced at runtime.

Operation Definition 3.14 (form_dependency) *When extended transaction* t_1 *issues the command* `form_dependency(t`$_1$`, dependency_name, t`$_2$`, label)`, *a dependency of type* `dependency_name` *is formed between* t_1 *and extended transaction* t_2, *and tagged with the assigned label. The label field is simply a handle that can be used to reference the dependency, and typically is used to record the name of the data object that induced a dynamic dependency.*

For an extended transaction to form a dependency, we must first verify that the command is *well-formed.* Specifically, both extended transactions must have dependency permissions appropriately set and the specified dependency must have been defined using the `define_dependency` command.

Guard 3.6 (Well-Formed Dependency) *For a dependency operation of the form* `form_dependency(t1, dependency_name, t2, label)` *we have the following guard:*

- *dependency_table(dependency_name)* $\neq$ *error* AND
- *dependency_enabled*(t_1) = *True* AND
- *dependency_enabled*(t_2) = *True*

After the guard has verified that the dependency operation is well-formed, the dependency is recorded in a dependency set managed by t_1.

Definition 3.9 (Dependency Set) *The dependency set for an extended transaction* t_1*, denoted by* $DepSet_{t1}$*, is the set of inter-transaction dependencies formed during the execution of* t_1.

Understanding the Sources of Dependencies

Dependencies between extended transactions may be a direct result of the structural properties of a particular advanced transaction model, referred to as a *structural dependency*, or may indirectly develop as a result of the interactions between extended transactions over shared data objects, referred to as a *dynamic dependency.*

Structural Dependency The structure of an advanced transaction model defines its component transactions and the relationships between them. Transaction dependencies can express these relationships, and thus specify the links in the structure. For example, in the Nested Transaction model the parent/child relationship is established at the time the child transaction is *spawned.* This can be expressed by the child transaction, say t_c, establishing a weak-abort dependency on its parent, say t_p: `form_dependency(`t_c`,` WA`,` t_p`, nolabel)`; and, the parent establishing a commit dependency on its child: `form_dependency(`t_p`,` CD`,` t_c`, nolabel)`. The weak-abort dependency (WA) guarantees the abort of an uncommitted child if its parent aborts, while the commit dependency (CD) guarantees that an orphan, i.e., a child transaction whose parent has terminated, will not commit. These structural dependencies would be formed when the child transaction is first created, in the processing of the **Spawn** event.

Similarly, transaction dependencies can be used to define structural relationships between the member transactions of a number of other advanced transaction models. For example, in the Structured Task model [BHMC90, GMGK+91] and Nested Sagas model [GMGK+91], a parent can commit only if its *vital* children commit; that is, a parent transaction forms an abort dependency on each *vital* child transaction. Cooperative Group Transaction models [NZ90, MP92, RKT+95] define similar dependencies between the

transaction coordinating the group and individual member transactions. Individual transactions may also form structural dependencies with other extended transactions if the advanced transaction model supports coupling modes. For example, component transactions of a Saga [GMS87] can be paired according to a compensate-for/compensating relationship [KLS90]. Relationships between a compensated-for and a compensating transaction, as well as those between them and the saga itself, can be specified via begin-on-commit dependency, begin-on-abort dependency, force-commit-on-abort dependency, and strong-commit dependency [CR92]. In a similar fashion, dependencies that occur in the presence of alternative transactions and contingency transactions can also be specified [BHMC90].

Dynamic Dependency Transaction dependencies can also be formed at runtime by the interaction of extended transactions over a shared data object. However, unlike structural dependencies which are determined by the semantics of the particular advanced transaction model, dynamic dependencies are determined by the data object's synchronization properties. As discussed in Section 3.3.2, two operations conflict if the order of their execution matters. Depending on the semantics of the operation and its relationship to other active operations, this conflict can be *relaxed*, but the reordering may produce a transaction dependency. That is, if a conflicting lock request is granted to an extended transaction t_i because of a relaxed conflict, then a dependency must be formed between t_i and each transaction that owns a lock in a mode incompatible with t_i. For example, the recoverability protocol [BR91] defines a compatibility relation in which two operations can be freely reordered, but the reordering produces a commit dependency between the two transactions. Thus, if t_1 invokes an operation p and later a t_2 invokes an operation q on the same data object $obName$, then t_2 can perform q but is commit-dependent on t_1. Using the recoverability relation, the CONFLICT ADAPTER can relax the conflict, but will first record the commit dependency using the command `form_dependency(`t_2`, CD, `t_1`, obname)`.

Adding Transaction Execution Control

We now discuss issues in the design of extended transaction execution control. Specifically, we shall identify what actions the TRANSACTION MANAGEMENT ADAPTER can take to enforce transaction dependencies, and then discuss how to ensure that a dependency rule can be enforced at runtime. We then conclude with application programming interface commands that the TRANSACTION MANAGEMENT ADAPTER provides to support transaction execution control.

Enforcing Transaction Dependencies The TRANSACTION MANAGEMENT ADAPTER acts as a *passive scheduler* that coordinates and synchronizes the execution of transaction significant events such that no transaction dependency is violated. The scheduling is passive in the sense that the TRANSACTION MANAGEMENT ADAPTER does not raise transaction significant events or perform the state changes of an extended transaction by itself. Rather, a transactional application raises a significant event and the TRANSACTION MANAGEMENT ADAPTER decides whether the requested event can be permitted and at what point in time. When a transaction significant event is raised, the TRANSACTION MANAGEMENT ADAPTER can only take the following actions to enforce a transaction dependency:

- ALLOW – the event does not violate any dependencies and is permitted to execute.
- DELAY – the event is *dependent* on some other event so it is delayed until the dependency is resolved.
- REJECT – execution of the event would violate a dependency rule, so the event is rejected and an error returned to the issuing transaction.
- RAISE – there is a dependency such that another event is raised (triggered) prior to allowing the event execution to proceed.

Thus, the TRANSACTION MANAGEMENT ADAPTER can enforce transaction dependency rules by variously *allowing*, *delaying*, *rejecting* or *triggering* events to occur, so that the resulting extended transaction computation satisfies the given dependencies.

The enforceability of a transaction dependency rule depends crucially on the *attributes* of the transaction significant events that occur in it. We now show how event attributes can be naturally incorporated into our approach. The following event attributes were introduced in [ASSR93]: (a) *forcible*: events that the system can initiate; (b) *rejectable*: events that the system can prevent; and, (c) *delayable*: events that the system can delay.

We note, however, that in an implementation of the Reflective Transaction Framework a nondelayable event would also be nonrejectable, because it happens before the TRANSACTION MANAGEMENT ADAPTER learns of it. Intuitively, such a transaction significant event is not attempted, but rather the TRANSACTION MANAGEMENT ADAPTER is notified of its occurrence *after the fact.* Further, it is possible to have nonrejectable but delayable events in the execution of a transaction; for example, the `begin` of a compensating transaction, or the `abort` of a member transaction in a cooperative group. To capture the above restrictions and to reason more easily about the attributes of transaction significant events, we find it useful to introduce the attributes *immediate* and *inevitable* as combinations of the above. We also believe that *triggerable* is a more appropriate name for forcible

events, because of their actual effect during execution – the TRANSACTION MANAGEMENT ADAPTER can merely trigger an event, not force it to complete. Thus our attributes for transaction significant events are as follows. (Triggerability is orthogonal to the other attributes, which are easily seen to be mutually exclusive.)

- *Normal*: events that are delayable and rejectable;
- *Inevitable*: events that are delayable and nonrejectable;
- *Immediate*: events that are nondelayable and nonrejectable; and
- *Triggerable*: events that are forcible.

To illustrate, in Example 3.3 we present an attribute table for the transaction significant events of an ACID transaction.

Example 3.3 (Event Attributes) *The* TRANSACTION MANAGEMENT ADAPTER *may trigger a transaction* `begin`*, but not a* `commit`*. It can reject and delay a* `commit`*, but can neither delay nor reject an* `abort`*. In other words,* `commit` *is normal;* `begin` *is both triggerable and normal.*

EVENT	*Normal*	*Inevitable*	*Immediate*	*Triggerable*
COMMIT	×	–	–	–
ABORT	–	–	×	×
PREPARE	×	–	–	–
BEGIN	×	–	–	×

As described in Section 3.2.4, an event attribute is set using the metalevel command `event_property`, while the triggerable attribute of a transaction event is specified using the metalevel command `can_trigger`.

Some dependency rules, however, cannot be enforced at all. For example, consider the following ordering dependency: $abort(t_1) \prec abort(t_2)$. This dependency rule specifies an ordering between the abort of two transactions. But this dependency cannot be enforced because it specifies an ordering between two *immediate* events that can be triggered at any time, e.g. by a crash of the database or transactional application. The TRANSACTION MANAGEMENT ADAPTER has no control over the decision or the triggering of this event, and hence it *cannot* guarantee this dependency rule. The TRANSACTION MANAGEMENT ADAPTER can only enforce ordering dependencies if the event on the right side of the

dependency rule is a *normal event*, such that the TRANSACTION MANAGEMENT ADAPTER can delay or even reject the corresponding state transition of an extended transaction.

Enforceability is also an issue for causal dependency rules, such as $abort(t_2) \rightarrow commit(t_1)$. This dependency rule is also not enforceable at runtime, since on abort of t_2, in general, it cannot be guaranteed that t_1 will eventually commit. Such a dependency rule can only be enforced by the TRANSACTION MANAGEMENT ADAPTER if the event on the right side is *triggerable*, or is an event that is somehow *guaranteed* to succeed eventually. For example, the TRANSACTION MANAGEMENT ADAPTER could decide that a transaction must eventually be aborted by rejecting its commit. The dependency rule above could also be enforced, e.g., if t_1 is a compensation transaction for which the system guarantees that it will finally commit, even in the case of failures. In general, however, if at least one term of each clause of a causal dependency rule (in conjunctive normal form) is *triggerable*, then the TRANSACTION MANAGEMENT ADAPTER can enforce the dependency rule. To illustrate, although the ordering dependency in the previous example is not semantically correct, it can be part of a more complex dependency, such as: $(abort(t_2) \prec commit(t_1)) \vee abort(t_3)$. In this dependency rule, the term abort(t_3) is triggerable, therefore, the dependency rule can be enforced by the TRANSACTION MANAGEMENT ADAPTER. Note that this is not a necessary condition for the enforceability of a dependency rule, however it is sufficient and can be checked efficiently at runtime.

Guard 3.7 (Enforceable Dependencies) *When a transaction dependency is defined using the command* `define_dependency(dependency_name, event_name`$_a$`, event_name`$_b$`, dtype)`, *the* TRANSACTION MANAGEMENT ADAPTER *will attempt to determine if the dependency can be enforced at runtime, based on the dependency type and the attributes of the associated events. If it cannot determine whether the dependency is enforceable at runtime, it will create the dependency but return a status code warning that the dependency may not be runtime enforceable.*

Operations for Dependency Management

To support the specification and management of transaction dependencies for execution control, the TRANSACTION MANAGEMENT ADAPTER provides the following operations in its metalevel interface. Where appropriate, selected status codes are provided.

- `define_dependency(dependency_name:string, event_name`$_a$`:string, event_name`$_b$`:string, dtype:string)`: Installs a new transaction dependency type for applications to use in controlling the execution of extended transactions. The dependency_name is a string that applications will refer to when forming the dependency, event_name$_a$ and event_name$_b$ are strings identifying the constituent events, and dtype is a string specifying the type of the dependency (`causal` or `order`).

  - `Success`
  - `Dependency name is not unique`
  - `Transaction significant event(s) not defined`
  - `Dependency type not provided or invalid`
  - `Success with error: dependency not runtime enforceable`

- `form_dependency(t_i:etrid, dependency:string, t_j:etrid, label:string)`: Attempts to form a transaction dependency of the specified type between between t_i and t_j. If the dependency is permitted, it assigns the label to the dependency and installs it in the dependency set.
  - `Success`
  - `Transaction not allowed to form dependencies`
  - `Dependency not defined`

- `delete_dependency(t_i:etrid, dependency:string, t_j:etrid, label:string)`: Removes the named dependency between t_i and t_j from the dependency set. If the keyword ALL is used for the name of the dependency, then the Dependency Set of t_i is emptied.
  - `Success`
  - `Dependency not found`

- `enable_dependency(t_i:etrid)`: Sets a boolean flag indicating that t_i can form and participate in transaction dependencies.

- `disable_dependency(t_i:etrid, DependencyType:string)`: Specifies that extended transaction t_i cannot form the named dependency.
  - `Success`
  - `Dependency of this type is already recorded in dependency set`

- `list_dependency(t_i:etrid, buffer:string)`: Returns a list of the transaction dependencies in the Dependency Set of extended transaction t_i. Each dependency is represented in the form: (dependency_name:string, transaction:etrid, label:string).
  - `Success`
  - `Invalid transaction name`

These operations provided by the TRANSACTION MANAGEMENT ADAPTER enable an application to define, record and manage both structural and dynamic transaction dependencies for the execution control of extended transactions.

3.4 Closing Remarks

This chapter presented the Reflective Transaction Framework. First, our design objectives were stated, followed by an overview of the framework architecture and description of the Open Implementation that the framework provides to an underlying TP monitor. Three kinds of implementation opening were discussed, *introspection*, *explicit invocation* and *intercession*, each serving a different purpose. Introspection allows the programmer to

look into selected aspects of the TP monitor and active extended transactions, through an appropriate abstraction layer. Explicit invocation makes available selected transaction processing functionality that was previously hidden. Intercession lets the programmer add transaction extensions to the substrate and make modifications to the conventional transaction processing, all within boundaries defined by the metalevel interface.

The balance of the chapter utilized the Open Implementation provided by the Reflective Transaction Framework to introduce three new extended transaction services – transaction restructuring through resource delegation, transaction synchronization through application-defined conflict, and execution control through the management of transaction dependencies. These extensions are encapsulated in reflective software modules called *transaction adapters*, which are implemented over TP monitor software. Transaction adapters do not duplicate existing functionality to implement these transaction extensions, but instead extend the functionality provided by the TP monitor – that is, the framework *augments* existing transaction behaviors. This not only eliminates unnecessary infrastructure development by building on existing services, but is designed to provide efficient, robust base processing for extended transactions. Another way of looking at this is that we have taken a divide-and-conquer approach of first identifying services for extended transactions and then incrementally implementing functional extensions in individual transaction adapters. This is in sharp contrast to related research efforts that have attempted to define and construct an extended transaction facility from scratch.

Chapter 4

Demonstration

The previous chapter introduced the main abstractions and interfaces which the Reflective Transaction Framework provides for implementing extended transactions, as well as the means by which these can be tailored to meet the needs of specific situations. Small examples were used to illustrate how these were embodied in the framework and used in practice. The purpose of this chapter is to pull together the various ideas encountered earlier by presenting longer, more detailed examples which also serve to illustrate the range of extended transaction behaviors the Reflective Transaction Framework can support.

4.1 Application Structure

While applications are free to select and use framework features in whichever ways are appropriate, there is a general schema which characterizes most transactional applications. The scheme, illustrated in Figure 4.1, has two sections – initialization and general running.

The initialization phase sets up and initializes the various structures that will be used when an application is running. The objective of this initialization phase is to augment the set of available handlers for extended transaction control operations in order to satisfy new application requirements, starting most likely from a published description of an advanced transaction model. There are three principal steps in this process. The first step is to identify the set of transaction significant events associated with the advanced transaction model. This establishes the set of transaction control operations that an application can invoke to control the execution of an extended transaction based on this advanced transaction model. The second step is to define the actions (handler implementation) for these transaction significant events, characterized first in terms of the different types of transaction dependencies (for example, commit dependency and abort dependency), and second, in terms of transactions' effects on data objects (their state and concurrency status, that is, synchronization state). Through the former, one can specify relationships between significant (transaction management) events, such as `begin`, `commit`, `split`, and

join, pertaining to different transactions. Also, conditions under which such events can occur can then be specified precisely as structural dependency rules. The third step is to consider available semantic information which may be used to relax conflicts between extended transactions, beginning with any *ignore-conflict relationships* between extended transactions, identifying those operations with respect to which R/W conflicts do not need to be considered, and conflicts that can be relaxed by specifying the semantics of data accesses performed by an extended transaction. Once handlers for the extended transaction significant events have been defined, the new control operations can be added to the *extended transaction interface* where they will be available for application programmers to use.

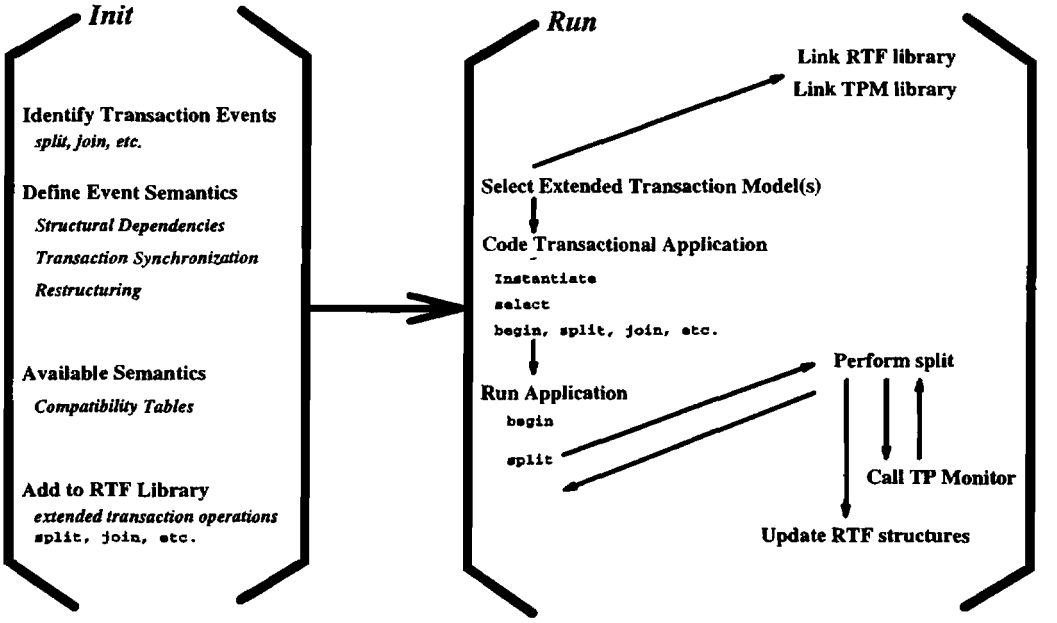

Figure 4.1: Schematic structure of developing transactional applications using the RTF.

Once these initialization tasks have been completed, the extended transaction interface is ready for general use in transactional application programs. In general, application development proceeds as follows. First, an application programmer links their program to the transactional constructs provided by the TP monitor and to the RTF library containing the extended transaction control operations. The available advanced transaction model control operations may be organized into libraries in a number of ways – one library containing the extended transaction control operations for all available extended transaction models, one library for each particular class of application program, or possibly a library for each particular class of advanced transaction model (cooperative, controlled

execution, long-lived, etc.). The application programmer will then code the transactional application, in the 'C' programming language, using the commands `instantiate` to indicate the transactions that require extended transaction services and `select` to select a particular advanced transaction model (i.e., the set of control operations the transaction can invoke at runtime) for each extended transaction. The programmer will then finish coding the application, using commands from both the *base transaction interface* and commands from the *extended transaction interface.*

At runtime, the Reflective Transaction Framework will initialize an extended transaction descriptor with event descriptors for each extended transaction. At some point, triggered either by an extended transaction control operation or a transaction processing event such as a lock conflict, base transaction processing halts and computation will shift from the transactional application to a transaction adapter in the framework for extended transaction processing. The various actions defined for the extended transaction during the initialization phase will be performed, which may involve making function calls to the underlying TP monitor on behalf of the extended transaction. Once extended transaction processing is complete, the data structures managed by the framework will be updated to reflect the newly established state of the extended transaction and control is returned to the transactional application for default transaction processing.

4.1.1 Configuring or Extending the Base Level

Before going on to review commands in the Reflective Transaction Framework's metalevel interface, there is one particular feature pertaining to the use of the base/meta distinction in the framework that is worth exploring. This aspect of the Reflective Transaction Framework's design represents a departure from earlier Open Implementation designs.

In Open Implementations, the separation of base and meta interfaces is normally organized around the distinction between *what* the client application requires of the abstraction, and *how* the abstraction should go about providing (aspects of) that functionality. One way of thinking about this is that the base level sets the terms of the abstraction, while the meta level *configures* that abstraction appropriately for the needs of the client. The meta interface typically deals in terms of different sorts of objects – those used (on some level) to realize (implement) the abstraction. In this way, Open Implementation "opens up" the implementation of the underlying abstractions through the meta interface.

The Reflective Transaction Framework uses Open Implementation to the same end — that is, to allow applications to specialize transaction facilities to their own needs. However, the metalevel objects in the Reflective Transaction Framework – extended transaction descriptors, lock conflicts, delegate sets, semantic compatibility specifications, transaction dependencies, etc.– are abstract. They are quite distant from the implementation level

objects managed by the underlying TP monitor – locks, latches, log records – and much closer to application level semantics. This in turn affects the way in which the metalevel works. Activity at the metalevel in the Reflective Transaction Framework largely specializes base level transaction processing with semantic features of the application domain (such as application-defined conflict, dynamic transaction restructuring or execution control). Once this specialization has been done, those semantic features become available for application programmers to use in base-level (application) programming. In other words, we can think of this not so much as *configuring* the base level, but more as *extending* it. So it is not simply that the framework specializes the structures of the underlying TP monitor and the implementation which lies behind it, but it specializes and extends the *base level* TP monitor services to the needs of the application.

4.1.2 Metalevel Interface Commands

The basic purpose of the *metalevel interface* is to facilitate the implementation of extended transactions. The metalevel interface provides an implementation view of extended transactions, intended for expert transaction system programmers with skills in transaction model specification to implement primitives for new extended transactions. The individual commands in the metalevel interface generalize extended transaction behaviors and allow transaction system programmers to master one set of interfaces that can be used to develop a variety of advanced transaction models and semantics-based concurrency control protocols. The commands in the metalevel interface are summarized below in Table 4.1.

Table 4.1: Summary of Transaction Adapter Command Set (TRACS).

Command	*Description*
`instantiate`	Generates an extended transaction identifier (ETRID) and creates an extended transaction descriptor for the transaction.
`register_event`	Creates a descriptor for a named transaction event and stores the structure in the extended transaction descriptor.
`bind_handler`	Binds a handler to the specified event, recording a pointer to the handler function in the HANDLER field of the event descriptor.
`assign_guard`	Records the name of the guard (predicate) in the GUARDS field of the event descriptor.

continued on next page

continued from previous page

Command	*Description*
`event_property`	Sets the ATTRIBUTE field of the event descriptor to one of the values: *normal*, *inevitable*, or *immediate*.
`event_trigger`	Sets the TRIGGERABLE field of the event descriptor to a boolean value, indicating whether the event can be triggered.
`getetrid_using_name`	Returns the ETRID of the named transaction.
`getetrid_using_trid`	Returns the ETRID of the extended transaction identified by TRID.
`getname_using_etrid`	Returns the name of the extended transaction identified by ETRID.
`getname_using_trid`	Returns the name of the extended transaction identified by TRID.
`gettrid_using_name`	Returns the TRID of the named extended transaction.
`gettrid_using_etrid`	Returns the TRID of the extended transaction identified by ETRID.
`transtate`	Returns the extended transaction state, which is one of: *initiated*, *active*, *pending*, *committed*, *aborted*, or *terminated*.
`set_type`	Sets the optional type of the named extended transaction.
`get_type`	Returns the optional type of the named extended transaction.
`set_state`	Sets the optional application state of the named extended transaction.
`get_state`	Returns the optional application state of the named extended transaction.
`set_etranprop`	Sets the value of the property list identified by *key* to the supplied *value*; if the key is not found, a new property list entry is created.
`get_etranprop`	Searches the property list for *key* and returns the associated value.
`setp`	Sets the value of the selected field to the supplied value.
`getp`	Returns the value of the specified field.
`record_event`	Records that a transaction significant event has occurred by appending the entry (event descriptor, timestamp) to the field EVENTHISTORY.
`find_event`	Searches the EVENTHISTORY field of the extended transaction descriptor for the specified event starting from a specified point in the event history. Returns either the value of the timestamp (event found), or the value 0 (event not found).
`begin_tran`	Begin normal application processing for a specified transaction.
`commit_tran`	Calls TP Monitor commit command for a specified transaction.
`abort_tran`	Calls TP Monitor abort command for a specified transaction.

continued on next page

continued from previous page

Command	*Description*
`thread_to_trid`	Returns the TRID of the transaction associated with the application thread, or an error indicating the thread is not running in the context of an active transaction.
`define_dependency`	Installs a new transaction dependency type for applications to use in controlling the execution of extended transactions.
`form_dependency`	Attempts to form a transaction dependency of the specified type between two extended transactions.
`delete_dependency`	Removes the specified dependency between two extended transactions.
`enable_dependency`	Sets a boolean flag indicating that the extended transaction can form and participate in transaction dependencies.
`disable_dependency`	Specifies that the extended transaction cannot form or participate in the named transaction dependency.
`list_dependency`	Returns a list of the dependencies currently active for the specified extended transaction. Each entry in the list is of the form: dependency_name, etrid, label.
`unlock`	Directs the LOCK ADAPTER to release the lock that the transaction holds on the specified data object.
`lock`	Directs the LOCK ADAPTER to attempt to acquire a lock on the specified data object for the transaction.
`unlock_all`	Directs the LOCK ADAPTER to release all locks currently held by the transaction.
`locks_held`	Returns a list of all locks currently held by the transaction.
`locks_waitfor`	Returns a list of all locks the transaction is waiting to acquire.
`lock_list`	Returns a list of all transactions that hold a lock on the specified data object.
`create`	Creates a named (empty) container for transferring access to and responsibility for data objects from one transaction to another, referred to as a *delegate set.*
`delete`	Deletes a named delegate set.
`insert`	Inserts the name of a data object into a specified delegate set.
`remove`	Removes the name of a data object from a specified delegate set.
`delegate`	Directs the LOCK ADAPTER to transfer ownership of and responsibility for the data objects listed in a specified delegate set from one extended transaction to another extended transaction.

continued on next page

continued from previous page	
Command	*Description*
`acquire`	Directs the LOCK ADAPTER to *complete* the transfer of a deferred delegation, moving the data objects from an intermediary transaction to the delegatee transaction.
`load_table`	Loads the name compatibility table from the specified pathname. Assigns a unique name to the table and records the class name.
`remove_table`	Removes the name compatibility table.
`ignore_conflict`	Creates an ignore-conflict record for an extended transaction and places the record in the cooperative transaction set.
`remove_icrecord`	Removes the specified ignore-conflict record from the cooperative transaction set of the specified extended transaction.
`clear_icset`	Removes all ignore-conflict records from the cooperative transaction set of the specified extended transaction.
`select_table`	Appends the specified name to the SBCC_POLICY field in the extended transaction descriptor. The SBCC_POLICY field identifies sources that are to be checked by the CONFLICT ADAPTER in attempting to relax a R/W CONFLICT.
`clear_policy`	Clears the SBCC_POLICY field for an extended transaction.

Most of the metalevel commands outlined here have been seen, in one form or another, in the discussion and examples laid out in the previous chapter.

4.2 Implementing Extended Transactions

In this section we demonstrate the application of the Reflective Transaction Framework to implement a number of important extended transactions from the literature. These examples serve to demonstrate the various facilities and principles put forth in the previous chapter, putting them together in larger, more detailed examples which demonstrate both the range of the Reflective Transaction Framework and the style of programming it supports. The first set of examples illustrates the implementation of selected advanced transaction models, while the second set illustrates the implementation of selected semantics-based concurrency control protocols. Specifically, we present the implementation of the following advanced transaction models:

- Split and Join Transactions [PKH88];
- Chain Transactions [CR94];
- Reporting Transactions [CR94];
- Cooperative Transaction Groups [RC92, NZ90].

And, the following methods for semantics-based concurrency control:

- Commutativity [Wei88];
- Recoverability [BR91];
- Epsilon-Serializability [RP95];
- Altruistic Locking [SGMS94].

4.2.1 The Split/Join Advanced Transaction Model

In the Split/Join Transaction model [PKH88, KP92] it is possible for an application to *split* an extended transaction t_1 into two transactions, t_1 and t_2, and to *join* two extended transactions t_1 and t_2 into one joint transaction t_2. For simplicity, we will discuss Split Transactions and Joint Transactions as two distinct advanced transaction models.

Split Transactions Split transactions allow an application to dynamically split the database resources held by a (long) transaction into two or more smaller transactions. An application can use split transactions to release partial results, by committing the transaction that has been split off before the splitting transaction is committed. This makes selected changes visible to the other transactions, even though the transaction that made the changes is still in progress. Splitting also allows other short-duration transactions, that are waiting for the data objects released as a result of the partial commitment to proceed. This has the potential for increasing concurrency, as short duration transactions would not be made to wait until the long transaction commits. Such possibilities are especially beneficial for CAD/CAM, VLSI design, and software development applications because of their long-running activities [RC92, CR94].

Extended transactions in the split transaction model are associated with four transaction control operations, namely `begin`, `split`, `commit`, and `abort`. The `begin`, `commit`, and `abort` operations have the same semantics as the corresponding operations of the default ACID transaction model. In our implementation of split transactions, an application *splits* an extended transaction, say t_1, by executing the transaction control operation `split(name`$_{t1}$`, name`$_{t2}$`, objSet)`. Arguments to the `split` command include the name of the split transaction, which must already exist and have an extended transaction descriptor, and the names of data objects that are to be split off, referred to in the literature as the *object set* [PKH88, KP92]. At the time of the split, t_1 will transfer to t_2 the locks on data objects listed in `objSet`. In practice, applications define the object set by selecting the data objects to split from the re-structured transaction. Once the split operation is complete, t_1 and t_2 can `commit` or `abort` independently. In addition, the transactions can further split, creating new split transactions. The following code segment outlines how

the split transaction control operation handler is synthesized using commands from the metalevel interface:

```
split_procedure(tranfrom, tranto, lockList) {
  splitFrom = getetrid_using_name(tranfrom);
  splitTo = getetrid_using_name(tranto);
  if active(transtate(splitTo)) {
    /* create a delegate set for the lock transfer */
    status=create(splitFrom, splitSet, DTOR);
    if (status != success) error(splitFrom, status);
    /* insert locks from lockList into delegate set */
    for each lockname in lockList do {
       status=insert(splitFrom, splitSet, lockname);
       if (status != success) error(splitFrom, status); }
    /* delegate locks */
    status = delegate(splitFrom, splitTo, splitSet, immediate);
    if (status != success) error(splitFrom, status);
    /* create and record event descriptor with timestamp */
    stime = timestamp();
    eventd = strcat(''split:'', tranto);
    recordEvent(splitFrom, eventd, stime);
    /* delete the delegate set */
    status = delete(splitFrom, splitSet)
    /*  return execution control to splitting transaction */
    return(success); }
}
```

Figure 4.2: Definition of the split transaction control operation.

Joint Transactions Extended transactions in the joint transaction model are associated with four transaction control operations, namely, begin, join, commit and abort. The begin, commit and abort operations have the same semantics as the corresponding operations of the default ACID transaction model. The transaction control operation join is a termination event, in addition to the standard commit and abort events. That is, it is possible for an extended transaction, instead of committing or aborting, to *join* another extended transaction. The *joining* transaction transfers its data objects to the *joint* transaction and then terminates. The effects of the joining transaction are made permanent in the database only when the joint transaction commits; otherwise, they are discarded. Thus, if the joint transaction aborts, the joining transaction is effectively aborted.

In our implementation, an application can *join* an extended transaction, say t_1, with another extended transaction t_2 by executing the transaction management operation $\texttt{join}_{t1}(\texttt{name}_{t1}, \texttt{name}_{t2})$. The argument to the `join` command simply identifies the name of the joint transaction, which must already exist and have an extended transaction descriptor. The join procedure transfers all locks held by t_1 to t_2, then terminates the execution of t_1. This is accomplished by first creating a delegate set, inserting the names of all data objects t_1 holds into the set, and delegates the locks. Since the join operation transfers all locks an extended transaction holds, the transaction system programmer will use the argument ALL for the `insert` command. After the delegation is complete, t_2 can freely access the data objects t_1 delegated and is responsible for committing or aborting the effects of t_1. Thus, we synthesize the `join` operation using commands from the metalevel interface as follows:

```
join_procedure(fromtran, totran){
  joinFrom = getetrid_using_name(fromtran);
  joinTo = getetrid_using_name(totran);
  trid_joinFrom = gettrid_using_name(fromtran);
  if active(transtate(joinTo)) {
    /* create a delegate set for the lock transfer */
    status=create(joinFrom, joinDelSet, dtor);
    if (status != success) error(joinFrom, status);
    /* insert the locks currently held */
    status=insert(joinFrom, joinDelSet, all);
    if (status != success) error(joinFrom, status);
    /* delegate locks */
    status = delegate(joinFrom, joinTo, joinDelSet, immediate);
    if (status != success) error(joinFrom, status);
    /* create and record event descriptor with timestamp */
    stime = timestamp();
    eventd = strcat(''join:'', fromtran);
    recordEvent(joinFrom, eventd, stime);
    /* delete the delegate set */
    status = delete(joinFrom, joinDelSet);
    /* commit transaction */
    commit_tran(trid_joinFrom);
    /* return control to invoking application */
    return(success); }
}
```

Figure 4.3: Definition of the `join` transaction control operation.

Once these handlers for the `split` and `join` control operations have been defined using the *metalevel interface*, the operations can be added to the *extended transaction interface* where they will be available for transactional application programmers to use.

4.2.2 The Chain Transaction Model

A special case of the joint transaction model is one that restricts the structure of joint transactions to a linear chain of transactions, which are called *Chain Transactions* [CR94][1]. As with joint transactions, extended transactions in the chain transaction model are associated with four transaction control operations, namely, `begin`, `join`, `commit`, and `abort`. A chain transaction is formed initially by a transaction joining another extended transaction and subsequently by the joint transaction joining another extended transaction. We implement chain transactions by introducing a test in the `join` operation that restricts the invocation such that only linear structures result, as illustrated in Figure 4.4.

```
chain_join_procedure(fromtran, totran){
  joinFrom = getetrid_using_name(fromtran);
  joinTo = getetrid_using_name(totran);
  trid_joinFrom = gettrid_using_name(fromtran);
  if active(transtate(joinTo))
    if firstsplit(joinFrom) {
      /* create a delegate set for the lock transfer */
      status=create(joinFrom, joinDelSet, dtor);
      if (status != success) error(joinFrom, status);
      /* insert the locks currently held */
      status=insert(joinFrom, joinDelSet, all);
      if (status != success) error(joinFrom, status);
      /* delegate locks */
      status = delegate(joinFrom, joinTo, joinDelSet, immediate);
      if (status != success) error(joinFrom, status);
      /* create and record event descriptor with timestamp */
      stime = timestamp();
      eventd = strcat(''join:'',totran);
      recordEvent(joinFrom, eventd, stime);
      /* delete the delegate set */
      status = delete(joinFrom, joinDelSet);
      /* commit transaction */
      commit_tran(trid_joinFrom);
      /* return control to invoking application */
      return(success);
    }
}
```

Figure 4.4: Definition of `join` for the Chain Transaction Model.

Chain transactions can more appropriately capture a reliable computation consisting of a varying sequence of tasks, each of which can execute in the context of a transaction. That is, each task in the computation is structured as a transaction. The beginning of the first extended transaction initiates the computation. The computation expands dynamically

[1] Chain transactions were designed as a more general form of IBM's Chain transactions.

when an extended transaction completes its execution by joining another transaction and hence extending the sequence of transactions. The commitment of any transaction in the sequence successfully completes the computation. The abort of any transaction terminates the computation, and its effects, together with those of all previous transactions in the sequence, are obliterated.

4.2.3 The Reporting Transaction Model

A variation of the joint transaction model is an advanced transaction model in which `join` is not a termination event. That is, a joining transaction continues its execution and periodically *reports* its results to the joint transaction by transferring more data objects to the joint transaction. These transactions are called *Reporting Transactions* [CR94]. As with joint transactions, extended transactions in the reporting transaction model are associated with four transaction control operations — `begin`, `join`, `commit` and `abort`. With the exception of `join`, the definitions of the other control operations are the same as in the joint transaction model. Our implementation of the `join` operation for Reporting Transactions is presented in Figure 4.5.

Following the semantics of the reporting transaction model [CR94], we prevent a reporting transaction from joining more than one transaction and prevent the joint transaction from joining back. Furthermore, to maintain the termination semantics of joining transactions in the joint transaction model we establish an abort-dependency that guarantees the abort of the joining transaction if the joint transaction aborts. Since `join` is no longer a termination event, the reporting transaction must call either `commit` or `abort` to complete their computation.

Reporting transactions provide a more flexible control structure than the joint transaction model for structuring *data-driven computations.* For example, consider a computation that requires remote access to a database over an expensive communication link, such as in a mobile computing environment [IB94]. This computation can be split across the two sites, using reporting transactions where the joining transaction executes on the remote site. The joining transaction accesses the database and performs the initial processing on the data, delegating data objects to the joint transaction only when they need further processing at the remote site.

Variations on the reporting transaction model are possible — for example, reporting transactions can be restricted to a linear form in a manner similar to chain transactions, in which case they would support pipeline-like computations, or allowed to form more complex control structures by permitting a reporting transaction to join more than one transaction.

```
report_join_procedure(fromtran, totran, reportSet){
  reportFrom = getetrid_using_name(fromtran);
  reportTo = getetrid_using_name(totran);
  if active(transtate(reportTo))
    if (firstreport(joinFrom) || repeatreport(reportTo)) {
      /* create a delegate set for the lock transfer */
      status=create(reportFrom, joinDelSet, dtor);
      if (status != success) error(reportFrom, status);
      /* insert locks from reportSet into delegate set */
      for each lockname in reportSet do {
        status=insert(reportFrom, joinDelSet, lockname);
        if (status != success) error(reportFrom, status);
      }
      /* delegate locks */
      status = delegate(reportFrom, reportTo, joinDelSet, immediate);
      if (status != success) error(reportFrom, status);
      /* create and record event descriptor with timestamp */
      stime = timestamp();
      eventd = strcat(''join:'', totran);
      recordEvent(reportFrom, eventd, stime);
      /* delete the delegate set */
      status = delete(reportFrom, joinDelSet);
      /* form an abort dependency with the reporting transaction */
      status = form_dependency(reportFrom, AD, reportTo, report);
      /* return control to invoking application */
     return(success);
  }
}
```

Figure 4.5: Definition of `join` for the Reporting Transaction Model.

4.2.4 The Cooperative Transaction Group Model

The *Cooperative Transaction Group* model was designed to support applications that wish to perform collaborative work [MP92, RC92]. Using this transaction model an application is able to *create* a cooperative group that individual transactions can *join* to share access to data objects. These *member* transactions cooperate to accomplish a single task, and their interactions are structured to reflect the decomposition of the task they are working on together. Because of the cooperative nature of the transaction group, the operations of a single member transaction may not necessarily leave the database in a consistent state. Thus, the effects of member transactions are only made permanent in the database when the entire group commits. If the transaction managing the cooperative group aborts, then all member transactions are forced to abort. Member transactions, however, are allowed to abort independently without forcing the abort of the cooperative group.

Our implementation of cooperative transaction groups defines two types of extended transactions, namely *group* and *member* transactions, each having its own set of transaction control operations. The group transaction can *create* a named cooperative group and is *responsible* for committing the results of the member transactions that have joined the group. A member transaction can *join* a single named cooperative transaction group and share cooperative access to data objects held by member transactions, while executing atomically with respect to the group. Below, we present our implementation of transaction control operations for *group* and *member* transactions.

A *group transaction* is associated with four transaction control operations: **begin**, **commit**, **abort**, and **create_group**. By recording transaction commit and abort dependencies when a member transaction joins a group, our implementation of group transactions can use the default ACID transaction **commit** and **abort** control operations; the TRANSACTION MANAGEMENT ADAPTER enforces transaction group termination dependencies. However, the transaction control operation **create_group** is new and requires a special handler function.

```
boolean_t create_group_procedure(group_name)
/* IN group_name: name of the extended transaction that serves as group tran; */
/* OUT boolean: indicates success of group creation. */
{
/* Initialize the extended transaction descriptor of the named transaction, setting
the group transaction attribute to indicate this transaction is group coordinator, and
initialize the members list. */
   /* get extended transaction identifier for transaction group_name */
   group_etrid = getetrid_using_name(group_name);
   /* set the group transaction attribute, noting this is a group tran */
   set_etranprop(group_etrid, grouptran, IS_GROUP);
   /* set the list of member transactions to null, waiting for members */
   set_etranprop(group_etrid, members, NULL_STR);
   return(TRUE);
   }
 }
```

Figure 4.6: Implementation of the **create_group** operation.

A cooperative group grows through member transactions joining the transaction group. Member transactions may join and leave the cooperative group at any time as the overall task progresses. Since it is not possible to determine the members of a transaction group *a priori*, our implementation of the cooperative transaction group model provides support functions that relate a member transaction to a cooperative group dynamically. These functions can be loosely grouped into two classes. The first class is used to gather information on a cooperative group, such as the function **get_groupid**, which returns the

identifier of the cooperative group that a member transaction belongs to, and the function **get_members**, which returns a list of identifiers of the members of a cooperative group. We synthesize these support functions using commands from the metalevel interface in Figures 4.7 and 4.8.

```
etrid_t get_groupid(member_etrid)
/* IN member_etrid: extended transaction identifier of the member transaction; */
/* OUT group_etrid: extended transaction identifier of the group. */
{
/* Returns the identifier (etrid) of the cooperative transaction group in
which the transaction member_etrid is a member. If a group is not found,
then the constant NOT_FOUND (value 0) is returned. */
  str_groupid = get_etranprop(member_etrid, groupid);
  if (str_groupid != NULL) {
    group_etrid = atol(str_groupid);
    return(group_etrid);
  }
    else return(NOT_FOUND);
} /* End of get_groupid */
```

Figure 4.7: Implementation of the **get_groupid** function.

```
etrid_list_t get_members(group_etrid)
/* IN group_etrid: extended transaction identifier of the group transaction; */
/* OUT *group_etrid: list of extended transaction identifiers. */
{
/* Returns a list of identifiers (etrids) of the member transactions that
belong to the cooperative transaction group group_etrid. */
  member_list = NULL;
  str_member_list = get_etranprop(group_etrid, members);
  if (str_member_list != NULL)
    while (str_member_list != NULL) {
      str_member_etrid = first(str_member_list);
      member_etrid = atol(str_member_etrid);
      append(&member_list, member_etrid);
      remove(&str_member_list, str_member_etrid);
    }
  return(member_list);
} /* End of get_members */
```

Figure 4.8: Implementation of the **get_members** function.

The second class of functions is used to modify a cooperative transaction group, such as the function **add_member** that adds a transaction to a cooperative group, establishes the necessary commit and abort dependencies, and registers the appropriate ignore-conflict records, and the function **drop_member** that removes a transaction from a cooperative group and deletes the associated ignore-conflict records and transaction dependencies. We synthesize these support functions using commands from the metalevel interface in Figures 4.9 and 4.10.

```
boolean_t add_member(member_etrid, group_etrid)
/* IN member_etrid: extended transaction identifier of the member transaction; */
/* IN group_etrid: extended transaction identifier of the group; */
/* OUT boolean: indicates success or failure for the operation. */
{
/* Adds the extended transaction identified by member_etrid to the
cooperative transaction group identified by group_etrid. */
  sprintf(str_member_etrid, "%d", member_etrid);
  sprintf(str_group_etrid, "%d", group_etrid);
  /* First verify group_etrid is in fact a cooperative group transaction. */
  if (strcmp(get_etranprop(group_etrid, grouptran), IS_GROUP) != 0)
     return(FALSE);
  /* Next verify member_etrid does not already belong to a trans group. */
  if (get_etranprop(member_etrid, groupid) == NULL)
     set_etranprop(member_etrid, groupid, str_group_etrid);
  else return(FALSE);
  status = form_dependency(group_etrid, AD, member_etrid, str_group_etrid);
  status = form_dependency(member_etrid, CD, group_etrid, str_group_etrid);
  str_member_list = get_etranprop(group_etrid, members);
  list_bu = str_member_list;
  while (str_member_list != NULL) {
    str_member_etrid = first(str_member_list);
    other_etrid = atol(str_member_etrid);
    status=ignore_conflict(member_etrid, other_etrid, all, all, na, nd, "group");
    status=ignore_conflict(other_etrid, member_etrid, all, all, na, nd, "group");
    remove(&str_member_list, str_member_etrid);
  }
  str_member_list = list_bu;
  strcat(str_member_list, ";");
  strcat(str_member_list, str_member_etrid);
  set_etranprop(group_etrid, members, str_member_list);
  /* member_etrid was successfully added to the cooperative group */
  return(TRUE);
} /* End of group_addmember */
```

Figure 4.9: Implementation of the **group_addmember** function.

```
boolean_t group_dropmember(group_etrid, member_etrid)
/* IN member_etrid: extended transaction identifier of the member transaction; */
/* IN group_etrid: extended transaction identifier of the group; */
/* OUT boolean: indicates success or failure for the operation. */
{
/* Removes the extended transaction identified by the identifier (etrid) from
the cooperative transaction group identified by group_etrid. */
  /* drop group identifier from member transaction */
  set_etranprop(member_etrid, groupid, NULL);
  /* remove member identifier from group transaction */
  sprintf(str_member_etrid, "%d", member_etrid);
  str_group_members = get_etranprop(group_etrid, members);
  remove(&str_group_members, str_member_etrid);
  set_etranprop(group_etrid, members, str_group_members);
  /* remove commit and abort dependencies to prevent group termination deadlock */
  status = delete_dependency(group_etrid, AD, member_etrid, str_group_etrid);
  status = delete_dependency(member_etrid, CD, group_etrid, str_group_etrid);
  /* removal is complete */
  return(TRUE);
} /* End of group_dropmember */
```

Figure 4.10: Implementation of the `group_dropmember` function.

A *member transaction* in the cooperative group model is associated with four transaction control operations: `begin`, `join`, `commit`, and `abort`. Both `begin` and `abort` operations are the same as the ACID transaction model, while the `join` and `commit` operations require additional functionality. Individual member transactions can *join* a named cooperative group when they wish to share data objects with other transactions in that group. Cooperation between the member transactions is specified using ignore-conflict records. Since member transactions executing concurrently may interact with each other in undesirable ways, an application may need to specify that member transactions are adequately isolated from each other. This can be accomplished by specifying a restricted set of data objects and operations over which conflicts can be relaxed in the ignore-conflict record. Conflicts specify how the members' operations cannot be ordered to prevent unwanted side-effects. The CONFLICT ADAPTER ensures that the members interact only in the ways allowable by the active set of ignore-conflict records, and in that way guarantees that the operations by the member transactions as a group leave the database in a correct state. Thus, the ignore-conflict specifications as a whole identify the allowable interleaving of operations in the transaction group's history. Intuitively, a history for a cooperative group is *correct* when it only contains conflicts that conform to the ignore-conflict specifications.

Once a member transaction joins a cooperative group, its eventual commit is determined by the commit of the group transaction. When a member transaction executes the

commit operation, all locks on data objects acquired by the transaction are transferred to the group transaction, as is the responsibility to make the effects on data objects permanent in the database. In this sense, a member transaction only *pseudo-commits* its results when it commits. When a member transaction aborts, it simply releases all locks that it acquired on data objects and the effects of the transaction on those data objects are discarded. Aborting a member transaction may mean that other member transactions need to be aborted as well, either because they read the effects of the aborted transaction, or because the abort caused the history to become incorrect in some way. This requirement is application-dependent and can be easily met by specifying an abort dependency in the ignore-conflict record. We synthesize the new join and commit transaction control operations for a member transaction using commands from the metalevel interface in Figures 4.11 and 4.12.

```
boolean_t join_group_procedure(member_name, group_name){
  member_etrid = getetrid_using_name(member_name);
  group_etrid = getetrid_using_name(group_name);
  set_etranprop(member_etrid, grouptran, IS_MEMBER);
  if (add_member(member_etrid, group_etrid))
     return(TRUE)
  else
     return(FALSE);
}
```

Figure 4.11: Implementation of the member transaction join function.

```
boolean_t commit_member_procedure(member_name){
   member_etrid = getetrid_using_name(member_name);
   group_etrid = get_etranprop(member_etrid, groupid);
   member_trid = gettrid_using_name(member_name);
   /* create delegate set, insert all locks being held, then delegate */
   status = create(member_etrid, commitSet, dtor);
   status = insert(member_etrid, commitSet, all);
   status = delegate(member_etrid, group_etrid, commitSet, immediate);
   if (status != success) error(member_etrid, status);
   status = delete(member_etrid, commitSet);
   /* drop the member transaction from the group */
   set_etranprop(member_etrid, grouptran, NULL);
   if group_dropmember(group_etrid, member_etrid){
     commit_tran(member_trid);
     return(TRUE); }
   else
     return(FALSE);
}
```

Figure 4.12: Implementation of the member transaction commit function.

4.2.5 Operation Commutativity

Operation commutativity is the traditional semantic notion used to determine if two operations can be allowed to execute concurrently [Wei88]. When two operations *commute*, their effects on the state of a data object and their return values are the same, irrespective of their execution order (for example, two **read** operations commute). When using operation commutativity for transaction synchronization, a R/W conflicting operation invoked by a transaction is allowed to execute if it commutes with every other uncommitted operation that holds a lock on the data object. Further, if the transaction processing system allows only commuting operations to execute concurrently, then it prevents cascading aborts.

To implement commutativity we utilize semantic compatibility tables, as described in Section 3.3.2, to identify which operations on a data object are *semantically compatible*. A semantic compatibility table is typically constructed in advance by the database administrator or TP systems programmer based on the semantics of the operations. Each entry of the table is of the form: `[Action, Dependency]`, where `Action` is one of: SOK – the operations are semantically compatible and the conflict can be relaxed, NOK – the operations conflict, or *event* – a named event (predicate) that must be evaluated to determine semantic compatibility, and **Dependency** is a named transaction dependency that is to be recorded between the two corresponding transactions if the conflict is relaxed.

As a simple example, consider operations on a bank account data object for commercial banking applications. For this data type we have the operations **Deposit**, **Withdraw**, and **Balance**. The **Deposit** operation adds a specified amount to the account balance, **Withdraw** subtracts a specified amount from the account balance, and **Balance** returns the current value of the account. From the semantics of these operations the TP systems programmer can construct an operation compatibility table based on commutativity, as illustrated in Table 4.2. Columns in the compatibility table represent operations currently holding a lock, while rows represent operations requesting a lock.

Table 4.2: Operation commutativity for the ACCOUNT data type.

ACCOUNT:OPNAME	**Balance**	**Deposit**	**Withdraw**
Balance	SOK;ND	NOK;ND	NOK;ND
Deposit	NOK;ND	SOK;ND	NOK;ND
Withdraw	NOK;ND	SOK;ND	NOK;ND

4.2.6 Operation Recoverability

Operation recoverability is another semantic notion proposed to relax conflicts among operations, weaker than operation commutativity [BR91]. An operation q is *recoverable*, relative to another operation p, if q returns the same value whether or not p is executed immediately before q. For example, a successful **push** operation on a stack is recoverable relative to a preceding **push** operation on the same stack. Even if the preceding **push** operation is aborted and its pushed value is removed from the stack, the pushed value and the return value of the second **push** operation are not affected. Recoverability demands that transactions involving p and q commit in the order of invocation of the two operations. When used with lock-based transaction synchronization, recoverability, like commutativity, avoids cascading aborts while also avoiding the delay in the processing of many noncommutative operations [BR91].

As with commutativity, we implement operation recoverability using semantic compatibility tables. This is illustrated in Table 4.3 for an ACCOUNT data object, in which the commit dependencies that arise due to recoverability are specified as CD. When the CONFLICT ADAPTER is evaluating a R/W conflict between two extended transactions and relaxes the conflict using recoverability semantics, the commit dependency between the two transactions will be recorded and tracked through the execution of the transactions and used to sequence transaction completion.

Table 4.3: Operation recoverability for the ACCOUNT data type.

ACCOUNT:OPNAME	**Balance**	**Deposit**	**Withdraw**
Balance	SOK;CD	SOK;CD	SOK;CD
Deposit	NOK;ND	SOK;CD	NOK;ND
Withdraw	NOK;ND	SOK;CD	NOK;ND

4.2.7 Epsilon Serializability

Epsilon Serializability (ESR) is a generalization of classic serializability that relaxes R/W conflicts, to *explicitly* allow a bounded amount of inconsistency in transaction processing. The amount of inconsistency is given by some measure of the database operations or a distance function over the database state space [RP95]. In a commercial banking application, for example, inconsistency would be measured in dollars. ESR enhances concurrency by permitting query transactions to read uncommitted data from a concurrent update transaction and by permitting update transactions to write to data items locked by a concurrent query transaction. For example, an epsilon transaction (ET) that can tolerate

a bounded amount of inconsistency, measured in dollars, can query the balance of bank account data objects and execute in spite of ongoing concurrent updates to the database.

In the rest of our discussion, we will use the term ET to refer to both kinds of epsilon transactions: query ETs denoted by Q^{ET}, and update ETs denoted by U^{ET}. A query ET *imports* some inconsistency when it reads a data item while uncommitted updates on that data item exist. Conversely, an update ET *exports* some inconsistency when it updates a data item while query transactions are in progress. Each ET is associated with an inconsistency specification, referred to as an $\epsilon spec$, which is divided into two parts — an *import* inconsistency limit denoted by $\epsilon spec^{ET}_{implimit}$, and an *export* inconsistency limit denoted by $\epsilon spec^{ET}_{explimit}$. For $\epsilon spec^{ET}_{implimit} > 0$ and $\epsilon spec^{ET}_{explimit} = 0$, query ETs may import inconsistency up to $\epsilon spec^{ET}_{implimit}$. For $\epsilon spec^{ET}_{implimit} = 0$ and $\epsilon spec^{ET}_{explimit} > 0$, update ETs may export inconsistency up to $\epsilon spec^{ET}_{explimit}$. If, however, an ET both imports and exports inconsistency, it may introduce new and unbounded inconsistency into the database. Such ETs are the subject of active research and beyond the scope of our implementation work. Our focus is on the situation where query ETs run concurrently with *consistent* update transactions. That is, update transactions are *not* allowed to view uncommitted data and hence *will* produce consistent database states.

Under ESR, a R/W conflicting lock request can be relaxed for an extended transaction if the resulting inconsistency is within the bounds of both import and export limits. Conflict in ESR is formally defined as:

Definition 4.1 (Epsilon Serializability (ESR) and Conflict) *For two extended transactions t_i and t_j, we say that t_i* epsilon-conflicts *with t_j if t_i's lock request for the data object* R/W *conflicts with the lock held by t_j and $\neg Safe(t_i)$. The safety precondition of an extended transaction with respect to performing the operation Oper on data object Obj is defined as follows [RP95]:*

$$Safe(t_i) = \begin{cases} import_{t_i} + import\_inconsistency^{t_i}_{(Oper,Obj)} \leq \epsilon spec^{t_i}_{import} \\ export_{t_i} + export\_inconsistency^{t_i}_{(Oper,Obj)} \leq \epsilon spec^{t_i}_{export} \end{cases}$$

$Import_{ti}$ and $export_{ti}$ are accumulators which record the amount of inconsistency that has already been imported and exported by t_i. And, the value of $import\_inconsistency_{(Oper,Obj)}$ is the maximum amount of inconsistency that t_i can import with respect to performing operation Oper on data object Obj, while $export\_inconsistency_{(Oper,Obj)}$ is the maximum amount of inconsistency exported by t_i performing Oper on data object Obj.

In our implementation, two *inconsistency accumulators* are associated with an extended transaction that utilizes ESR for semantic synchronization: `import_accum` and `export_accum`, which record the total amount of inconsistency the ET has imported and

exported. These accumulators are stored in the extended transaction descriptor using the metalevel command `set_etranprop(etrid,key,value)`, and retrieved using the metalevel command `get_etranprop(etrid,key)`. Similarly, we store the inconsistency specification *espec* associated with the extended transaction: `implimit` and `explimit`, where `implimit` records the $espec^{ET}_{implimit}$ and `explimit` records the $espec^{ET}_{explimit}$. Since we are only concerned with Q^{ET} (`implimit` > 0 and `explimit` $= 0$) and U^{ET} (`implimit` $= 0$ and `explimit` > 0), we maintain only `import_accum` for a query ET and only `export_accum` for an update ET. To bound inconsistency, then, our implementation must ensure for each ET that `import_accum` $\leq$ `implimit` and `export_accum` $\leq$ `explimit`.

Our implementation of ESR follows a two-step methodology: *detection* and *relaxation*. In the first stage, *detection*, we construct a semantic compatibility table that identifies R/W conflicts detected by the Lock Manager that potentially may be relaxed under ESR. In this semantic compatibility table, presented in Table 4.4, columns represent locks held and row locks requested. Under ESR, two concurrent query ETs are always compatible, while two concurrent update ETs are incompatible. Accordingly, the semantic compatibility table entry SOK indicates that two read LOCK requests are compatible, while the entry NOK indicates that two WRITE lock requests conflict. In both of these cases the CONFLICT ADAPTER can immediately determine whether or not to relax the conflict and return. However, entries marked ESR require further processing.

Table 4.4: Compatibility relation based on epsilon-serializability (ESR).

ALL:LOCKMODE	Read(R)	Write(W)
Read(R)	SOK;ND	ESR;ND
Write(W)	ESR;ND	NOK;ND

As described in Section 3.3.2, a lookup in a semantic compatibility table must return one of SOK, NOK, or the name of a predicate to evaluate to determine semantic compatibility. Two entries in our compatibility table hold the value ESR, which is the name of the predicate we will implement to determine if the conflict can be relaxed — if the predicate returns TRUE the conflict will be relaxed.

The definition of the predicate ESR is the second stage of our implementation, *relaxation*, in which we attempt to relax R/W conflicts for an ET using its inconsistency specification and current `import_accum` and `export_accum` values. There are two interesting cases in the implementation of the predicate ESR: first, when a Q^{ET} attempts to read an uncommitted data object that a U^{ET} has modified and, second, when a U^{ET} attempts to update a data object that a Q^{ET} has read. Each of these R/W conflicts, identified in Table 4.4, can be relaxed as long as the resulting inconsistency is within the bounds of both import and export limits of the ETs. Figure 4.13 presents our implementation of the predicate ESR. For simplicity of presentation, we have used the number of R/W conflicts as the inconsistency measure to describe our implementation. Below, we discuss our implementation of the two special cases:

- **Conflict between Q^{ET} and U^{ET}**
  A Q^{ET} has requested a read (R) lock and an active U^{ET} holds the lock in write mode (W). The Q^{ET} will export a certain amount of inconsistency to the transaction holding the lock, so the predicate tests the `import_accum` of the Q^{ET} and the `export_accum` of the conflicting U^{ET} to see if the inconsistency increment is acceptable. If so, the `increment_accum` function is invoked to increment the appropriate *espec* values for the interfering transactions and the conflict is ignored. If either the U^{ET}'s `export_accum` exceeds its `explimit` or the Q^{ET}'s `import_accum` exceeds its `implimit`, then we must prevent the lock from being granted.

- **Conflict between U^{ET} and Q^{ET}**
  A U^{ET} requests a write (W) lock and an active Q^{ET} holds a conflicting R lock. We first check to see if the inconsistency introduced by the U^{ET} requesting the lock will invalidate the query by the Q^{ET} holding the lock. If the inconsistency can be tolerated, the `increment_accum` function is invoked to update the inconsistency accumulators and the conflict is ignored; otherwise, we prevent the lock from being granted.

```
boolean_t esr(tridhold, modehold, lockname, tridreq, modereq)
/* IN tridhold: identifier of transaction holding lock; */
/* IN modeheld: mode lock is being held; */
/* IN lockname: logical lock name; */
/* IN tridreq: identifier of transaction requesting lock; */
/* IN modereq: mode lock is being requested; */
/* OUT boolean: relax conflict (true) or not (false). */

/* Measure inconsistency by number of conflicts */
#define inconsistency 1

{
etridreq = getetrid_using_trid(tidreq);
etridhold = getetrid_using_trid(tidhold);

/* conflict between a query transaction requesting a read lock and an update transaction
holding a write lock. Verify the resulting increase in inconsistency will be tolerated. */

if ((modereq == LOCK_MODE_READ) && (modeheld == LOCK_MODE_WRITE))
  if valid_tolerance(etridhold, etridreq, inconsistency) {
    increment_accum(inconsistency, etridhold, etridreq);
    return TRUE;
  }
  else return FALSE;

/* conflict between an update transaction requesting a write lock and a query transaction
holding a read lock. Verify the resulting increase in inconsistency will be tolerated. */

if ((modereq == LOCK_MODE_WRITE) && (modeheld == LOCK_MODE_READ))
  if valid_tolerance(etridreq, etridhold, inconsistency) {
    increment_accum(inconsistency, etridreq, etridhold);
    return TRUE;
  }
  else return FALSE;

return FALSE; /* Unable to relax conflict. */
} /* End of ESR */
```

Figure 4.13: Implementation of the predicate ESR.

```
boolean_t valid_tolerance(update_etrid, query_etrid, amount)
/* IN update_etrid: etrid of update ET; */
/* IN query_etrid: etrid of query ET; */
/* IN amount: amount of inconsistency being introduced; */
/* OUT boolean: within epspec limits (true) or exceed limits (false). */

{
/* Get current import inconsistency and import limit using
get_etranprop, then convert return string(s) to long integer */
str_import = get_etranprop(query_etrid, import_accum);
current_import = atol(str_import);
str_limit = get_etranprop(query_etrid, implimit);
import_limit = atol(str_limit);
/* Now get current export inconsistency and export limit */
str_export = get_etranprop(update_etrid, export_accum);
current_export = atol(str_export);
str_limit = get_etranprop(update_etrid, explimit);
export_limit = atol(str_limit);

/* Perform epsec verification */
if ((current_import + amount) > import_limit) return FALSE;
if ((current_export + amount) > export_limit) return FALSE;
/* Passed espec tests, so return true to indicate valid tolerance */
return TRUE;
} /* End of valid_tolerance */
```

Figure 4.14: Implementation of `valid_tolerance` function.

Summing up our implementation of ESR, we store three new pieces of information (`import_accum`, `export_accumu` and either `implimit` or `explimit`) with each extended transaction. This is accomplished by using the metalevel commands `set_etranprop` and `get_etranprop`. The implementation itself is carried out in two steps: In the first step, we constructed a semantic compatibility table, presented in Table 4.4, that identifies R/W conflicts that may potentially be relaxed under ESR. This step is similar to our implementations of commutativity and recoverability, except that the semantic compatibility table identifies a predicate to evaluate to determine semantic compatibility. The second step of our implementation was to define the predicate ESR, presented in 4.13, that determines if the conflict can be relaxed using the ET's inconsistency specification and current `import_accum` and `export_accum` values. If the resulting inconsistency is within the bounds of both import and export limits of the ET, the inconsistency accumulators are incremented and the conflict is allowed.

```
void increment_accum(amount, update_etrid, query_etrid)
/* IN amount: amount of inconsistency being introduced; */
/* IN update_etrid: etrid of update ET; */
/* IN query_etrid: etrid of query ET; */

{
/* Get current import inconsistency value using get_etranprop,
then convert string to long integer */
/* Get current import inconsistency value */
str_import = get_etranprop(query_etrid, import_accum);
current_import = atol(str_import);
/* Get current export inconsistency value using get_etranprop,
then convert string to long integer */
str_export = get_etranprop(update_etrid, export_accum);
current_export = atol(str_export);

/* Calculate new import inconsistency level and store using get_etranprop */
new_import = current_import + amount;
sprintf(str_import, "%d", new_import);
set_etranprop(query_etrid, import_accum, str_import);
/* Calculate new export inconsistency level and store using get_etranprop */
new_export = current_export + amount;
sprintf(str_export, "%d", new_export);
set_etranprop(update_etrid, export_accum, str_export);
} /* End of increment_accum */
```

Figure 4.15: Implementation of `increment_accum` function.

There are a number of strategies for measuring the amount of inconsistency that a conflict will introduce, more detailed than the one presented here [WYP92, RP95, LHP94]. The selection of an appropriate inconsistency measure is dependent on both the application and database [WYP92]. However, once an inconsistency measure has been selected, the implementation can be accomplished by simply replacing the constant value `inconsistency` in the implementation presented here with a function that computes the inconsistency measurement.

4.2.8 Altruistic Locking

Altruistic locking [SGMS94] is an extension to two-phase locking that is designed to accommodate long-lived transactions. Under two-phase locking, short transactions may encounter serious delays when a long transaction ties up database resources for a significant length of time. In altruistic locking, several transactions can hold conflicting locks on a data object if constraints AL1 and AL2 in Table 4.5 are satisfied. In two-phase locking a *well-formed* transaction always locks data objects before accessing them, and does not lock any new data objects once it has unlocked a data object. Under altruistic locking an application can use the donate operation, a new extended transaction control operation, to announce that it will no longer access a given data item, thus allowing other extended transactions to access it. The donate operation is not an unlock, so the transaction retains its lock on data objects that it has donated and is free to continue locking other data objects. Donate operations are optional and are used to permit extended transactions to lock a donated data object before the original extended transaction unlocks it.

An extended transaction t_j enters the *wake* of another extended transaction t_i when t_j locks a data object that has been donated, but not yet unlocked, by t_i. An extended transaction t_j is *completely* in the wake of t_i if all the objects it locks are donated by t_i. If t_j locks a data object that has been donated by t_i, then 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. For example, even though two read locks are compatible the second read becomes indebted to the first when an intervening write occurs between the two reads. The altruistic locking protocol presented in [SGMS94] upgrades all read locks to write locks solely to preserve the indebted relationship between transactions. Instead of altering the locks held by an extended transaction, our implementation will maintain several sets for each database object *obname* and transaction t_i, as identified in Table 4.5.

Table 4.5: Altruistic locking requirements.

AL1	Two extended transactions may not simultaneously hold *conflicting* locks on the same data object unless one first *donates* the data object.
AL2	If extended transaction t_i is *indebted* to extended transaction t_j, then t_i must be completely in the wake of t_j until t_j terminates.
D(OBNAME)	Set of transactions that have donated, but not released their lock on *obname*.
IN(OBNAME)	Set of transactions that readers of *obname* must be in the wake of.
W(T_i)	Set of transactions whose wake that t_i is completely within.
J(T_i)	The set of transactions whose wakes t_i *should* be completely within (based on AL1 and AL2).

We introduce IN(OBNAME) in our implementation to replace both RL(OBNAME) and WL(OBNAME) specified in the original model definition [SGMS94]. The framework maintains information on the wake of a transaction (i.e., W(T) and J(T) for each extended transaction) and enforces the indebted constraint AL2. Initially, for all data objects *obname* and any extended transaction t_i, $J(t_i)$ = D(OBNAME) = IN(OBNAME) = NULL. By default, when an extended transaction begins, it enters the wake of all active transactions; transactions are removed and inserted into $J(t_i)$ based upon the behavior of t_i.

Under altruistic locking a transaction is associated with the usual control operations, namely **begin**, **commit**, and **abort**, along with a new operation **donate**. The handler for donate is defined in Figure 4.16 – the function simply records that a transaction has donated its lock on a specified data object.

```
void donate_procedure(tran_name, lock_name)
/* IN tran_name: name of the extended transaction donating the lock. */
/* IN lock_name: name of the lock being donated. */
{
   /* log that the extended transaction donated its lock on lock_name */
   tran_etrid = getetrid_using_name(tran_name);
   add_member(D[lock_name], tran_etrid);
}
```

Figure 4.16: Implementation of the altruistic locking **donate** function.

The framework initializes the structure $W(t_i)$ by tracking the set of active extended transactions. To register an extended transaction a call to the procedure **begin_al_tran** is placed in the handler for the begin control operation.

```
void begin_al_tran(tran_name)
/* IN tran_name: name of the extended transaction. */
{
   tran_etrid = getetrid_using_name(tran_name);
   /* initialize the wake list W to all active transactions */
   copy_list(copy_active, &active_set);
   while (first(copy_active) != null_etrid) {
     active = first(copy_active);
     add_member(W[tran_etrid], active)
     remove(&copy_active, active);
   }
   /* initialize J to NULL */
   J[tran_etrid] = NULL
   /* add this transaction to the list of active transactions */
   insert(&active_set, tran_etrid);
}
```

Figure 4.17: Implementation of the **begin_al_tran** function.

When an extended transaction terminates, it calls `complete_tran` to update the list of active transactions.

```
void complete_tran(term_etrid)
/* IN term_etrid: etrid of the extended transaction that is terminating */
/* OUT no values returned */
/* Transaction term_etrid can no longer have any impact on other extended */
/* transactions, so update the appropriate W(term_etrid) sets */
{
   /* first remove transaction from the active transaction list */
   remove(&active_set, term_etrid);
   /* copy the active transaction list for processing */
   copy_list(copy_active, &active_set);
   while (first(copy_active) != null_etrid) {
     tranetrid = first(copy_active);
     /* update the wake list W */
     if member(W[tranetrid], term_etrid)
        remove(W[tranetrid], term_etrid);
     remove(&copy_active, tranetrid);
   }
}
```

Figure 4.18: Implementation of the `complete_tran` function.

To manage the lists J(T) and W(T), a callback to the function `lock_after` is made *after* a lock is granted; these sets cannot be updated beforehand, as a locking conflict that failed to set a lock would incorrectly update this information. In addition, a callback is also attached to the unlock function to manage the donate set D(OBJNAME).

```
void lock_after(trantrid, objectname)
/* IN trantrid: etrid of extended transaction that acquired the lock; */
/* IN objectname: name of the data object that was locked; */
{
   /* Update the wake list J[tran_etrid] */
   tran_etrid = getetrid_using_trid(trantrid);
   list_union(J[tran_etrid], IN[objectname], &temp_list);
   J[tran_etrid] = temp_list;
   /* Update the wake list W[tran_etrid] */
   list_intersect(W[tran_etrid], D[objectname], &temp_list);
   W[tran_etrid] = temp_list;
}
```

Figure 4.19: Implementation of the `lock_after` function.

```
void after_unlock(trantrid, objectname)
/* IN trantrid: etrid of extended transaction that acquired the lock; */
/* IN objectname: name of the data object that was locked; */
{
   /* Removes downstream transactions from the wake of trantrid */
   /* and maintains IN[objname] and D[objname].                  */
   tran_etrid = getetrid_using_trid(trantrid);
   remove(D[objname], tran_etrid);
   remove(IN[objname], tran_etrid);
   copy_list(copy_active, active_set);
   while(first(copy_active) != null_etrid) {
     worketrid = first(copy_active);
     if member(J[worketrid], tran_etrid)
        remove(J[worketrid], tran_etrid);
   }
}
```

Figure 4.20: Implementation of the **after_unlock** function.

Our implementation of altruistic locking is not complete, however, without some way of specifying the conflicts that can be relaxed. Entries in the altruistic locking compatibility table, presented in Table 4.6, hold the values AL1 and AL2, which are the names of the predicates we implement to determine if a conflict can be relaxed.

Table 4.6: Compatibility relation based on altruistic locking.

ALL:LOCKMODE	Read(R)	Write(W)
Read(R)	SOK;ND	AL1;AD
Write(W)	AL1;AD	AL2;AD

The AL1 predicate is invoked for all R/W conflicts on any data object. The predicate allows an extended transaction to obtain a read or write lock on a data object that was donated, and maintains the indebted relationship. The AL2 predicate allows multiple writers if the conflicting object was donated first. In both cases, predicates AL1 and AL2, an abort dependency is established between the two extended transactions to prevent the abnormal termination of the donating transaction from introducing inconsistency into the database system.

```
boolean_t al1(tridhold, modehold, lockname, tridreq, modereq)
/* IN tridhold: identifier of transaction holding lock */
/* IN modeheld: mode lock is being held */
/* IN lockname: logical lock name */
/* IN tridreq: identifier of the transaction requesting lock */
/* IN modeheld: mode lock is being requested (not used) */
/* OUT boolean: relax conflict (true) or not (false) */
{
etridreq = getetrid_using_trid(tridreq);
etridhold = getetrid_using_trid(tridhold);
/* check if the lock has been donated by etridhold */
if (is_donated(lockname, etridhold)) {
  /* enter etrid into front of the wake */
  update_in_set(lockname, etridhold, modehold);
  return TRUE:
}
return FALSE;
}
```

Figure 4.21: Implementation of the predicate AL1.

```
boolean_t al2(tridhold, modehold, lockname, tridreq, modereq)
/* IN tridhold: identifier of transaction holding lock */
/* IN modeheld: mode lock is being held */
/* IN lockname: logical lock name */
/* IN tridreq: identifier of the transaction requesting lock */
/* IN modeheld: mode lock is being requested (not used) */
/* OUT boolean: relax conflict (true) or not (false) */
{
etridreq = getetrid_using_trid(tridreq);
etridhold = getetrid_using_trid(tridhold);
if (is_donated(lockname, etridhold, modehold)) return TRUE;
else return FALSE;
}
```

Figure 4.22: Implementation of the predicate AL2.

The implementation of the support function **wake_test**, **is_donated**, and **update_in_set** for the predicates AL1 and AL2 is outlined below.

The function **update_in_set** maintains the indebted relationship by recording which transactions access a donated data object in a conflicting (write) mode.

```
void update_in_set(object_name, tranetrid, modeheld)
{
   /* does this read request conflict with a write lock? */
   if (modeheld == write_type)
     add_member(IN[object_name], tranetrid);
}
```

Figure 4.23: Implementation of the **update_in_set** operation.

```
boolean_t wake_test(etrid, lockname, lockmode)
{
  /* Return TRUE if etrid is not completely in the wake of another    */
  /* transaction.  Otherwise, return TRUE if etrid remains completely */
  /* in the wake of J[etrid].                                         */
  list_intersect(W[etrid], D[objname], &temp_list1);
  list_union(J[etrid], IN[objname], &temp_list2);
  if subset(temp_list1, temp_list2)
    return TRUE;
  else
    return FALSE;
}
```

Figure 4.24: Implementation of the predicate WAKETEST.

The function **is_donated** searches the list of transactions that have donated their lock on a data object and returns TRUE if the specified extended transaction is found.

```
boolean_t is_donated(object_name, tranetrid)
{
   /* Check whether the transaction donated this data object */
   if member(D[object_name], tranetrid)
     return(TRUE)
   else
     return(FALSE);
}
```

Figure 4.25: Implementation of the **is_donated** operation.

4.3 Application Development Using Extended Transactions

In this section we demonstrate how an application programmer can use the extended transaction interface to implement a transactional application using extended transactions. These are not intended as examples of real-world applications, but rather serve to illustrate the use of the extended transaction interface and the style of application programming that it supports. The first example outlines the implementation of an application using an advanced transaction model. The second example outlines the implementation of an application using semantics-based concurrency control protocols.

4.3.1 Programming Using an Advanced Transaction Model

To motivate the application of an advanced transaction model, consider the requirements of CAD support for a team of engineers designing a computer chip. Since the design process may take an arbitrarily long time and involve multiple engineers, at some point in the project the principal engineer might like to split off responsibility for the design of specific subsystems to component engineers. These component engineers can either join their results back into the working chip design at a later time, or choose to commit or abort their designs independently. Such requirements are not satisfied by traditional database transactions in a straightforward manner, but can be satisfied by the split/join transaction model easily. The code fragment below outlines how an application programmer might use the split and join operations to restructure a transaction dynamically to release subsystem data objects to a separate extended transaction, and later join with another transaction that performs quality assurance on the design.

```
Begin_Transaction PE_Tran                          (1)
begin
    instantiate(PE_Tran, trid)                     (2)
    select(PE_Tran, splitjoin)                     (3)
    ...
    ...{ data manipulation }
    ...
    split(PE_Tran, CE_Tran, Subsystem)             (4)
    ...
    ...{ data manipulation }
    ...
    join(PE_Tran, QA_Tran, ALL)                    (5)
end
Commit_Transaction {CAD_design}                    (6)
```

Line 1 declares the beginning of the principal engineer's transaction, denoted as *PE_Tran*, using the `Begin_Transaction` command found in the base transaction interface. This is significant, because it notifies the transaction management system that the operations between this point and the `Commit_Transaction` command in line 6 are to be executed atomically, according to the traditional transaction model. Thus, lines 1 and 6 bracket the transaction. The purpose of the `instantiate` metalevel interface command in line 2 is to notify the Reflective Transaction Framework of the programmer's intention to "renegotiate" the base transaction model. The `select` command in line 3 details the terms of the renegotiation, selecting the split/join model for the transaction. The importance of the `select` command is twofold. First, it determines the control operations and semantics that are available to the transaction. In this example, the split/join model adds two new transaction control operations, namely `split` and `join`, while the `begin`, `commit`, and `abort` commands have the same semantics as the corresponding commands in the traditional database transaction model. Second, the `select` command informs the transaction adapters in the Reflective Transaction Framework how to process transaction events on behalf of this transaction, such as lock request conflicts, transaction dependencies that might arise during execution, etc. In line 4 the application programmer uses the new extended transaction control operation `split`, where *CE_Tran* is the name of the transaction that the component engineer is running and *Subsystem* is the name of the subcomponent that is to be delegated to the component engineer's transaction. Finally, in line 5 the application programmer uses the new extended transaction control operation `join` to merge the results and resources held by the transaction *PE_Tran* with an existing quality assurance transaction named *QA_Tran*.

One can see from this example that with the exception of the `instantiate` and `select` operations, the application programmer simply uses familiar transaction control operations to code an application. There is no explicit delegation of the locks held on data objects in *Subsystem*, no need to explicitly relax the lock conflict that arises during the transfer, and no explicit delegation of data objects held by *PE_Tran* when the transaction joins with the quality assurance transaction *QA_Tran*.

Transaction Adapters Behind the Scenes. Continuing with our CAD example, we now examine how transaction adapters work behind the scenes to support extended transaction processing on a legacy TP monitor. We begin with the `instantiate` metalevel interface command in line 2. During execution, the instantiate command causes control to be passed to the TRANSACTION MANAGEMENT ADAPTER, which first generates an

extended transaction identifier and then creates and initializes a descriptor for the transaction, reifying initial state for transaction *PE_Tran*, such as the transaction identifier (TRID) and current execution state of the transaction. When completed the TRANSACTION MANAGEMENT ADAPTER returns control back to the base transaction for processing. The `select` command in line 3 also causes control to be passed to the TRANSACTION MANAGEMENT ADAPTER, which updates the extended transaction descriptor to contain the transaction control operations `split` and `join`, specified by the split/join advanced transaction model.

Processing resumes on the base TP monitor until the transaction control operation `split(PE_Tran, CE_Tran, Subsystem)` is processed in line 4. Split is a transaction control operation defined in the extended transaction interface. When an application invokes a transaction management control operation, the actual code executed is determined by the transaction's extended transaction descriptor. Processing the `split` operation, *PE_Tran* first verifies this control operation is permitted and then calls the handler function. As defined in Section 4.2.1, the split handler of TRANSACTION MANAGEMENT ADAPTER confirms that the extended transaction *CE_Tran* is *active*, creates a named delegate set, and inserts the name of all data objects in *Subsystem*. Once the handler is complete, the LOCK ADAPTER delegates locks on all data objects in the delegate set from *PE_Tran* to *CE_Tran*. It then directs the CONFLICT ADAPTER to create no-conflict records in order to relax lock conflicts that may arise during transfer, and calls the TP monitor API commands `lock` and `unlock` to transfer the locks. Once the transaction restructuring is complete, the TRANSACTION MANAGEMENT ADAPTER returns control to the TP monitor to continue base level transaction processing.

4.3.2 Programming Using SBCC Protocols

An application programmer can construct semantic compatibility tables for objects that are *hot spots* or *concurrency bottlenecks* in an application. Once created, applications can load these compatibility tables for semantics-based transaction synchronization. To illustrate we will continue with our CAD example introduced in the previous section, in which a team of engineers are working together to design a computer chip. During the initial design several component engineers define new components for the chip, performing lookups on existing components, modifying existing specifications, and deleting outdated or unnecessary components. One possible concurrency bottleneck in this activity are data objects of type Component_Log — a container for specifications of the individual components in the chip, each identified by a component identifier (key).

A data object of type Component_Log supports five operations: `insert`, `delete`, `lookup`, `sort`, and `modify`. The operation `insert` adds a new entry of the form (key,

item) into the Component_Log and returns success; if the key already exists in the table it returns failure. The operation **delete** removes the entry with the given key from the Component_Log and return success; if the key is not found it returns failure. The **sort** operation sorts the entries by key value in ascending order. The operation **lookup** searches the Component_Log for an entry that matches the specified key and, if found, returns the value of the item; otherwise it returns failure. The operation **modify** replaces the current value of the item with the new value for the given key.

Table 4.7: Operation commutativity for the COMPONENT_LOG data type.

LOG:OPNAME	insert	delete	lookup	sort	modify
insert	SOK;ND	SOK;ND	SOK;ND	NOK;ND	SOK;ND
delete	SOK;ND	SOK;ND	SOK;ND	NOK;ND	SOK;ND
lookup	SOK;ND	SOK;ND	SOK;ND	SOK;ND	SOK;ND
sort	NOK;ND	NOK;ND	NOK;ND	SOK;ND	NOK;ND
modify	SOK;ND	SOK;ND	SOK;ND	NOK;ND	SOK;ND

Table 4.8: Operation recoverability for the COMPONENT_LOG data type.

LOG:OPNAME	insert	delete	lookup	sort	modify
insert	SOK;CD	SOK;CD	SOK;CD	NOK;ND	SOK;CD
delete	SOK;CD	SOK;CD	SOK;CD	NOK;ND	SOK;CD
lookup	SOK;CD	SOK;CD	SOK;CD	NOK;ND	SOK;CD
sort	SOK;CD	NOK;ND	SOK;CD	SOK;CD	NOK;ND
modify	SOK;CD	SOK;CD	SOK;CD	NOK;ND	SOK;CD

Tables 4.7 and 4.8 illustrate the commutativity and recoverability properties of the operations performed on data objects of type Component_Log; for simplicity, it is assumed that transactions operate concurrently on different parameters (keys) on the objects of type Component_Log. These operation compatibility tables are described in files, for example using a text editor or a graphical data entry tool. The following code fragment shows how an application programmer could load and activate the tables.

```
Begin_Transaction CE_Tran                                    (1)
begin
    instantiate(CE_Tran, trid)                               (2)
    select(CE_Tran, SBCC)                                    (3)
    load_table(CE_Tran, logcomm, logcommtbl, comm)           (4)
    load_table(CE_Tran, logrecv, logrecvtbl, recv)           (5)
    select_table(CE_Tran, logcomm)                           (6)
    select_table(CE_Tran, logrecv)                           (7)
    lookup(CID_87, compspec)                                 (8)
    ...
    ...{ data manipulation }
    ...
    modify(CID_87, compspec)                                 (9)
    ...{ data manipulation }
    insert(CID_109, nullspec)                                (10)
    ...
    ...{ data manipulation }
    ...
    modify(CID_109, compspec)                                (11)
end
Commit_Transaction {CE_Tran}                                 (12)
```

The `Begin_Transaction` command in line 1 declares the beginning of the component engineer's transaction, and together with the `Commit_Transaction` in line 12 brackets the transaction. The command `instantiate` in Line 2 creates an extended transaction descriptor and *registers* the transaction with the Reflective Transaction Framework. The `select` meta interface command in line 3 indicates the application's intention to use semantic information to relax lock conflicts. The `load_table` command in lines 4 and 5 directs the framework to load the specified compatibility tables, logcomm and logrecv (a full file pathname could be supplied), for the extended transaction and assigns a unique name to each. The `select_table` command in lines 6 and 7 specifies the order in which these compatibility tables are to be applied when attempting to relax lock conflicts.

If a R/W conflict is detected by the Lock Manager during transaction execution, the Lock Manager raises a *conflict event* and the CONFLICT ADAPTER is invoked for semantic compatibility testing. For example, if an uncommitted transaction performs a `lookup` operation (holds a read lock) on the data object *compspec* and transaction *CE_Tran* calls the `modify` operation (a write lock request) in line 8, the Lock Manager detects a R/W conflict. Since the CONFLICT ADAPTER registered a handler for the event and *CE_Tran* selected a commutativity table to relax lock conflicts (Table 4.7), the CONFLICT ADAPTER performs a table lookup to determine if the operations are semantically compatible and

can be executed concurrently. If the operations are semantically compatible (SOK), the conflict adapter grants the lock, which enable both transactions to access the data object.

In summary, to use semantics-based concurrency control for transaction synchronization, the application programmer must first create compatibility tables for data objects that have been identified as concurrency bottlenecks, and then registers transactions with the framework and selects from the available semantic compatibility tables. During application execution, the framework permits transactions to perform operations on data objects without conflicting with other transactions that hold locks on the object if the semantic specification relaxes the conflict. In certain cases, where the order of the access to a data object implies dynamic dependencies between transactions, the framework records and tracks transaction dependencies throughout transaction execution.

Transaction Adapters Behind the Scenes Continuing with our example, we now examine how transaction adapters work behind the scenes to support semantics-based concurrency control. The metalevel interface command **instantiate** in line 2 performs the same initialization as our previous advanced transaction model example. The **select** commands in lines 3 and 4 perform two functions. First, they inform the framework of the transaction's intension to utilize semantic information to relax lock conflicts. The TRANSACTION MANAGEMENT ADAPTER responds by registering the CONFLICT ADAPTER as the handler for lock conflict events. Second, they instruct the CONFLICT ADAPTER to load the specified compatibility tables for the transaction. If the file cannot be found or an error occurs loading the file, then the CONFLICT ADAPTER is unregistered and an error code is returned. During the execution of *CE_Tran*, all lock conflict events are handled by the CONFLICT ADAPTER.

During transaction execution, the **Lock** function performs standard conflict testing for all lock requests. If a lock conflict is detected, a conflict event is raised. Information passed to the CONFLICT ADAPTER in the conflict event descriptor includes the identifier of the transaction requesting the lock, the mode in which the lock is being requested, the operation being requested, and a list of the transactions currently holding a lock on the data object. The CONFLICT ADAPTER uses the function **relaxConflict** to implement semantic compatibility testing.

Operationally, **Lock** and **relaxConflict** combine to form a two-step semantic conflict test. Step one, executed by **Lock**, performs a standard syntactic conflict test based on the update type of the operation (e.g. **read** or **write**). Step two, which is performed only when a conflict is detected, is executed by the **relaxConflict** function which performs semantic compatibility testing to determine if the two operations are semantically compatible.

The function `relaxConflict` uses compatibility table(s) that define compatibility relations, and an ignore_conflict table that records conflicts explicitly *relaxed* between transactions, and will relax a R/W conflict if either of the following conditions hold:

1. the semantics of the data object indicate the operation for which the lock is being requested is semantically compatible with all uncommitted operations holding a lock;
2. the transaction holding the conflicting lock has explicitly indicated the transaction requesting the lock has permission to perform the operation.

This semantic conflict rule effectively states that an extended transaction may acquire a lock if all other transactions owning the lock in an incompatible mode are relaxed by either operation semantics or explicit agreement between the transactions. This semantics based concurrency control is all performed through extensions of the underlying conflict detection and locking mechanism, demonstrating that the use of a conventional locking mechanism does not preclude the use of semantics-based concurrency control protocols.

4.4 Summary

Building on the concepts and mechanisms introduced in Chapter 2 and Chapter 3, this chapter presented the application of the Reflective Transaction Framework to implement a number of extended transaction types. These examples vary significantly in their scope, structure and style of interaction. The first set of examples consisted of advanced transaction models that selectively relax the ACID properties in a controlled manner, while the second set consisted of semantics-based concurrency control protocols that employ various forms of semantic information to relax the definition of conflict. Although the behaviors, and hence internal organization, of these extended transaction examples differ considerably, they are all supported within the framework that the Reflective Transaction Framework defines and implements. Application and transaction systems programmers can use the extended transaction and metalevel interfaces to tailor the basic framework mechanisms to match the needs of their particular applications or domains, while maintaining the overall structure of their code and effecting a simple separation between application code, framework use, and framework specialization.

Chapter 5

Implementation and Evaluation

The previous two chapters presented the detailed design of the Reflective Transaction Framework and demonstrated how it can be used to implement a number of extended transaction types. To complete the picture, this chapter presents ENCINA/ET, an implementation of the Reflective Transaction Framework on the commercial TP monitor Encina [Tra94a], and an evaluation of the Encina implementation.

We begin in Section 5.1 with an implementation overview, addressing issues specific to an Encina implementation, and in Section 5.2 we describe the implementation of ENCINA/ET. In Section 5.3 we present an evaluation of ENCINA/ET that quantifies the cost of supporting the extended transaction services, along with a qualitative evaluation of the framework design. We conclude in Section 5.5 with a summary of the experience gained and lessons learned from the implementation and evaluation effort.

5.1 Implementation Chapter Overview

This section presents an overview of ENCINA/ET, an implementation of the Reflective Transaction Framework on the commercial TP monitor Encina [Tra94a]. We begin by describing the overall architecture and main components of the system. Because many of the basic mechanisms of the framework have already received in-depth coverage in Chapter 3, we focus on issues specific to the Encina implementation. These include internal extended transaction representation, connection with the underlying TP monitor, and the implementation of key extended transaction services.

5.1.1 Design of the Encina TP Monitor

Our implementation of ENCINA/ET is constructed on top of the Encina TP monitor, in particular the Encina Toolkit [Tra94b]. The Encina Toolkit, illustrated in Figure 5.1, consists of transaction middleware service modules that provide the core transaction services of the Encina TP monitor, which include:

- Transaction Service Module (TRAN), which provides transaction execution control and default transaction control operations (begin, commit, abort).
- Lock Service Module (LOCK), which provides a logical locking package to guarantee transaction isolation.
- Recovery Service Module (REC), which provides undo/redo logic required to implement roll-back after abort and roll-forward after system failure.
- Log Service Module (LOG), which provides write-ahead log support for transaction updates and crash recovery.
- Volume Service Module (VOL), which provides logic to view multiple physical and mirrored disks as a single virtual file.

In addition, the Encina Toolkit includes the Transactional-C (TRAN-C) library, which consists of macros and routines that enhance ANSI/Standard C for transactional application development. The toolkit also includes the Base Development Environment (BDE) library, which provides services such as POSIX threading, file I/O, and memory allocation to isolate the toolkit from operating system dependencies. With the exception of VOL and LOCK, these transaction services are the basic building blocks present in most modern TP monitors [Ber90, GR93, BN96].

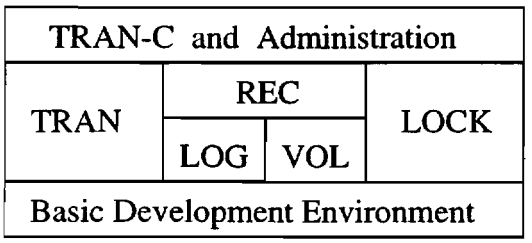

Figure 5.1: Software modules in the Encina Toolkit.

Each module in the Encina Toolkit provides access to its transaction services and behaviors through a relatively simple and uniform application programming interface (API). In addition, each module provides a *transaction event callback facility* in which an application may arrange for a procedure to be called when a selected event occurs during the processing of a transaction. These events include transaction initialization, transaction preparation, transaction resolution, transaction commit, transaction abort, lock conflicts, and others. The procedure callback is made by an Encina library routine, in a thread managed by Encina, when the requested event occurs. From the point of view of the application process, the procedure call happens asynchronously. ENCINA/ET uses the callback facility extensively to coordinate the execution of a transaction running on the

Encina TP monitor with various extended transaction services in ENCINA/ET. In addition, we use the API calls to leverage transaction services of the toolkit to implement the extended transaction services in ENCINA/ET.

5.1.2 Design of ENCINA/ET

ENCINA/ET is implemented as a user-level C library — a collection of functions and an associated header file — residing in the same address space as the transactional application. The ENCINA/ET library is modularly structured. Each module corresponds to a specific transaction adapter, to allow one to experiment with different adapter implementations. The functions in the ENCINA/ET library implement the extended transaction services detailed in Chapter 3, and are linked to the Encina Toolkit so they can invoke Encina transaction management functions. The relationship between ENCINA/ET and the transactional application is illustrated in Figure 5.2. Note that ENCINA/ET is actually linked to modules in the Encina Toolkit, but from the application program point of view all communication is through the Encina TP monitor.

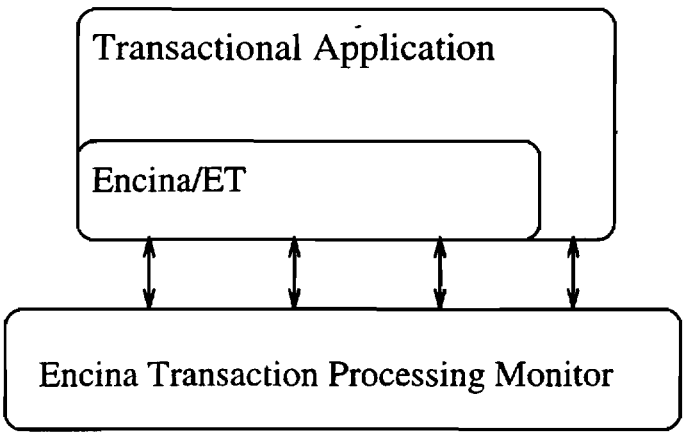

Figure 5.2: Relationship between applications, ENCINA/ET and Encina TP monitor.

Figure 5.2 above shows that the extended transaction library isolates the ENCINA/ET library from the application – programmers access extended services through available extended transaction control operations. The figure also shows that both ENCINA/ET and the application itself can access the resources of the underlying Encina TP monitor.

5.1.3 Design of the Metalevel Interface

An underlying principle of our metalevel interface was that it should be as small as possible. Specifically, the number of operations in the metalevel interface was kept to a minimum. Each argument of an operation expresses some real information that the extended transaction service needs from the programmer to perform its function. The programmer should never have to pass information if the extended transaction service can determine the value. For example, when an application invokes the `ignoreConflict` command, the only argument *required* is the name of the *cooperative* transaction — the identity of the extended transaction creating the ignore-conflict record is simply the one invoking the command.

The "minimalist" principle outlined in the preceding paragraph advances our goal of ease of use and simplicity. Unfortunately, it conflicts with our *internal* use of metalevel operations. In our implementation of ENCINA/ET we make use of operations from the metalevel interface to implement extended transactions services. In these cases it is not always possible to determine the default value(s) correctly. For example, when the LOCK ADAPTER is performing a lock delegation, it must establish an ignore-conflict relationship between the two transactions involved in the delegation. There is no way to determine the callee, so to guarantee that the values are set correctly, the LOCK ADAPTER must provide the identity of both extended transactions to the `ignoreConflict` command.

Our approach in ENCINA/ET is to provide two versions of metalevel operations for which we would like default argument values. The simple version always uses the default(s). The extended version has the same name as the simple version followed by the characters "_2", and it allows the programmer to specify the argument's value in question. For example, the extended version of the `ignoreConflict` command, called `ignoreConflict_2`, requires the callee to specify both transactions in the ignore-conflict relationship. This approach increases the number of constructs in the library, but reduces the number of arguments in frequently used operations. The net result is that system programmers using the metalevel interface to implement new extended transactions generally have fewer arguments to worry about and their code is much neater. Programmers using the metalevel interface to implement extended transaction services, such as delegation, semantic transaction synchronization, and transaction execution control, have the necessary power to do so.

5.2 Implementation of ENCINA/ET

This section presents the implementation of ENCINA/ET, beginning with a description of the key data structures. As our implementation discussion proceeds, we shall identify how these structures are used to implement specific extended transaction functions. Following, in subsections 5.2.2 – 5.2.4, we describe the implementation of key extended transaction services. This presentation parallels our framework design discussions presented in Section 3.3. Throughout these discussions we identify the Encina API commands, callbacks, and functionality that we build on in our implementation. In this sense, we stress the boundaries between ENCINA/ET and the Encina TP monitor, identifying the features that are important for our implementation.

5.2.1 Extended Transaction Data Structures

The implementation of the internal representations of extended transaction structures is very important to overall system performance and resource consumption. In this section we describe the main data structures in ENCINA/ET, illustrated in Figure 5.3.

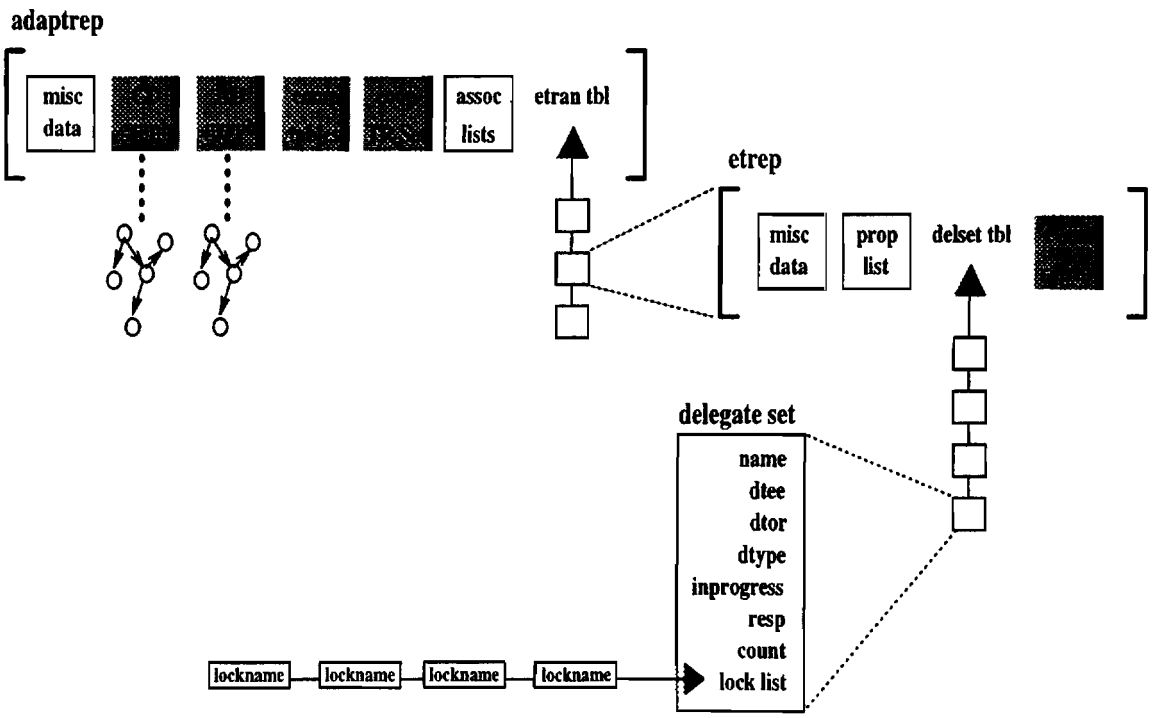

Figure 5.3: Main data structures in the internal extended transaction representation. Each rectangular box corresponds to a major data item and the shaded areas represent data structures that are further explained in subsequent discussions.

Main Data Structures

To centralize data management, all information for the transaction adapters in ENCINA/ET is stored in a structure called `adaptrep`, while all important information for an extended transaction is stored in a structure called `etrep`.

- **Extended Transaction Dependency Set**
  The extended transaction dependency set, `etranDepSet`, records the dependency graphs used to support transaction execution control. Entries are created in the `etranDepSet` structure using the metalevel command `define_dependency`. Our current implementation of ENCINA/ET maintains two dependency graphs; one for commit dependencies (CD) and another for abort dependencies (AD) between extended transactions. Internally, `etranDepSet` is represented by an array in which each entry holds a distinct dependency graph. The internal representation of the individual dependency graphs is described further in Section 5.2.4 and illustrated in Figures 5.7 and 5.8.

- **Semantic Compatibility Tables**
  The semantic compatibility table set, `CompTblSet`, stores semantic compatibility tables loaded by an application. It is a key data structure in the implementation of semantic transaction synchronization. The internal representation of the tables stored in `CompTblSet` is described in Section 5.2.3 and illustrated in Figure 5.5.

- **Cooperative Transaction Set**
  The cooperative transaction set, `CoopTrSet`, stores the active ignore conflict records created between extended transactions. It is a key data structure in the implementation of semantic transaction synchronization. The internal representation of the ignore conflict records recorded in `CoopTrSet` is described further in Section 5.2.3 and illustrated in Figure 5.6.

- **Miscellaneous Data**
  There are miscellaneous data items mainly used by internal ENCINA/ET operations. One example is the wait-for graph constructed for detecting transaction deadlock.

- **Extended Transaction Table**
  The most important `adaptrep` component is the table of extended transaction descriptors, `etran_tbl`. Each `etran_tbl` entry holds the internal representation of an extended transaction (`etrep`). In ENCINA/ET, `etran_tbl` is represented as an array of structures, each entry of which contains a pointer to an extended transaction descriptor `etrep`. Each `etrep` structure contains the following fields:

- **etrid:** a unique extended transaction identifier, represented by an integer that also serves as an index into `etran_tbl`. There are three support functions: `new_etrid()` generates an etrid value by locating an available entry in `etran_tbl`; `create_etrep(etrid)` allocates and initializes an `etrep` structure; and `delete_etrep(etrid)` frees the space allocated to an entry no longer in use.
- **tid:** storage location for the underlying Encina transaction identifier of type `tran_tid_t`. The value is set using the Encina TRAN module call `get_tid()` when the application issues the metalevel command `instantiate(name)`.
- **name:** a string variable that records the name assigned to the extended transaction by the application. Application programs use this name in performing extended transaction control operations, such as forming transaction dependencies, establishing ignore-conflict relationships, etc. The support function `getetrid_using_name(name)` searches the `etrep` entries in `etran_tbl` and returns the `etrid` (location) of the extended transaction matching the name, or indicates that it was not found.
- **state:** an enumerated type consisting of the values {initiated, active, pending, committed, aborted} that records the transaction state.
- **type:** an optional transaction type, internally represented as a character string. The support functions `set_type(name, type)` and `get_type(name)` set and get this value, respectively.
- **internal_state:** an optional application specific transaction state, internally represented as a character string. The support functions `set_state(name, state)` and `get_state(name)` set and get this value, respectively.
- **eventlist:** a collection of transaction management events associated with the extended transactions; essentially, this defines the interface applications can use to access extended transaction functionality. An eventlist is represented as a linked list of *event descriptors.* Each event descriptor contains a string that identifies the name of the event, a linked list of guards (predicates) that are represented as strings, a pointer to the function that serves as the *handler* for the event, an enumerated type variable that characterizes the event execution properties, and, finally, a boolean value that indicates whether the event is triggerable.
- **sbcc_enabled:** a boolean variable that indicates whether the application intends to use semantic transaction synchronization for the extended transaction.

After `sbcc_enabled` is set to TRUE, the CONFLICT ADAPTER is called when a lock conflict is detected for the transaction.

- **delegate_enabled:** a boolean variable that indicates whether the extended transaction can delegate locks to other extended transactions.
- **acquire_enabled:** a boolean variable that when set to TRUE indicates that the extended transaction can acquire locks on delegated data objects.
- **dependency_enabled:** a boolean variable that indicates whether the extended transaction can form and participate in transaction dependencies. After `dependency_enabled` is set to TRUE, the TRANSACTION MANAGEMENT ADAPTER scheduler is called when the transaction attempts to execute *transaction significant events.*
- **sbcc_policy:** sources to be checked by the CONFLICT ADAPTER in an attempt to relax a R/W lock conflict for this extended transaction. Each source identifies a compatibility table class name, or the keyword "ignoreconflict" that indicates that records in the `CoopTrSet` are to be used. The internal representation of `sbcc_policy` is a linked list of strings.
- **delegate set:** stores information pertaining to the active delegate sets created by the extended transaction. It is the main data structure in the implementation of transaction restructuring. Its internal representation is presented in Section 5.2.2 and illustrated in Figure 5.4.
- **proplist:** a list for associating property data with an extended transaction, as illustrated in the implementation of ESR presented in Section 4.2. Property data is a list of *(key, value)* pairs. Property values are assigned and retrieved using `set_etranprop(etrid, key, value)` and `get_etranprop(etrid, key)`, respectively. Internally, `proplist` is represented as a linked list of structures that contain a key and value field, both of which are string variables.

5.2.2 Implementing Transaction Restructuring

Initializing an Extended Transaction To use the services of ENCINA/ET, an application must first *register* a transaction using `instantiate(name)`. The TRANSACTION MANAGEMENT ADAPTER locates an open position in the extended transaction table `etran_tbl` using `new_etrid()` and creates an extended transaction descriptor `etrep` using `create_etrid(etrid)`. The newly created `etrep` structure is initialized, the necessary transaction callbacks are registered, and the state of the extended transaction is set to *initialized.* Specifically, the TRANSACTION MANAGEMENT ADAPTER issues the following operations:

```
create and initialize extended transaction descriptor...
etrid = new_etrid()                                                    (1)
TID = getTID()                                                         (2)
etrep_ptr = create_etrep(name,TID)                                     (3)
status = insert_etrep(etrid,etrep_ptr)                                 (4)
status = tran_CallBeforeAbort(TID,et_event)                            (5)
if (status == TRAN_SUCCESS) then continue else return(status)          (6)
status = tran_CallBeforeCommit(TID,et_event)                           (7)
if (status == TRAN_SUCCESS) then continue else return(status)          (8)
status = tran_CallAfterFinished(TID,et_event)                          (9)
if (status == TRAN_SUCCESS) then continue else return(status)          (10)
set_state(etrid,initialized)                                           (11)
return(success)                                                        (12)
```

In line 1 `new_etrid()` is used to locate an available entry in the extended transaction table `etran_tbl`. In line 2 the current transaction's TID is obtained using the Encina `getTid` function. Then, in line 3 `create_etrep()` allocates space for the extended transaction descriptor, storing the name of the extended transaction and the TID in the newly created extended transaction descriptor. The extended transaction descriptor is then inserted into the table `etran_tbl` using `insert_etrep`.

Next, the Encina callbacks are registered for the extended transaction. Line 5 registers the TRANSACTION MANAGEMENT ADAPTER event handling function *et_event* as the callback function to be executed before transaction$_{TID}$ is aborted. Similarly, line 7 registers *et_event* as the callback function to be executed before transaction$_{TID}$ commits, and line 9 registers *et_event* as the function that is to be executed after transaction$_{TID}$ has completed (i.e., Encina commit or abort processing is complete). Lines 6, 8 and 10 perform error checking using the Encina defined constant TRAN_SUCCESS. Finally, the extended transaction state is set to *initialized* in line 11 using `set_state` and in line 12 the function `instantiate` returns.

The `tran_CallAfterFinished` callback might appear redundant from the extended transaction processing point of view. However, there are callbacks in the Encina Recovery Service that developers may wish to utilize at a later date. The `tran_CallAfterFinished` event serves as notice that transaction execution is truly complete, and at that point the extended transaction descriptor can be removed from the extended transaction table `etran_tbl`. Together, these three callbacks effectively enable the TRANSACTION MANAGEMENT ADAPTER to track the execution of a transaction from the time an application issues the `instantiate` command until the time the transaction terminates and its extended transaction descriptor is deleted.

Transaction Restructuring As described in Section 3.3.1, the LOCK ADAPTER provides extended transactions with the ability to restructure dynamically, by delegating ownership of some or all of the acquired locks on data objects. To implement transaction restructuring, the LOCK ADAPTER utilizes the services of the Encina LOCK service module. Specifically, ENCINA/ET includes the file **lock/lock.h**, which contains LOCK data type and function interface declarations, and is linked to the library **libEncServer.a** which contains LOCK service functions. The primary data structure in the implementation of transaction restructuring is the *delegate set*, which is illustrated in Figure 5.4.

```
TYPE
  delset_type: STRUCT;
    name: char*;
    dtee: etrid;
    dtor: etrid;
    dtype: enumerated type, one of ''immediate'' or ''deferred'';
    inprogress: boolean;
    resp: enumerated type, one of ''dtee'' or ''dtor'';
    count: integer;
    locklist: list of lock_name_t; (* Encina Lock Manager data type *)
  end; (* delset_type *)
```

Figure 5.4: Basic data structure for a delegate set.

The implementation of operations that **create** and **delete** a delegate set, along with operations to **insert** and **remove** the names of data objects from a delegate set is straightforward. Of interest, however, is the implementation of the **delegate** operation and how it interacts with the Lock Manager in the Encina Toolkit. In the paragraphs below we detail our implementation of **delegate** and its supporting guards.

To perform a **delegate** operation, of the form $\texttt{delegate}_{t1}(t_2,$ **delset, deltype**), the LOCK ADAPTER must first determine if the operation is well-formed. The first step is to evaluate ((**State**(t_1, **Active**) == TRUE) AND (**State**(t_2, **Active**) == TRUE)), which tests that both extended transactions are running — otherwise, a call to the Encina Lock Manager would result in an application runtime error. The next step is to evaluate ((Delegate_Enabled(t_1) == TRUE) AND (Acquire_Enabled(t_2) == TRUE)), which verifies that delegation has been appropriately enabled for both extended transactions. Next, we evaluate ((**deltype** == **immediate**) OR (**deltype** == **deferred**)), to confirm the type of delegation has been correctly set. These tests are implemented using simple functions that either look-up information in the extended transaction table or test local call variables.

The final step to determine if **delegate** is well-formed is to verify that t_1 holds a lock on each data object in **delset**. Otherwise, a call to the Encina Lock Manager to release a lock not held by the transaction would result in an application runtime error. This test is facilitated by the Encina Lock Manager command `lock_GetTranInfo(TID`$_{t1}$`)`, which returns a list of the locks held by a transaction, and for each lock its mode (lockmode), the space in which the lock resides (lockspace), and its duration (duration). The LOCK ADAPTER does not interpret these values, but uses them later in Lock Manager calls that carry out the actual lock transfer. It simply uses the list of its locks to verify that each lockname in the delegate set is present in the list.

To avoid introducing transaction deadlock, the LOCK ADAPTER first verifies the lock delegation will not introduce a deadlock. Deadlock detection is implemented by the function **cycleFree**, which returns TRUE if no cycles are detected and FALSE if a cycle (deadlock) is detected. **cycleFree** effectively constructs the wait-for graph that will result after the delegation, "marking" transactions as *visited* by recording their identifier (TID) in the list *visited*. It uses the Lock Manager functions `lock_GetTranInfo(TID`$_{t1}$`)` and `lock_GetLockInfo(lockmode, lockname, lockspace)`. The latter returns the list of transactions waiting for a lock on a specific data object. The **cycleFree** function is outlined below:

`cycleFree(t`$_i$`:tran_tid_t, delset:delset_type)`

- Check whether transaction t_i is waiting for any locks using `lockTran_waitfor(TID`$_{ti}$`)`. Return TRUE if t_i is not waiting for any locks.
- If *firstpass* then insert the name of each data object in the delegate set (*delset*) into the list *holdlock* and set *firstpass* to FALSE, else gather the list of locks t_i holds using `lock_getTranInfo(TID`$_{ti}$`)` and insert them into *holdlock*.
- If *holdlock* is empty then return TRUE, else for each lockname in *holdlock* and each transaction t_j waiting for a lock on lockname do:
  - If the waiting transaction's identifier (TID_{tj}) is in the list *visited* then a cycle has been detected, so return FALSE.
  - Recursively call `cycleFree(t`$_j$`, null_list)` for the waiting transaction entry.
  - If the recursive call returns FALSE then propagate the result by returning FALSE, else add the transaction identifier (TID_{tj}) to the list *visited*.
- Return TRUE. *All waiting transactions have been checked, no cycles were found.*

Once the LOCK MANAGER has determined the **delegate** operation is well-formed and that no transaction deadlocks will result, it can proceed with the actual transfer. For each lockname identified in locklist of the delegate set, the following operations are performed:

1. Prepare for a lock conflict. The lock transfer will require two extended transactions to lock the data object concurrently, potentially resulting in a lock conflict. Thus, we first create an ignore-conflict record by issuing the CONFLICT ADAPTER COMMAND: `ignoreConflict_2(`t_1`, `t_2`, lockname, null, null, null, lockname)`, where the name of the lock being transferred is used as the handle for the ignore-conflict record.

2. Transfer lock ownership. First, ownership of the lock is granted to extended transaction t_2, by issuing the Encina Lock Manager command `lock_Acquire(`TID_{t2}`, lockmode, lockname, lockspace, duration)`. A R/W conflict will be detected by the Lock Manager, but relaxed by the Conflict Adapter using the ignore-conflict record created in step 1. Then, the lock is released from t_1 by issuing the Encina Lock Manager command `lock_Release(`TID_{t1}`, lockmode, lockname, lockspace)`.

3. Record for undo. Insert the lockname in a temporary data structure called the *undolist*. If an error is encountered during subsequent lock transfers, this transfer can be rolled-back using *undolist*.

4. Clean up. Remove the ignore-conflict record using the CONFLICT ADAPTER COMMAND: `removeIC_2(`t_1`,lockname)`.

5. Update the dependency graphs. Adjust the CDREL and ADREL graphs to reflect the delegation of the lock on *lockname*: Any (t_i, t_k) edge tagged with *lockname* becomes a (t_j, t_k) edge tagged with *lockname*. Similarly, any (t_k, t_i) edge tagged with *lockname* becomes a (t_k, t_j) edge tagged with *lockname*.

5.2.3 Implementing Semantic Transaction Synchronization

As described in Section 3.3.2, the CONFLICT ADAPTER provides a transaction synchronization service that allows an application to define and select semantic compatibility definitions for individual extended transactions. To implement semantic transaction synchronization, we utilize the services of the Encina LOCK service module. Specifically, the CONFLICT ADAPTER module of ENCINA/ET includes the file **lock/lock.h** during compilation, which contains the LOCK data type and function interface declarations and is linked to the library **libEncServer.a**, which contains LOCK service functions.

Semantic transaction synchronization is implemented, in part, using the Encina Lock Manager *conflict callback* facility. The conflict callback facility allows an application to specify a function to call when a R/W lock conflict occurs. When a lock conflict is detected, the Lock Manager invokes the registered function, passing it arguments pertaining to the lock conflict. If the function returns (votes) TRUE, the Lock Manager will ignore the conflict and grant the lock request; otherwise the Lock Manager will let the conflict stand. Thus the conflict callback facility enables the CONFLICT ADAPTER to participate in resolving R/W conflicts.

The function *relaxConflict* implements semantic transaction synchronization in ENCINA/ET. It is not, however, automatically registered as the conflict callback function for an extended transaction. Instead, `relaxConflict` is registered only when an application calls $\texttt{setp}_{t1}$(sbcc_enabled, TRUE). In response the CONFLICT ADAPTER registers the `relaxConflict` for t_1 using the command `lock_RegisterConflictCallback`, as illustrated below.

sbcc_enabled has been set to true...
`status = lock_RegisterConflictCallback(`$\texttt{TID}_{t1}$`, TRUE, relaxConflict)` (1)
`if (status ==` LOCK_SUCCESS`) then` *continue* `else return(status)` (2)

Line 1 registers the CONFLICT ADAPTER function *relaxConflict* as the function to call when the Lock Manager detects R/W CONFLICTS involving t_1. The argument $\texttt{TID}_{t1}$ is the transaction identifier of t_1, and the argument TRUE indicates that the registered function, *relaxConflict*, will *vote* on the decision to ignore the conflict. Line 2 performs error checking using the Encina defined constant LOCK_SUCCESS. When *relaxConflict* is invoked, it will applies available semantic information and returns either TRUE or FALSE to the Lock Manager, indicating whether to ignore the conflict or not.

As stated in our design of semantic transaction synchronization, presented in Section 3.3.2, the Lock Manager passes the following information to the CONFLICT ADAPTER when a conflict is detected: $hold_{TID}$ – identifier of the transaction holding the lock, $hold_{op}$ – operation currently active, $hold_{mode}$ – mode of the lock being held, *lockName* – logical name of the lock, $request_{TID}$ – identifier of the transaction requesting the lock, $request_{op}$ – operation pending, and $request_{mode}$ – mode of the lock being requested. The Encina Lock Manager does not store operation names in the lock table, nor does it pass operation names in lock requests. Thus, it cannot include operation name in the conflict callback to *relaxConflict*. As a result, the *relaxConflict* can only apply semantic information pertaining to lock modes and transaction identifiers to determine if the conflict can be relaxed.

The implementation of *relaxConflict* is built around two main data structures: a collection of *semantic compatibility tables* and a set of ignore-conflict records stored in a *cooperative transaction set*. As described in Section 3.3.2, a semantic compatibility table specifies for a specific data object *objname* whether an operation op_i can be executed while operation op_j is uncommitted. The value of each (op_i, op_j) entry is of the form: `[Action, Dependency]`, where `Action` is one of: SOK – the operations are semantically compatible and the conflict can be relaxed, NOK – the operations conflict, or *event* – a named event (predicate) that is evaluated to determine semantic compatibility, and where `Dependency` is a transaction dependency that is to be recorded between the two corresponding transactions if the conflict is relaxed.

In ENCINA/ET, compatibility tables are stored in a single table, referred to in Figure 5.3 as `CompTblSet`. Each entry in `CompTblSet` stores a unique compatibility table, as illustrated in Figure 5.5. Semantic compatibility tables are loaded and deleted from the `CompTblSet` using `loadTbl(pathname:string, name:string, class:string)` and `removeTbl(name:string)`, respectively.

```
TYPE
  comptbl_type: STRUCT;
    name: char*; (* name of this table *)
    class: char*; (* semantic class *)
    lockname: char*; (* keyword ALL means this table applies for all data objects *)
    entry: pointer to entry_type; (* linked list of table entries *)
  end; (* comptble_type *)

  entry_type: STRUCT;
    hold: lock_mode_t; (* mode lock is being held *)
    request: lock_mode_t; (* mode lock is being requested *)
    action: enumerated type, one of SOK, NOK or ESR;
    depname: enumerated type, one of ND, AD or CD;
    next: pointer to entry_type;
  end; (* entry_type *)
```

Figure 5.5: Basic data structures for semantic compatibility table.

The second structure used to store semantic information is the *cooperating transaction set*, referred to in Figure 5.3 as `CoopTrSet`. `CoopTrSet` is implemented as an array of ignore conflict records, illustrated in Figure 5.6.

```
TYPE
  icrecord_type: STRUCT;
    creator: etrid;
    cooptran: etrid;
    lockname: char*;
    event: char*;
    depname: enumerated type, one of ND, AD or CD;
    handle: char*;
  end; (* icrecord_type *)
```

Figure 5.6: Data Structure for an ignore-conflict record in the cooperative transaction set.

The field *creator* is the identifier of the transaction that created the ignore-conflict record, *cooptran* is the identifier of the transaction it will allow conflicting lock requests, *lockname*(optional) specifyies the data object CoopTran can access, *event* (optional) specifyies the predicate to evaluate to determine compatibility, *depname* (optional) specifies a dependency to record if the conflict is relaxed and, finally, *handle* (optional) specifies a unique name for the ignore-conflict record.

On receiving a conflict event, *relaxConflict* will use these two data structures to determine if the R/W conflict can be relaxed and the lock on the data object can be granted to the requesting transaction. In accordance with the *Semantic Conflict Rule*, described in Section 3.3.2, the conflict can be relaxed (i.e., is *semantically compatible*) if either a compatibility table indicates the operation for which the lock is being requested is semantically compatible with the uncommitted operation holding the lock, or the transaction holding the lock has explicitly indicated that the transaction requesting the lock has permission to perform the operation. The function *relaxConflict* returns TRUE as soon as it finds one source that relaxes the conflict or returns FALSE if no source relaxes the conflict. A high-level description of the *relaxConflict* function is outlined below.

```
BEGIN relaxConflict
  IN tidhold: identifier of transaction holding lock;
  IN modehold: mode lock is being held;
  IN lockname: logical lock name;
  IN tidreq: identifier of transaction requesting lock;
  IN modereq: mode lock is being requested;

   etridreq = getetrid_from_tid(tidreq);
   etridhold = getetrid_from_tid(tidhold);
```

namereq = getname_from_etrid(etridreq);
namehold = getname_from_etrid(etridhold);

Get the list of policy_names from the *sbcc_policy* field of the requesting transaction descriptor. For each policy_name listed do:

- If (policy_name == ignoreconflict) then
  - Search the ignore_conflict records in **CoopTrSet** for a match, using etridhold, etridreq, and lockname. If a match is found then:
    1. Check the event field of the ignore conflict record to see if an event (predicate) name is specified. If no event is specified go on to the next step, otherwise evaluate(event). If the predicate returns TRUE then go to the next step, otherwise continue search;
    2. Check the dependency name field of the ignore conflict record to see if a transaction dependency needs to be recorded. If not, then **return**(TRUE);
    3. Attempt to form the dependency using the command **form_dependency_2(depname, namereq, namehold, lockname)**. If successful then **return**(TRUE), otherwise continue search;
- Else for each table in **CompTblSet** where ((policy_name == table.class) AND ((table.lockname == lockname) OR (table.lockname == ALL)) do
  - Search the entries in the compatibility table for a match, using modehold and modereq. If a match is found then:
    1. Check the event field of the compatibility table entry to see if an event (predicate) name is specified. If no event is specified then go on to the next step, otherwise evaluate(event). If the predicate returns TRUE then go to the next step, otherwise continue search;
    2. Check the dependency name field of the ignore conflict record to see if a transaction dependency needs to be recorded. If not, then **return**(TRUE);
    3. Attempt to form the dependency using the command **form_dependency_2(depname, namereq, namehold, lockname)**. If successful then **return**(TRUE), otherwise continue search;

return(FALSE). *All sources checked, unable to relax conflict.*
END (* *relaxConflict* *)

5.2.4 Implementing Transaction Execution Control

To implement transaction execution control, the TRANSACTION MANAGEMENT ADAPTER utilizes the transaction service calls and callback facility of Encina TRAN service module. During compilation, ENCINA/ET includes the file **tran/tran.h**, which contains TRAN data type and function interface declarations, and is linked to the library **libEncina.a** which contains TRAN service functions.

Transaction event scheduling is implemented by the function *schedule_et*. An application declares its intension to use event scheduling for an extended transaction by calling `setp`$_{t1}$`(enable_dependency,TRUE)`. Once transaction dependencies have been enabled, *schedule_et* is called each time an extended transaction raises an event. Specifically, when the `tran_CallBeforeAbort` or `tran_CallBeforeCommit` callback is raised, the function *tran_event* will first invoke *schedule_et* to determine if the event can be processed.

To demonstrate the ability of *schedule_et* to coordinate the execution of extended transactions, we consider two well-known transaction dependencies, commit dependencies and abort dependencies. The main data structures used to implement *schedule_et* are transaction *dependency graphs*.

Commit Dependency Graph The graph, CDREL, keeps track of the commit dependencies between extended transactions. Its vertices correspond to extended transactions. An edge exists from t_i to t_j if t_j is commit dependent on t_i, and this edge is tagged with the name of the data object that caused the dependency.

Abort Dependency Graph The graph, ADREL, keeps track of the abort dependencies between extended transactions. Similarly, its vertices correspond to extended transactions, and an edge exists from t_i to t_j if t_j is abort dependent on t_i. Each edge is tagged with the name of the data object that caused the abort-dependency relation to form.

In our current implementation, the dependency graphs used for transaction execution control are stored in the structure `etranDepSet`. The internal representation of `etranDepSet` is an array of dependency graphs, the structure of which is illustrated in Figure 5.7. Each entry in `etranDepSet` records the unique name of the dependency, the type of the dependency (either CAUSAL or ORDER) the transaction significant events (BEGIN, COMMIT or ABORT) and an array of structures that records the edges of the dependency graph, detailed in Figure 5.8.

A transaction dependency type is created using the TRANSACTION MANAGEMENT ADAPTER command `define_dependency(dependency_name, event_name`$_a$`, event_name`$_b$`, deptype)`. This command searches `etranDepSet` to verify that the dependency name is

```
TYPE
  dependency_type: STRUCT;
    depname: char*;
    deptype: enumerated type, one of CAUSAL, ORDER;
    preevent: enumerated type, one of BEGIN, COMMIT, ABORT;
    postevent: enumerated type, one of BEGIN, COMMIT, ABORT;
    dependency: array of dependency_entry_type; (* indexed by etrid *)
  end; (* dependency_type *)
```

Figure 5.7: Data structure for an extended transaction dependency graph.

unique, then creates a new entry and initializes the name, dependency type, and event fields. For example, a commit dependency graph is created with `define_dependency(CD, COMMIT, COMMIT, ORDER)`.

```
TYPE
  dependency_entry_type: STRUCT;
    disabled: boolean_t;
    count: integer;
      with: etrid_t;
      label: char*;
      next: pointer_t;
  end; (* dependency_entry_type *)
```

Figure 5.8: Data structure for recording individual dependencies.

Once the graph has been defined (created), an application can record and remove dependencies between extended transactions using `form_dependency`$_{t1}$`(CD,`t_2`,lockname)` and `delete_dependency`$_{t1}$`(CD,`t_2`,lockname)`, respectively.

Processing Commit Events Recall from our discussion in Section 3.3.3, when an extended transaction attempts to commit, the event can be rejected and delayed. Commit, like begin and prepare, is a *normal event*. Since the dependency type of the CDREL graph is ORDERING (dependency), *schedule_et* delays the commit of an extended transaction t_i to enforce the dependency rules.

- If there is an edge (t_j, t_i) in CDREL, then t_i is commit dependent on the uncommitted transaction t_j and cannot be committed. Delay, by calling `set_state(pending)` to put the commit request in event pending list, and retry later when t_j terminates. Otherwise, execute the remaining steps below.

  1. Remove all edges in CDREL and ADREL involving t_i. For each successor, t_j, of t_i in CDREL that is in the *pending* state, if (t_i, t_j) was the only edge entering t_j in CDREL, perform commit(t_j).
  2. Finally, call `set_state(committed)` to set the state of t_i to *committed.*

Processing Abort Events When an extended transaction aborts the event cannot be delayed or ignored — abort is an *immediate event* (see Section 3.3.3). Since the dependency type of the ADREL graph is CAUSAL (dependency), the only option is to accept the event and trigger the abort of other extended transactions to enforce the dependency rules. To abort a transaction, we use the Encina TRAN function `abortNamedTran(`TID_{tj}`, ENFORCE_ABORT_DEPENDENCY)`, where TID_{tj} is the TID of the transaction to be aborted and `ENFORCE_ABORT_DEPENDENCY` is a string constant that describes the reason for aborting the transaction.

1. For each transaction t_j such that (t_i, t_j) in ADREL, abort t_j using `abortNamedTran(`TID_{tj}`, ENFORCE_ABORT_DEPENDENCY)`. *Remove the corresponding edge* in ADREL and decrement the dependency counter. Continue this process until all transactions reachable from t_i in ADREL have been aborted.

2. For each successor, t_j, of t_i in CDREL that is pending, perform commit(t_i) – *Recall, a commit dependency simply orders the occurrence of commit events, but this dependency has been resolved by the abort of t_i and can be removed.*
   Remove all edges in ADREL and CDREL involving t_i.

3. Finally, call `set_state(aborted)` to set the state of extended transaction t_i to *aborted.*

5.3 ENCINA/ET Evaluation Overview

In this section we evaluate the Reflective Transaction Framework design and the implementation of ENCINA/ET. Recall our specific goals from Section 3.1: new extended transaction functionality; ease of use; ease of implementation; and, acceptable overall performance. Chapter 3 presented the motivation and detailed design of three new extended transaction services to implement advanced transaction models and semantics-based concurrency

control protocols. Chapter 4 demonstrated the ease with which the framework could be used to implement advanced transaction models and semantics-based concurrency control protocols. Section 5.2 of this chapter described how the extended transaction services defined in Chapter 3 can be implemented as extensions of the base transaction services of a commercial TP monitor. In this section we focus on the final goal: showing that the performance and resource cost for supporting the extended transaction services defined by the Reflective Transaction Framework are indeed acceptable.

Our evaluation approach consists of an analysis of ENCINA/ET source and framework design, along with controlled experiments. We consider two distinct perspectives — a *software engineering perspective* and a *systems perspective*. From a software engineering perspective, we first ask, in Section 5.3.1, whether ENCINA/ET's code size and complexity are commensurate with its functionality. Next, in Section 5.3.2, we evaluate the performance of extended transaction services and resource costs of ENCINA/ET based on quantitative experimental results. Finally, in Section 5.4, we evaluate the usability of the framework and compare its flexibility with that of related extended transaction systems, and ask how easy it is to use the framework to construct new extended transactions; however, because of its subjective nature, only a preliminary assessment of the usability of the framework is presented.

5.3.1 System Size and Functionality

In Section 5.2 we presented the implementation of ENCINA/ET. This realization of the Reflective Transaction Framework on a conventional TP monitor allows us to demonstrate the practicality and viability of our design. In this section, we present code size data from the implementation to explore whether ENCINA/ET's size and complexity are commensurate with its functionality.

The ENCINA/ET source code lives in three modules, corresponding to the three transaction adapters that make up the framework – the TRANSACTION MANAGEMENT ADAPTER, LOCK ADAPTER, and CONFLICT ADAPTER. The TRANSACTION MANAGEMENT ADAPTER module contains approximately 700 lines of C code, which enables an application to define and manage dependencies between extended transactions for explicit execution control, and reifies transaction-specific information in an extended transaction descriptor. The LOCK ADAPTER module contains approximately 400 lines of C code, which allows an application to control the locks held by an extended transaction explicitly and to restructure an extended transaction dynamically by delegating data objects. And, finally, the CONFLICT ADAPTER module contains roughly 450 lines of C code, which enables an application to define semantic notions of conflict and select semantic synchronization for individual extended transactions. In addition, there are auxiliary files that contain code,

such as macro functions and header files, that define key data structures. In total, the ENCINA/ET source is on the order of 2000 lines of C code.

Table 5.1: Breakdown of lines of code (loc) in ENCINA/ET software modules.

ENCINA/ET *Module*	*Total loc*	*Callback Handling*	*Encina API*
Transaction Management Adapter	700	80	50
Lock Adapter	400	0	60
Conflict Adapter	450	20	25

A breakdown of the ENCINA/ET implementation in terms of lines of code (*loc*) is presented in Table 5.1. This presentation of code size includes the total *loc* to implement each transaction adapter module and a breakdown of each module that identifies the *loc* required to process Encina transaction event callbacks, and *loc* required to perform API calls to the Encina Toolkit. In contrast, the Encina Toolkit on which ENCINA/ET is built has a code size of over 100,000 lines of C. Of this the transaction service module (TRAN) has approximately 14,000 lines of C code, and the lock service module (LOCK) has approximately 4,000 lines of C code.

The extended transaction services of ENCINA/ET use the services of TRAN and LOCK. The TRAN module provides services to manage the definition, execution, and termination of transactions. This includes the creation and management of the transaction table, transaction initialization and termination, a thread-to-TID mapping service, remote procedure call management, and an application interface. The LOCK module provides a logical locking facility to manage the lock space. It records locks held by a transaction, transactions holding a lock, and transactions waiting for a lock on a data object. It also provides efficient functions to acquire and release locks, detect transaction deadlocks, and detect R/W conflicts. These base transaction services serve as the cornerstones for our implementation of the extended transaction services in ENCINA/ET.

By reusing the transaction services of the Encina Toolkit, we were able to implement ENCINA/ET in approximately one man-year. The value of the framework for ease of implementation, then, is the way it allows us to stand on the work of others so as to provide implementation support for extended transactions. At the same time, this architectural layering does not preclude access to the underlying TP monitor, so applications can continue to use base transaction services. As a result, the amount of code to be implemented, debugged and tuned for implementing the extended services defined by the Reflective Transaction Framework is significantly reduced, as demonstrated by our ENCINA/ET implementation. We believe these benefits will carry over when porting ENCINA/ET to another TP monitor, but some details may differ, e.g. due to the lack of a lock manager in the target TP monitor. We leave this conjecture open for future validation.

We present this information on the source code size of ENCINA/ET to advance our claim that the Reflective Transaction Framework can be efficiently implemented as a *thin* software layer over the transaction processing services of a conventional TP monitor, and does not require a system of excessive size or complexity. Thus, the approach can be seen as a judicious blending of existing transaction system functionality and careful addition of extended transaction functionality, to yield a system which provides support for implementing extended transactions. Novelty is thus more manifest in the methodology adopted than in the individual components which have been implemented.

5.3.2 Performance Overhead for Library Operations

This section presents a series of experiments that measure the performance of the extended transaction services provided by the ENCINA/ET library. This performance data isolates the cost of various functions in the system and enables us to not only identify basic system functions that are computationally expensive, but also to determine *where* future efforts should be concentrated to improve the performance of the implementation. These measurements also serve to define the bounds of system performance and provide users with a basis for understanding larger operations, such as implementing new extended control operations for an advanced transaction model that would make use of these services.

Methodology

Our goal was to measure the average cost for each extended transaction service. Performance numbers presented in the following experiments were obtained by measuring operations over repeated trials. For each experiment we collected measurements and observed the results, and when the results converged the experiment was terminated. Outliers, resulting from transient system events, such as system interruptions, network activity, aborted transactions, etc., were discarded.

Each experiment involves executing a test that exercises a specific set of extended transaction functions. For testing purposes we have used a modified version of the TPC-B transaction processing benchmark [Ser91]. The TPC-B benchmark models a teller at a bank. There is one bank with one or more branches, and multiple tellers and multiple accounts per branch. The database represents the cash position of each entity (branch, teller, and account) and a history of recent transactions run by the bank. Each transaction is a deposit or withdrawal on an account by a teller in a branch. The transaction profile is presented below, where Aid (Account_ID), Tid (Teller_ID), and Bid (Branch_ID) are keys to the relevant records/rows.

```
/* Given Aid, Bid, Delta by caller */
BEGIN TRANSACTION
  Update Account where Account_ID = Aid:
     Read Account_Balance from Account
     Set Account_Balance = Account_Balance + Delta
     Write Account_Balance to Account
  Write to History:
     Aid, Tid, Bid, Delta, Time_stamp
  Update Teller where Teller_ID = Tid:
     Set Teller_Balance = Teller_Balance + Delta
     Write Teller_Balance to Teller
  Update Branch where Branch_ID = Bid:
     Set Branch_Balance + Delta
     Write Branch_Balance to Branch
COMMIT TRANSACTION
Return Account_Balance to driver
```

In our test program, the benchmark driver selects an account (Aid) and branch (Bid), generates a random amount (Delta) to withdraw from or deposit to the account, then calls the teller transaction. The teller first obtains a lock on the account and then updates the balance, followed by updates to the branch, teller and account balances, and finally appends a history record to the audit trail. This simple debit/credit transaction clearly does not require extended transaction support. However, using this benchmark we can compare the performance of conventional ACID transactions against transactions using extended services, and verify our extensions are functioning correctly.

Our implementation of the benchmark differs from the TPC-B specification in three aspects. First, the specification requires that the database keep redundant logs on different devices. We only used a single log. Second, we ran all tests on a single, centralized system, so there were no remote accesses. Third, we added input parameters to the driver program that allow us to specify the bank account (Aid), branch (Bid), teller (Tid) and transaction amount (Delta) directly, as well as to pause transactions during execution for running more controlled tests. We also ran different experiments than specified in TPC-B to measure specific extended transaction functions, since our goal is to evaluate the extended transaction services of ENCINA/ET, and not to measure TP monitor performance.

Our performance metric is *elapsed time*, abbreviated *Elapsed*. Elapsed time is needed to determine if applications will meet response requirements and to estimate the duration that

locks will be held while operations are taking place. When processing a lock conflict callback, for example, the Encina TP monitor must hold latches on the lock and transaction table entries until the registered callback function returns. Elapsed time measurements were made using the Encina (BDE) (Base Development Environment) `bde_GetTime` function call, which uses the `gettimeofday()` system call. The `gettimeofday()` call returns a timestamp expressed in elapsed seconds and microseconds since `00:00` GMT, January 1, 1970 (zero hour). Calls to `bde_GetTime` are made before and after the function being measured, the elapsed time is then accumulated over a number of trials and averaged to provide the numbers reported in the following tables.

The elapsed time metrics that are reported were measured using the Encina TP monitor version 1.0.1 and SunOS version 4.1.3_U1. The hardware was a Sun SPARCstation 10 Model 41 with a 40 MHz processor. The Sun workstation had 64 megabytes of main memory, 278 megabytes of swap space, a one-gigabyte internal disk drive, and two external Seagate Elite-2 two-gigabyte SCSI disks. The Encina TP monitor was configured to use a local (raw) logging partition on one of the external SCSI disks, with the Encina structured file server (SFS) running on a separate external SCSI disk acting as the data store for the test application. Both the Encina TP monitor and our testing application reside on the internal SCSI disk. The numbers reported in the following tables are accurate to two significant digits. In all tests, performance measurements were conducted with the Sun workstation under light load with no contention on any resources Encina consumes (i.e., no other disk activity and, unless specifically mentioned, no other transactional applications being executed).

Performance Overhead for Managing an Extended Transaction Descriptor

Prior to using any extended transaction service, an application must first *register* a transaction with ENCINA/ET, which in turn creates an extended transaction descriptor and registers the necessary Encina callbacks. This adds a certain amount of overhead. The question is, How much? More specifically, we want to know: *What is the performance overhead for creating an extended transaction descriptor, registering the callbacks with Encina to track the execution of the underlying transaction, and removing the extended transaction descriptor once the transaction has finished?* Our first experiment measures the costs to create an extended transaction descriptor for a teller transaction, to register the necessary callbacks, and to remove the extended transaction descriptor. Table 5.2 presents the performance measurements from this test.

Table 5.2: Execution times for managing an extended transaction descriptor.

Measurement	ENCINA/ET *Library Primitive*	*Average Elapsed*
1A	Execute teller transaction (ACID)	113.63 milliseconds
1B	Begin teller transaction (ACID)	2.93 milliseconds
1C	Commit teller transaction (ACID)	1.08 milliseconds
1D	Execute *extended* teller transaction	115.32 milliseconds
1E	Create and initialize **etrep** structure	420 microseconds
1F	Register CallBeforeAbort callback	84 microseconds
1G	Register CallBeforeCommit callback	85 microseconds
1H	Register CallAfterFinished callback	84 microseconds
1I	Remove **etrep** structure upon completion	309 microseconds
Total	Overhead for managing extended transaction descriptor	989 microseconds

To collect the measurements presented in Table 5.2, we used our TPC-B test program, modified to create an extended transaction descriptor for each teller transaction prior to executing normal account updates. To create an extended transaction descriptor, a name is required for an extended transaction. This name is generated for each teller transaction by converting the randomly selected teller identifier to a string and storing it in the variable `tellername`. Each teller transaction then creates an extended transaction descriptor using the command `instantiate(tellername)` and performs the account update and logging operations. When the teller transaction terminates, the extended transaction descriptor is removed. Once the performance runs were complete, balances for account, teller and branch were examined, along with the sum of deltas for the history file, to verify that all the values were changed in accordance with the deltas of the teller transactions.

As a baseline for our evaluation, we first measured the performance of a default (ACID) teller transaction that did not create an extended transaction descriptor. The timing for this default teller transaction is presented in Table 5.2 as measurement 1A. In addition, we measured the performance overhead of Encina operations `begin_transaction` and `commit_transaction` (measurements 1B and 1C, respectively). These timings are high, relative to published TPC-B results, so a few comments regarding our benchmark implementation and system configuration are in order.

The remote procedure call, or RPC, and the transactional remote procedure call, or TRPC, are among the more expensive mechanisms used by Encina; a TRPC consists of an RPC with additional data used to track the transaction state. Disk I/O can also be quite expensive. Our current test configuration uses the Encina structured file server (SFS) as the bank data store. Most SFS operations require an RPC between the program requesting the operation and the SFS, even in both reside on the same machine. For our TPC-B test program, a teller transaction has the following operations:

- add delta to account record
- add delta to branch record
- add delta to teller record
- add record to history file

Using `sfs_ReadByKey` and `sfs_UpdateByKey` calls, each transaction requires a total of six RPCs for the SFS calls that modify the account, branch, and teller files. Most commercial database systems, and more recent implementations of SFS, offer a batch update call that can replace these six RPCs with a single RPC. To further diagnose this performance problem, we examine system idle time. The Unix command `iostat` showed a significant amount of disk operations (idle time was consistently near zero), while the Unix command `vmstat` showed non-zero CPU idle time. Together, these indicate that disk storage is a bottleneck; `vmstat` also showed a high number of paging events, indicating additional memory would be beneficial. Better disk throughput could be obtained by allocating storage to SFS across multiple physical disks, each with its own SCSI controller. Since our goal is to measure the costs of the extended transaction services, not to optimize TPC-B throughput, we proceed with our current benchmark implementation and system configuration.

Next, we ran our modified TPC-B test program to measure the performance of an *extended* teller transaction, which creates an extended transaction descriptor and registers callbacks to report its execution state. The extended teller transaction's total execution time is presented in Table 5.2 as measurement 1D. To identify the sources of the performance overhead, we instrumented the `instantiate` operation in the ENCINA/ET library to collect timings for the individual operations that manage an extended transaction descriptor. In Table 5.2 we see that creating an extended transaction descriptor, presented as measurement 1E, is much slower than other operations being measured. The overhead comes from allocating memory to store the extended transaction descriptor `etrep`, initializing data fields, and storing the descriptor in `etran_tbl`. The operations that register the TRANSACTION MANAGEMENT ADAPTER with the transaction callback facility for transaction abort, commit and finished events have roughly the same overhead (measurements 1F, 1G and 1H). An explanation for this is that the algorithm that implements callback registration is common to all events – it must latch the transaction table entry for the transaction, add the callback function and arguments to the list of callbacks maintained for that event, and then release the latch and return a status code. Similar performance overheads for other callbacks, measured in experiments presented later in this section, support this conjecture. Finally, we measured the overhead for releasing the storage held

by the extended transaction descriptor and to set the extended transaction table entry to NULL when the teller transaction is finished (measurement 1I). Thus, the overhead for all operations that manage an extended transaction descriptor totals 989 microseconds on average, less than one millisecond per extended transaction.

A review of the measurements in Table 5.2 reveals that approximately one-half of the total cost for managing an extended transaction descriptor comes from storage allocation and initialization of the `etrep` structure. Another one-third of the total cost comes from freeing `etrep` storage when an extended transaction terminates. An optimized implementation of ENCINA/ET could use *pooling*, in which a collection of `etrep` structures are preallocated and reused, to reduce these overheads.

Performance of Transaction Restructuring

Next, we present the costs of the ENCINA/ET operations that perform transaction restructuring. This experiment involves two teller transactions, where the first teller selects an account and performs a balance update, then *delegates* the account data object to the second teller for further update. We use a modified version of our TPC-B test program, which initiates two concurrent teller transactions, then creates an extended transaction descriptor for each transaction. Each teller transaction is allowed to perform its individual account update, then writes a history record containing the account number, branch identifier, teller number, and amount of the update. At the end of the first update run, each teller transaction delegates its account data object to the other teller, then repeats the account update loop with the new account object. Once the performance runs were complete, balances for account, teller and branch were examined, along with the sum of the deltas in the history file to verify that all values were changed in accordance with the deltas of the teller transactions. Table 5.3 presents the performance measurements of the operations that perform transaction restructuring.

Table 5.3: Execution times for performing transaction restructuring.

Measurement	ENCINA/ET *Library Primitive*	*Average Elapsed*
2A	Create a named delegate set	516 microseconds
2B	Delegate a data object (one lock)	32.37 milliseconds
2C	Get the list of locks held (one lock)	9.41 milliseconds
2D	Get the list of locks held (10 locks)	66.39 milliseconds
2E	Lock a data object using `lock_Acquire` (no contention)	7.02 milliseconds
2F	Unlock a data object using `lock_Release`	11.83 milliseconds
3C	*Create an ignore-conflict record and store in* `CoopTrSet`	*2.70 milliseconds*
3D	*Process lock conflict callback (one ignore-conflict record)*	*7.27 milliseconds*

Timings were collected for the time required for a teller transaction to first create a delegate set using `create(accountlist, dtor)`, presented as measurement 2A. This cost is independent of the number of data objects that the extended transaction will eventually delegate – only one set is required to hold the lock names. Next we measured the overhead to perform the delegation of the account data object using the operation `delegate(tellerid, accountlist,` IMMEDIATE), presented as measurement 2B. Measurements for the `insert` and `remove` operations were not included, as they are implemented by simple C expressions. To better understand the cost of delegation, we measured the constituent primitives of the `delegate` operation. First, we measured the Encina operation `lock_GetTranInfo`, which returns the list of locks held by a transaction. This Encina function is used both to obtain the list of locks held by a transaction performing a global delegation, and to obtain the mode and lockspace of each lock being transferred. `lock_GetTranInfo` was first timed for a teller transaction holding the lock on one account (measurement 2C), and again for a teller transaction holding the locks on 10 accounts (measurement 2D). Next, a single teller transaction was started, which simply locks a random account data object to read the balance and then unlocks it using `lock_Acquire` and `lock_Release`, respectively. Costs for the lock and unlock operations are presented as measurements 2E and 2F. Recall from Section 3.3.1 that transaction restructuring requires support from the semantic transaction synchronization service to relax the lock conflict that results from the actual lock transfer. Thus, to complete the performance analysis, measurements for overheads required to create an ignore-conflict record and to relax a lock conflict using the ignore-conflict record were obtained, and presented as measurements 3C and 3D, respectively. These measurements are from our performance analysis of semantic transaction synchronization, presented in Table 5.4, discussed in the following section.

Performance of Semantic Transaction Synchronization

Next, we measured the performance costs for the ENCINA/ET operations that implement semantic transaction synchronization. This experiment involves two concurrent teller transactions that attempt to update the same account data object. The lock conflict that results from concurrent transactions attempting to access the same account is relaxed using semantic transaction synchronization, and the operation costs are measured. Table 5.4 presents performance measurements of the operations that perform semantic transaction synchronization.

The driver for the original TPC-B test program selects an account at random from the 10000 bank accounts in the Encina SFS (structured file server) database for each teller transaction. Given that the maximum number of tellers is 10, conflicts between teller transactions on an account data object are rare. To force a lock conflict to occur

Table 5.4: Execution times for performing semantic transaction synchronization.

Measurement	ENCINA/ET *Library Primitive*	*Average Elapsed*
3A	Register `relaxConflict` as conflict callback function	83 microseconds
3B	Perform lock conflict callback call (no processing)	39.44 milliseconds
3C	Relax R/W conflict with ignore-conflict record	47.82 milliseconds
3D	Create ignore-conflict record and store in `CoopTrSet`	2.70 milliseconds
3E	Process lock conflict callback (single IC record)	7.27 milliseconds
3F	Search `CoopTrSet` (10 IC records)	19.08 milliseconds
3G	Process lock conflict callback (single SC table)	9.54 milliseconds
3H	Search `CompTblSet` (10 semantic compatibility tables)	18.86 milliseconds

on each account update, we modified our TPC-B test program. In the new version, the driver module executes two concurrent teller transactions with fixed teller numbers (teller 1 and teller 2), and fixed the account numbers so that both tellers attempt to access the same account. Each transaction creates an extended transaction descriptor and selects an update amount (delta) at random. In the first run, teller 1 creates an ignore-conflict record specifying that teller 2 can access the account. Teller 2 is then delayed for one second using the BDE command `bde_ThreadSleep(delay_tv)`, to ensure teller 1 completes its update operation. Teller 1 performs its account update and logging operations, then blocks until teller 2 completes processing. Upon waking up, teller 2 can perform the account update. The conflict that results from attempting to update the account held by teller 1 is relaxed by the *relaxConflict* function and the processing time was measured. Once the performance runs were complete, balances for account, teller and branch were examined, along with the sum of the deltas from the history file, to verify that all the values were changed in accordance with the deltas of the two teller transactions.

As a baseline for measuring the performance of *relaxConflict*, we first measured the cost for Encina to perform a lock conflict call. That is, the elapsed time from the point the Lock Manager first detects a lock conflict to the point that the registered callback function returns a *vote* on the conflict. This effectively measures the amount of time it takes Encina to construct a conflict event, place a latch on the transaction table entry for the conflicting transaction, and then call the registered callback function. For this baseline measurement we were only interested in the Encina overhead, not the performance of our *relaxConflict* function. Thus, we registered a constant function that simply returned FALSE, thereby consuming minimal clock cycles; later in our evaluation, we shall register *relaxConflict* in place of this constant function. The time required for the Lock Manager to register the conflict callback function is presented in Table 5.4 as measurement 3A. To carry out this evaluation, we then modify the Encina Lock Manager source to capture timing information. Specifically, calls to `bde_GetTime` are placed in the Encina source

file lockConflict.c at the point that a lock conflict is detected and at the point that the registered callback function returns. The result of this evaluation is presented in Table 5.4 as measurement 3B.

Once this baseline evaluation was complete we returned to using the original Encina library and the modified TPC-B test program. Since both teller transactions attempt to access the same account, each trial results in a lock conflict. The function *relaxConflict* is invoked in response to this conflict event, and the conflict relaxed using the ignore-conflict record. We measured the time required to relax this R/W conflict using the available ignore-conflict record, presented in Table 5.4 as measurement 3C; note, this measurment includes the 39.44 milliseconds required by Encina to perform a lock conflict call to *relaxConflict*. To better understand the overhead involved in relaxing lock conflicts, we instrumented the support functions for semantic transaction synchronization. We first measured the time required to create and store an ignore-conflict record in the `CoopTrSet` (measurement 3D). Next, we measured the time *relaxConflict* actually required to destructure the conflict event and search `CoopTrSet` to relax the conflict (measurement 3E). In our initial test, `CoopTrSet` held only one ignore-conflict record, yet in actual applications we would expect there to be several ignore-conflict records – especially for cooperative applications consisting of a number of active transactions. Thus, we placed 10 ignore-conflict records in the `CoopTrSet` and measured the time to search through the records for a match (measurement 3F). Finally, we measured the time required for the function *relaxConflict* to search through the semantic compatibility table, first containing only one table (measurement 3G) and again containing 10 compatibility tables (measurement 3H). Regardless of the semantic synchronization algorithm being used by the transactional application, for example altruistic locking, cooperative serializability, commutativity, recoverability, etc., these microbenchmarks measure the basic mechanisms that would be used in their implementation.

Performance of Transaction Execution Control

In our final performance evaluation, we measure the performance costs for ENCINA/ET operations that perform transaction execution control. In this experiment we establish transaction dependencies between multiple concurrently executing teller transactions, then measure the operation costs for execution control. Specifically, we measure the overhead to define (create) a new transaction dependency type, to form a dependency between extended transactions, and to enforce transaction commit and abort dependencies. Table 5.5 presents the performance measurements from this experiment.

Table 5.5: Execution times for performing transaction execution control.

Measurement	ENCINA/ET *Library Primitive*	*Average Elapsed*
1F	*Register CallBeforeAbort callback*	*84 microseconds*
1G	*Register CallBeforeCommit callback*	*85 microseconds*
4A	Create dependency graph structure in `etranDepSet`	1.22 milliseconds
4B	Form a transaction dependency	870 microseconds
4C	Evaluate transaction commit dependency	3.09 milliseconds
4D	Evaluate transaction abort dependency	2.61 milliseconds

We first measured the overhead for creating a dependency type. This was accomplished by first instrumenting calls to the operation `define_dependency`, then issuing commands to create a commit dependency (CD), an abort dependency (AD), and a begin dependency (BD). Essentially this test measures the time required for the TRANSACTION MANAGEMENT ADAPTER to allocate and intialize a dependency graph structure for each dependency type. The result of this test is presented in Table 5.5 as measurement 4A. Much of this overhead is memory allocation costs. If the dependency types are known in advance, preallocation and caching would reduce this cost.

To measure the cost for forming and enforcing transaction dependencies, we prepared a modified version of our test program. The modified test program uses three teller transactions, with fixed teller numbers (1 through 3). Each teller transaction has a name, tellername, whose value is the corresponsing teller number converted to a string. Each teller transaction creates an extended transaction descriptor using the command `instantiate(tellername)`, and then forms the following commit and abort dependencies with other teller transactions:

- `form_dependency`$_{teller1}$`(CD, teller2, NOLABEL);`
- `form_dependency`$_{teller2}$`(CD, teller3, NOLABEL);`
- `form_dependency`$_{teller3}$`(CD, teller1, NOLABEL);`

- `form_dependency`$_{teller1}$`(AD, teller2, NOLABEL);`
- `form_dependency`$_{teller2}$`(AD, teller3, NOLABEL);`
- `form_dependency`$_{teller3}$`(AD, teller1, NOLABEL);`

Essentially, each teller transaction will commit only if all three teller transactions in the group commit, and all will abort if any one transaction in the group aborts. This modified test program was then used to collect measurements for the formation of transaction dependencies (measurement 4B) and to verify the function *schedule_et* works correctly for

both commit and abort dependencies. The cost for this operation is relatively inexpensive, which is expected since it simply creates a dependency record and records it in the appropriate structure (dependency type).

During the first series of runs, each teller transaction selects an account, branch and delta at random, and then performs the account update operation and log updates. As the transactions complete, the function *schedule_et* enforces the commit dependencies by delaying their commit until they all raise a commit event (i.e., the `tran_CallBeforeCommit` callback is raised). We measure the overhead for the framework to detect and evaluate the commit dependency for each extended transaction (measurement 4C). Included in this measurement, and the following abort dependency measurement (4D), is the time required for Encina to process the commit (abort) callback and invoke *schedule_et*. Unfortunately, we do not have access to source for the library **libEncina.a**, which contains TRAN service functions, and, thus, cannot instrument commit (abort) callback processing. Based on measurements for processing lock conflict callbacks (measurement 3B), we know these costs can be quite high. A series of tests on *schedule_et*, performed after this experiment was complete, showed that searching the dependency graphs (both commit and abort) is performed in less than 800 microseconds; an optimized graph implementation based on hashing could further reduce this cost.

A second series of test runs was performed in the same fashion, except that each transaction executes a conditional statement that randomly aborts the transaction. Again, we measure the time required for the framework to detect and evaluate the abort dependency (measurement 4D). In this case, the function *schedule_et* enforces the abort dependency by issuing the Encina command `abortNamedTran` to abort the active or pending transactions. The cost for enforcing an abort dependency is less than a commit dependency, as less time is spent evaluating an abort dependency – *schedule_et* simply aborts any dependent transactions.

Once these performance runs were complete, balances for account, teller and branch were examined, along with the sum of deltas for the history file, to verify that all the values were changed in accordance with the deltas of the committed teller transactions.

5.4 Reflective Transaction Framework Evaluation

The Reflective Transaction Framework was designed for flexibility, to implement a wide range of extended transactions readily. In Chapter 4 we demonstrated the use of the framework to implement selected advanced transaction models and semantics-based concurrency control protocols. So, having seen these extended transaction examples separately, it is worth stepping back to discuss the ways in which the framework meets this challenge.

First, we briefly compare the extended transaction implementations in terms of their different requirements and transaction control operations. Next, we discuss how the flexibility in the Reflective Transaction Framework that these extended transactions exploit compares to the facilities in other extended transaction implementations discussed earlier. Finally, we discuss how computational reflection and Open Implementation techniques make this possible.

5.4.1 Comparing the Extended Transaction Implementations

The advanced transaction models and semantics-based concurrency control protocols presented in Chapter 4 differ considerably in their intended domains. More importantly, they also differ considerably in their structures and styles. Consider the various differences:

- Split transactions use transaction restructuring to release partial results selectively and continue executing; joint transactions use transaction restructuring to transfer all database resources held and then terminate.
- Chain transactions, a special case of joint transactions, restrict the execution structure to a linear chain of extended transactions; joint transactions have no restriction on their execution structure.
- Reporting transactions use transaction restructuring to report results to another extended transaction periodically, without terminating execution; joint transactions terminate execution after performing transaction restructuring.
- Cooperative Transaction Groups utilize semantic transaction synchronization to facilitate cooperation between the individual extended transactions in a cooperative group.
- Commutativity can relax R/W conflicts based on operation semantics, without forming a transaction dependency; Recoverability also relaxes R/W conflicts based on operation semantics, but places a commit ordering restriction on the transactions.
- Epsilon Serializability relaxes R/W conflicts using application semantics, to explicitly allow a bounded amount of inconsistency in transaction processing; Commutativity and Recoverability both restrict extended transaction execution to consistent (serializable) schedules.

It is certainly not the case that the approach adopted by one advanced transaction model is right, and that adopted by the other is wrong. Nor is it the case that one model subsumes the other, or even that a particular transaction control operation in

an advanced transaction model is more correct or more general. Rather, an advanced transaction model reflects the transaction processing requirements of a particular advanced application domain, and design decisions embodied in the individual control operations can only be resolved in the context of a particular application or scenario. Consequently, each advanced transaction model has been optimized for a particular behavior desirable for only a particular advanced application domain. What's more, new advanced database applications will not simply require different sets of options for these various decisions, but will likely introduce entirely new extended transaction control operations, as well as opening up new areas for extended transaction services. In other words, supporting these extended transactions means supporting the different extended behaviors which they might use, mapping the infrastructure supplied by the Reflective Transaction Framework onto the needs of the application, rather than the other way around.

5.4.2 Comparing the Reflective Transaction Framework

In Chapter 2, related systems for implementing extended transaction were presented, with particular focus on the range of extended transaction behaviors they could support. Having now seen the core elements of the Reflective Transaction Framework design and examples that demonstrate the extended transaction services it offers, it seems appropriate to return to those systems and contrast the flexibility in the Reflective Transaction Framework with that offered in the other systems. Could they be used to implement the extended transactions presented in Chapter 4, and if not, why not?

There are two sets of reasons why this would be difficult or impossible. One set is fairly simple; the second is more significant.

Application Interface Flexibility

The first set of reasons arises from the inability of some of the systems to provide applications with the ability to specify the extended transaction services they require. For instance, APRICOTS and TSME do not support interface variability, so an application cannot select model-specific definitions for a transaction control operation such as `commit` and `abort`; APRICOTS operates in terms of predefined *contracts* which do not include arbitrary transaction control operations, while TSME forces an application to select a specific extended transaction model that will be used for all transactions. While both APRICOTS and TSME support highly structured transaction models, such as the chained transaction model or sagas, neither supports the dynamic transaction interactions found in the split-join or cooperative group models. This is a more significant issue for APRICOTS, since it is intended to support end-user variability without further programming.

Since TSME is organized as a toolkit for use within other application programs, it may be possible to build support for transaction restructuring, although no such applications have been described in the TSME literature. Simply put, APRICOTS and TSME provide their extended transaction support on an *"all-or-nothing"* basis.

Similarly, PERN does not allow an application to restructure a transaction, as is required to implement the Split-Join model. Moreover, neither PERN nor APRICOTS support the free-for-all access illustrated in the cooperative group model. While PERN provides flexible concurrency control, its control is in terms of rules based on predefined conditions and facilities, not in terms of application-specific needs. As a result, PERN, APRICOTS, and TSME cannot implement advanced transaction models such as the cooperative transaction group, in which arbitrary transactions can join a group and freely access selected objects. This level of control is simply outside of their design requirements.

System Architecture Flexibility

The second set of reasons, however, is more relevant to the basic design of these related systems, and to the use of the Open Implementation approach in the Reflective Transaction Framework.

Some of the systems described in Chapter 2 have no support for the forms of architectural variability seen in the range of extended transaction examples presented in Chapter 4. The cooperative group model requires execution control between member transactions, can utilize transaction restructuring to delegate locks from member transactions to the group transaction upon commit, with automatic relaxation of conflicts between member transactions. The chained transaction model utilizes execution control to sequence individual transactions and does not perform delegations, but can selectively relax conflict between individual transactions. However, among the systems, only TSME, APRICOTS and ASSET support execution control. PERN emphasizes concurrency control. While TSME and APRICOTS provide opportunities for semantic transaction synchronization, these do not extend to the more dynamic interactions illustrated by cooperative transaction groups and the altruistic locking protocol; thus, extended transactions cannot create delegate sets or transfer database resources. PERN, similarly, assumes highly structured extended transactions, while execution control and dynamic restructuring are simply not issues in its design.

Critically, where mechanisms exist for defining extended transaction functionality in the related systems presented in Chapter 2, their use of traditional abstraction techniques requires that the programmer *"drop down"* to the implementation level to gain control. For instance, ASSET's separation of mechanism and policy means the functionality of

the extended transaction services must be implemented within the transactional application, requiring application programmers to deal with a new level of abstraction. These two levels are inextricably mixed in ASSET. APRICOTS' *contract* approach constrains this slightly by dealing in terms of a specific contract for managing, say, execution control between individual extended transactions (contracts), but still requires a complete specification of extended transaction services; there is no provision for the incremental definition of new mechanisms or the optional reuse of existing facilities, since a contract must be *completely* defined for a transactional application in advance. To extend the concurrency control services of PERN to support application-specific concurrency control requirements, a transaction systems programmer would have to write a series of rules that re-implemented its concurrency control mechanisms. In other words, while the Open Implementation approach is designed to *allow* programmers to *become involved* in aspects of the infrastructure which support their applications, these other approaches *require* programmers to *take responsibility* for them.

5.4.3 OI and Reflection in the Reflective Transaction Framework

The value of the Reflective Transaction Framework lies in the provision of a framework within which new extended transaction behaviors and structures can be defined. Each of the extended transaction implementations presented in Chapter 4 has taken elements from the Reflective Transaction Framework and tailored them to its specific needs: to redefine the notion of conflict to implement semantic concurrency control or facilitate transaction cooperation; to control the execution of individual transactions for structuring cooperative groups or to chain transaction computations together; to utilize dynamic transaction restructuring to pass partial results between transactions; or to relax atomicity for open-ended activities. These specializations were performed simply and concisely, and fit naturally into the general structure for developing extended transactions which the framework sets up and implements. Furthermore, the code that implements the various control operations employed by these extended transaction examples is similarly straightforward. The implementation of the `split` and `join` operations, for example, required less than 50 lines of code; and the addition of application-specific concurrency control was on the order of a dozen lines of code or simply required the definition of compatibility tables.

The use of Open Implementation techniques, and the metalevel interface in particular, is critical to the way in which this flexibility is achieved.

First, it provides the structures for programmers to gain control over selected aspects of transaction processing. This means not only the opportunity to create new extended

transaction behaviors and transaction control operations that are usable within the framework, but also modifications that are seamlessly integrated into the framework's internal mechanisms (such as changes to the definition of conflict, which then take immediate effect on a per-transaction basis).

Second, it provides the means to do this more extensively than a parameterized approach. That is, extensions are made not only through the structural aspects of the extended transaction encoding, but also through the use of the metalevel interface, rather than simply *"switches."* The difference between the metalevel interface approach and pure parameterization is best seen in comparisons with *TSME*.

Third, the available metalevel interface retains the use of high-level specifications that *"dropping down"* to the implementation level would preclude. The components that metalevel commands address are just those that a transactional application uses, such as delegate sets, transaction dependencies, conflict relations, compatibility tables, etc. Transaction system programmers implement extended transactions in terms of application requirements on these metaobjects, while other, implementation-specific details which lie underneath remain hidden. The same metalevel interface commands can be maintained across various implementations of the Reflective Transaction Framework, since the metalevel interface is written in terms of the *revealed structure* of the underlying TP monitor, rather than the details of its implementation. This, in turn, encourages transaction system programmers to develop extended transaction implementations in terms of the specific requirements of the application, rather than the specifics of the framework implementation. So, for instance, the use of semantics-based concurrency control represents the expression of application-specific requirements, rather than the re-implementation of concurrency control in the framework (as would be required by, say, ASSET or *PERN*).

Each of these elements – application-specific control over aspects of the underlying transaction system's behavior, through programmatic access to a revealed model of its inherent structure – derives directly from computational reflection and the metalevel interface as elements of the Open Implementation design approach.

5.5 Discussion

In this chapter we presented the implementation of ENCINA/ET, which extends the Encina TP monitor to support the implementation of extended transactions. In addition, we presented an evaluation of ENCINA/ET and Reflective Transaction Framework. Our experience in designing the framework and implementing and evaluating ENCINA/ET has taught us a number of important lessons. Here we review the experience gained and lessons learned from the implementation and evaluation effort.

The basis of the Reflective Transaction Framework is to define extended transaction behaviors as careful extensions of existing transaction services. Instead of reimplementing base transaction services, our approach is to redefine and leverage available functionality in a conventional transaction processing facility to the extent possible. Implementing extensions to an existing transaction processing system is a significant departure from previous attempts, which implement extended transactions from scratch. It allowed us to ignore implementation aspects not specific to extended transaction functionality, and to focus on extended transaction implementation issues. The implementation of ENCINA/ET was carried out by the *incremental addition* of new extended transaction services, implemented as separate software modules called transaction adapters. On top of this structure we introduced the notion of *separation of interfaces*, providing a *metalevel interface* for transaction system programmers to define extended transaction control operations and an *extended transaction interface* for application programmers to develop transactional applications.

Our design of the Reflective Transaction Framework and implementation of ENCINA/ET poses the question, *How simple can a facility for implementing extended transactions be, while still supporting classic* ACID *transactions?* Our answer, as presented in this chapter, is an application-level library with minimal programming constraints, implemented in 2000 lines of mainline C code, and no more intrusive than a typical transaction library, such as Encina's TRAN-C. Transactional application programmers simply use calls from the extended transaction interface, such as `Split` and `Join`, to employ extended transaction functionality in their advanced applications.

Our implementation of ENCINA/ET demonstrates that new requirements for transaction processing do not necessarily imply a need for radically new transaction processing technology. ENCINA/ET also demonstrates that existing TP monitor functional components are applicable to extended transaction processing; however, their functionality has to be repackaged. The implementation of ENCINA/ET did not require the invention of any radically new approaches, merely the judicious selection, adaptation, and extension of the most suitable techniques.

The implementation of ENCINA/ET was facilitated by the transaction event callback mechanism and open API to the transaction services of the Encina Toolkit. A valid question is whether the additional work of exposing an API to the underlying transaction services and adding a transaction event callback mechanism to other transaction processing systems would be worthwhile. In our opinion, the answer to this is in part economic. There are only a handful of commercially significant TP monitors in circulation, which offer conventional ACID transaction support. This compares to thousands of transactional applications written on top of them, and possibly thousands more that could be developed

using extended transactions. It is our opinion that any additional work invested in TP monitor systems software to enable extensions, such as those introduced by Reflective Transaction Framework, to widen their application reach and make advanced application development easier should yield a large payoff.

Although our implementation was carried out in the context of the Encina transaction processing monitor, its results are not limited to Encina. The components of the Encina Toolkit are fairly representative of the core transaction facilities found in modern TP monitors. Thus, we are confident the approach taken and the lessons learned can be applied to other transaction processing systems. In particular, since the Encina toolkit has been used to implement IBM's CICS/6000, DEC's ACMS/xp, and Transarc's Encina TP monitors, so it is likely ENCINA/ET will run on all of these systems. Confirmation of this conjecture, however, awaits future portability experiments.

There are clearly performance costs to be paid for applications to use extended transaction services defined by the Reflective Transaction Framework. In our evaluation of ENCINA/ET we explored whether the Reflective Transaction Framework could be implemented efficiently on top of a conventional TP monitor, and what the performance overhead was for each extended transaction service. We presented a set of controlled experiments that cover the range of extended transaction services defined by the framework and that are implemented in ENCINA/ET. The observed performance overhead for the extended transaction services was modest across all the experiments. The measured operations were also in agreement with our relative evaluation to ACID transactions. In summary, the performance evaluation results presented in this chapter confirm our belief that the overhead imposed by the framework services is not unduly expensive. Since the current implementation has not been fully tuned for performance, more careful tuning could lead to further reduction in the performance overhead.

Chapter 6

Summary and Conclusion

In this chapter, we briefly summarize our work, identify our contributions, and outline opportunities for future research.

6.1 Recapitulation

We began in Chapter 1 by describing the problem, that is, the lack of practical extended transaction implementations and the inability of existing transaction processing systems to directly support the range of behaviors required to implement extended transactions. As a result, the vast majority of advanced transaction models and semantics-based concurrency control protocols have remained, at least thus far, mere theoretical constructs with no practical implementations. This problem has two aspects: one design and one implementation. The design aspect is the lack of extended transaction functional building blocks and accompanying application programming interfaces required to implement extended transactions. The implementation aspect is that traditional approaches to the design of transaction processing systems have required developers to make implementation decisions that subsequently restrict how those transaction processing systems can be used, and hence the range and form of the transactional applications that can be built using them.

These two aspects are related. In Chapter 2, we drew on recent work on Open Implementation to analyze these problems in terms of the use of abstraction, in both systems and applications. This analysis suggests a particular form of solution — the use of Open Implementation techniques to construct a framework that *"opens up"* transaction processing system functionality, resulting in a system in which the components and mechanisms that the framework offers can be manipulated, controlled and specialized by application developers to match the needs of particular applications and usage situations.

The main body of the dissertation (Chapters 3 – 5) presented our solution. In Chapter 3 we first described the basic form and design principles behind the Reflective Transaction

Framework, an extended transaction facility designed to be built on top of a conventional TP monitor. We then presented three novel extended transaction services that the framework provides for realizing extended transaction behaviors. The design of each service is focused on extending the underlying TP monitor and mapping framework structures onto transactional application needs, rather than the other way around.

The first extended transaction service is *dynamic transaction restructuring*, which allows an application to manage the database resources that it holds explicitly. This allows an application to help determine when an extended transaction will obtain and release database resources. This extended service is designed to support the ways in which database resources, that is specific data objects, are processed in a structured and collaborative manner. Specifically, transaction restructuring provides support for applications to selectively make tentative and partial results, as well as hints such as coordination information, accessible to other extended transactions, and to decouple the fate of updates from that of the transaction that performed the operations. As such, dynamic transaction restructuring offers direct support for implementing extended transactions used in collaborative and structured transactional applications.

The second extended transaction service is *semantic transaction synchronization*, which allows an application to define and select semantic compatibility for individual extended transactions. Semantic compatibility not only provides direct support for collaborative activity between extended transactions (unlike, for example, simple read/write locking protocols used in a conventional Lock Manager), but is also a means for application programmers to express the semantics of application operations. Application semantics provide a richer basis for decisions about transaction concurrency than would be available if all transaction operations were simply mapped to the most general read/write-semantics model. As a result, transactional applications developed using the Reflective Transaction Framework have increased potential for concurrency and direct support for transaction cooperation as appropriate for the particular application (rather than allowable concurrency embedded within the TP monitor's Lock Manager design).

The third extended transaction service is *transaction execution control*, which allows an application to control the execution of complex activities reliably. This extended service is designed to allow an application to place constraints on the execution of individual extended transactions. These constraints are expressed in terms of *dependencies* between the significant events of the extended transactions in an application. Applications can define new dependencies appropriate for the advanced transaction model they are using, and then form dependencies between extended transactions at runtime to determine execution order. As such, transaction execution control offers direct support for structuring an application as a sequence of activities, in which each activity is executed by an extended

transaction, and for controlling the interactions of extended transactions operating over a set of shared data objects.

Access to the extended transaction behaviors provided by the Reflective Transaction Framework and the exposed functionality of the underlying TP monitor is carefully organized through a well-documented *metalevel interface*. Transactional applications can use commands from the metalevel interface to "become involved" in tailoring the extended transaction infrastructure that supports them. The extended transaction services that the Reflective Transaction Framework provides (namely, transaction restructuring, semantic transaction synchronization, and transaction execution control) are designed not just to support specific extended transactions, but also to provide a basis for extension and specialization of the Reflective Transaction Framework's internal mechanisms.

Critically, the Reflective Transaction Framework does not simply provide a parameterized implementation in which users simply select from a set of extended transactions. Rather, it provides a framework within which new extended transaction behaviors and mechanisms can be crafted through the programmatic extension and specialization of revealed aspects of the TP monitor's internals. The view that the Reflective Transaction Framework provides into aspects of the underlying transaction processing systems structure, and the opportunities that it offers for applications to tailor and specialize this structure according to their particular needs, are the essence of the Open Implementations approach, and also the means by which the Reflective Transaction Framework offers considerable flexibility and control in the implementation of extended transactions.

To supplement the smaller examples which Chapter 3 used to illustrate technical points, Chapter 4 presented two sets of examples – the implementation of selected advanced transaction models, and the implementation of selected semantics-based concurrency control protocols. Individually, these examples illustrate how the Reflective Transaction Framework can be used to define and implement a number of important extended transactions, and how the relationship between framework facilities and application programming is managed. More importantly, when taken together, these examples illustrate the flexibility which the Reflective Transaction Framework embodies. Indeed, to the best of our knowledge, no system has been reported that can implement such a wide range of advanced transaction models and semantics-based concurrency control protocols. Together, these examples demonstrate how a single framework can embody radically different transaction extensions, and how applications can revise and adapt the framework mechanisms to leverage underlying TP monitor facilities for their own needs.

Finally, Chapter 5 presented ENCINA/ET, an implementation of the Reflective Transaction Framework on the commercial TP monitor Encina, and an accompanying evaluation of the Encina implementation and framework design. This chapter shows how simple a

facility for implementing extended transactions can be, while still supporting classic ACID transactions: an application-level library with minimal programming constraints, implemented in 2000 lines of mainline C code and no more intrusive than a typical transaction library. Our implementation of ENCINA/ET did not require the invention of any radically new transaction processing approaches, merely the judicious selection and careful extension of existing TP monitor functional components. This demonstrates both the practicality of the Reflective Transaction Framework design, and that new requirements for transaction processing need not require radically new transaction processing technology.

Our evaluation of the ENCINA/ET implementation demonstrated that the extended transaction services do not impose significant overhead. In addition, we explained how other extended transaction implementations, introduced in Chapter 2, would either fail altogether to support these different extended transactions, or would require the programmer to "step down" into the code of the implementation (if this were available) and provide implementation-specific extensions and modifications. In contrast, the design of the Reflective Transaction Framework allows customization at a high-level through the available metalevel interface.

In summary, our research shows that it is possible to extend a conventional TP monitor in a practical and modular manner to implement extended transactions. In doing so, we have presented not only the design of the Reflective Transaction Framework and extended transaction services it offers, but have also identified mechanisms for integrating these new services with the functionality provided by a conventional TP monitor. To demonstrate the practicality of these ideas and mechanisms, we have also presented a concrete implementation on a commercial TP monitor.

6.2 Contributions

This research is the first to demonstrate convincingly a practical method of extending a conventional transaction processing facility with mechanisms to support extended transactions, one that can readily implement a wide range of advanced transaction models and semantics-based concurrency control protocols. In this dissertation, we have presented a demonstration of our thesis, via design, application, implementation and evaluation of a working system. The specific technical contributions of each of the three main chapters are enumerated below.

1. *Reflective Transaction Framework Design* In Chapter 3 we presented the design of a software framework, the Reflective Transaction Framework, that balances several

design goals: new extended transaction functionality, ease of implementation, compatibility with legacy transaction systems, ease of use, and modest performance and resource costs. We highlighted the key design issues involved in the definition of new extended transaction services, specifically dynamic transaction restructuring, semantic transaction synchronization and transaction execution control. We presented a design in which these extensions could be smoothly integrated into a conventional TP monitor, advancing our goal for ease of implementation while maintaining compatibility with legacy transaction processing. Moreover, the design supports incremental extension — if only certain advanced transaction models or semantics-based concurrency control protocols are required, only those extended transaction services need be provided; other extended transaction behaviors can be incrementally added to the framework over time.

In addition, the framework offers principled access to extended transaction services and underlying TP monitor structures and mechanisms for examination and manipulation. This access is principled in the sense that the framework does not expose the functionality of the entire TP monitor, but only selected aspects of it. In addition, the metalevel interface encapsulates state so the TP monitor need not expose the internal data structures and functions that are actually used.

2. *Demonstration* In Chapter 4 we presented the implementation of several advanced transaction models and semantics-based concurrency control protocols, to demonstrate the flexibility of the Reflective Transaction Framework. The selected examples vary across a number of dimensions, differing not only in their intended application domains but also in the nature and structure of their implementation. These implementation variations — dynamically restructuring transactions versus strict atomic execution, controlled cooperation between transactions versus strict isolation, and execution control through both structural and dynamic dependencies — cut across the barriers that traditional transaction processing implementations erect.

3. *Encina/ET Implementation and Evaluation* In Chapter 5 we presented ENCINA/ET, an implementation of the Reflective Transaction Framework on the commercial TP monitor Encina. The implementation is based on transaction adapters, software modules built on top of the Encina Toolkit functional components. Each adapter uses transaction significant events to reify extended transaction state, and uses existing application programming interface calls to reflect changes to the computational state of the TP monitor. The extensions implemented by each transaction adapter builds on the available functionality of the underlying functional component of the TP monitor, to the extent possible, and provides the programmer with a

clean metalevel interface through which he or she can customize and extend system functionality. This allows new extensions and model improvements to be quickly incorporated, and as a result, the implementation can remain up to date with application requirements.

Our presentation illustrates the implementation of the extended transaction services defined by the Reflective Transaction Framework as extensions of base transaction services provided by the Encina Toolkit. As set forth in our design objectives, the implementation did not require the invention of any radically new approaches, merely the judicious selection, adaptation and extension of available transaction services. We also presented empirical measurements based on controlled experiments that confirmed ENCINA/ET's modest performance and resource costs.

6.3 Future Work and Opportunities

The analysis, design and implementation of the Reflective Transaction Framework is a practical approach to implement extended transactions on conventional TP monitors. It is a research area of great practical interest and one in which concerns of openness and extensibility are paramount. Building on this, there are a number of topics for further investigation which can be classified roughly into four areas: enhancements of the ENCINA/ET implementation to make the extended transaction services more complete; further evaluation of both the Reflective Transaction Framework and ENCINA/ET implementation through the application of extended transactions to real-world problems; research to develop further the Reflective Transaction Framework itself; and, research on the design and implementation of systems software using ideas from Open Implementations.

1. ENCINA/ET *Implementation Extensions*
   For implementation expediency, a few minor features logically belonging to the current Reflective Transaction Framework design have not yet been fully supported. Implementation enhancements, such as support for deferred delegation, could be added to make the extended transaction services of ENCINA/ET more complete. Another is the addition of persistence for key ENCINA/ET data structures.

   Persistent Data Structures Because a transactional application using ENCINA/ET can crash for various reasons, such as fatal runtime errors and machine shutdown, the system needs to maintain critical information in persistent storage to resume normal operations after system restart. One way this could be accomplished is by using RVM [SMK+94], a lightweight transaction facility for maintaining persistent data structures. RVM exports the abstraction of *recoverable virtual memory* to its

host application (ENCINA/ET) which can map *regions* of RVM's recoverable segments onto portions of its virtual address space. Accesses to mapped data are performed using normal memory read and write operations. If such accesses are bracketed with RVM's begin and end-transaction statements, failure atomicity is automatically provided. RVM asynchronously flushes updates to recoverable memory to the backing disk and allows the application to control the frequency of such flushes. Almost all the important information included in the ENCINA/ET data structures, described previously in Section 5.2, could be stored in RVM. Because RVM space is a scarce resource, the efficient design of ENCINA/ET data structures minimizes the portion that must remain persistent. Data items, such as operation compatibility tables that could be reloaded from the disk, would not need to be kept in persistent data structures.

2. *Further Evaluation and Application*
   So far, we have evaluated the performance overhead, resource cost, and some usability issues of the Reflective Transaction Framework and ENCINA/ET implementation based on controlled experiments and selected extended transaction implementations. Many other system usability issues are still unaddressed, pending further accumulation of usage experience. Moreover, the previous quantitative measurement results could be further strengthened or adjusted with more usage data. Issues such as these cannot be addressed until there is substantial system usage from a user community on an actual application. Thus another area for further investigation is to identify an area or a *killer application* that requires extended transaction support. This would enable a comprehensive usability study of the Reflective Transaction Framework and further performance evaluations of the ENCINA/ET implementation.

   One of the major areas positioned to exploit extended transaction capabilities is workflow management [SSU96]. In recent years, workflow management has emerged as a powerful tool to improve productivity of organizations [HC93]. Adopting a *process-centric* approach, industry has been promoting *workflow management* as a technique for modeling, executing, and monitoring such applications. A procedural description of how and what is to be performed to achieve work is termed a *workflow*. The individual steps that compromise a workflow are termed *activities*. Activities may involve humans as well as programs. Aided by advances in client-server computing and distributed database techniques, early office-automation systems have evolved into workflow management systems [MNB+94].

There are several prototype and commercial workflow management systems available [GHS95b], and many have features that address the needs of real working environments that advanced transaction models fail to consider. However, current workflow management systems do not have adequate support to satisfy the modeling and correctness requirements of advanced database applications. The deficiencies include no clear transaction concept, lack of support to keep track of data dependencies among different workflows, lack of support for cooperative activities, and insufficient support for recovery. Since these issues have been investigated extensively in the area of advanced transaction management, it would be valuable to cross-fertilize the two areas to develop a model and an architecture that provides flexibility in defining tasks and specifying the correctness and consistency requirements of advanced database applications. In particular, there is a need to support coordinated and cooperative tasks and to handle heterogeneity and interoperability.

One recent work that addresses these requirements is the Transaction Activity Composition Model (TAM) introduced by Ling Liu and Calton Pu [LP98a]. TAM provides a family of transaction activity restructuring operations in a unified framework for declarative specification and dynamic restructuring of workflows [LP98b]. The TAM framework is designed using concepts from computational reflection and Open Implementation, inspired by the design of the Reflective Transaction Framework. Reflection is employed in TAM to provide a specification interface for flexible workflow customization and to provide both application-level and system-level customization. As a result, TAM allows activity designers to incrementally specify the behavioral composition of complex activities and a wide variety of activity interaction dependencies through a declarative metalevel interface.

3. *Further Development of the Reflective Transaction Framework.*
   Another area for future work is extensions to the Reflective Transaction Framework. For instance, relaxed consistency guarantees, support for transaction compensation and application-specific correctness criteria are areas which our initial design does not address. These areas were omitted from the initial design and implementation so as to concentrate on core elements for implementing extended transactions, but are candidates for the same sort of development as dynamic transaction restructuring, semantic transaction synchronization, and transaction execution control.

   Beyond these enhancements, two major extensions worth exploring immediately are crash recovery and support for distributed execution of extended transactions. Work is, in fact, already underway in each of these areas by other members of our research group. The rest of this discussion describes that work in slightly more detail.

Extended Transaction Recovery Shu-Wie Chen has introduced a Modular Architecture for Recovery Systems (MARS) to construct flexible and efficient recovery systems to support extended transactions [Che98]. The MARS architecture is based on the observation that any recovery algorithm that implements transaction-oriented recovery must perform three tasks: identify the transactions to be aborted and committed, identify the operations associated with each transaction, and recover individual transactions by removing the effects of aborted transactions and inserting the effects of committed transactions. These tasks correspond to the three MARS recovery modules: transaction state analysis, transaction operation analysis, and transaction recovery. In keeping with traditional recovery systems, these recovery modules are organized so that the two analysis modules generate a recovery plan that can be executed by the recovery module. In this manner, MARS maintains backward-compatibility with existing recovery systems on TP monitors.

Associated with each MARS recovery module is a set of efficient, crash-aware algorithms that have been further decomposed into recovery microprotocols. These recovery microprotocols can be combined in various ways to implement different recovery functionality. This work includes an examination of various extended transaction models to identify the different recovery properties. In particular, Chen's work has considered the effects of dynamic transaction restructuring on transaction operation analysis, as well as the effects of semantic transaction synchronization and transaction execution control on transaction states analysis.

Distributed Extended Transactions Tong Zhou has recently introduced the Open Coordination Protocol (OCP) to support *distributed* extended transactions [ZPL96]. OCP is a coordination facility for constructing optimized coordination protocols for distributed extended transactions [ZPL96]. The main idea behind OCP is the decomposition of existing coordination protocols (e.g., two-phase commit protocol and its variants) into fine-grain *microprotocols*, which are then composed and specialized with respect to particular situations for flexibility, reliability, and performance. By applying OCP, both existing coordination protocols and new protocols can be developed; for example, the presumed-abort (PA) variant of the two-phase commit protocol [ML83, MLO86], the open commit protocol [RP90], optimistic commit protocol [LKS91], or unilateral commit [HS91]. Existing optimizations or new optimizations can be incorporated into these protocols as well, much as read-only, last-agent, voting reliable, etc. [SBCM93, SBCM95]. A key component of OCP,

with respect to the Reflective Transaction Framework, is the development of new coordination protocols for a variety of distributed extended transaction management control primitives, such as `delegate`, `split`, and `joingroup`.

4. *Open Implementation*

   A number of areas open for further work focus on the development of Open Implementation techniques and, in particular, their application to the design and implementation of transaction processing and database systems.

   First, open implementation is at an early stage of development, and general techniques building on the experiences of developers are only slowly being developed (e.g. recent work on OIA/D [KDLM95]). Each new experience, and each application to a new domain, brings refinements and insights into the model. As described in Chapter 4, one interesting aspect of the Open Implementation approach in the Reflective Transaction Framework design is the way in which programmers extend and enrich, rather than configure, the underlying transaction services. The ways in which this happens, its consequences, and its applicability to new domains, all remain avenues for fruitful investigation in the development of the Open Implementation approach.

   Second, as the focus of Open Implementation approach has broadened from its original grounding in programming language semantics and applications, researchers from other areas have begun to adopt aspects of the approach and apply them to their own work. This has included a number of investigations in distributed systems and operating systems of the value of reflective and metalevel techniques (e.g. [CM93, EPT95, Yok92, Str93, SW95]). These investigations aim principally at dynamic control and configuration of distributed systems, along with augmentation of programming languages in support of distributed programming, so they typically focus at a lower level than the work presented here; their focus is infrastructure (that is, "*below*" the application). However, they point towards an opportunity to use reflective techniques to integrate system and application issues by using metalevel information to coordinate the needs of both, and as such, share some of the motivations which have driven this work.

6.4 Parting Shot

The above list of possible extensions to the research described in this dissertation is not intended to be complete. We hope that the reader has found enough inspiration in this work to suggest further additions to the list. Supporting the next generation of advanced

database applications means we have to rethink how we slice up and present the functionality of transaction processing systems, in addition to broadening and narrowing specific functions. We believe that this dissertation is a step in this direction, and hope it leads to the further migration of extended transaction research results into practice.

Bibliography

[AAS93] D. Agrawal, A. El Abbadi, and A.K. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460–486, 1993.

[ABLL91] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Proceedings of the Thirteenth Symposium on Operating System Principles*, pages 95–109, 1991.

[ASK92] M. Rusinkiesicz A. Sheth and G. Karabatis. Using polytransactions to manage interdependent data. In Ahmed Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 555–582. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[ASSR93] P.C. Attie, M.P. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 134–145, Dublin, Ireland, 1993.

[AYWM90] Elmagarmid A., Leu Y., Litwin W., and Rusinkiewicz M. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 507–518, Brisbane, Australia, 1990.

[BD95] T. Braun and C. Diot. Protocol implementation using integrated layer processing. In *Proceedings ACM SIGCOMM'95*, pages 120–128, Boston, MA, 1995.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham. Asset: A system for supporting extended transactions. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, pages 44–53, Minneapolis, MN, 1994.

[Ber90] P.A. Bernstein. Transaction processing monitors. *Communications of the ACM*, 33(11):75–86, 1990.

[Ber96] P.A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[BGW93] D.G. Bobrow, R. Gabriel, and J.L. White. *CLOS in Context: The Shape of the Design Space.* MIT Press, 1993.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[BHMC90] A. Buchmann, M. Hornick, B. Markatos, and C. Chronaki. Specification of a transaction mechanism for a distributed active object system. In *Proceedings of the OOPSLA/ECOOP Workshop on Transactions and Objects*, pages 1–9, Ottawa, CA, 1990.

[BK91] N.S. Barghouti and G.E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–318, September 1991.

[BN96] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[BOH+92] A. Buchmann, M. Ozsu, M. Hornik, D. Georgakopoulos, and F. Manola. A transaction model for active distributed object system. In Ahmed Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 123–158. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[BR91] B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 16:163–199, September 1991.

[BS95] N. Bhatti and R. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings ACM SIGCOMM'95*, pages 138–150, Boston, MA, 1995.

[CDG+90] M.J. Carey, D.J. DeWitt, G. Graefe, D.M Haight, J.E. Richardson, D.T. Schuh, E.J. Shekita, and S.L. Vandenberg. The EXODUS extensible DBMS project: An overview. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[CFN96] J.C. Cleaveland, J.A. Fertig, and G.W. Newsome. Dividing the software pie. *AT&T Technical Journal*, 75(2):8–18, 1996.

[Che98] S.F Chen. *Recovery for Extended Transaction Models.* PhD thesis, Columbia University, New York, NY, Expected 1999.

[CM93] S. Chiba and T. Masuda. Designing an extensible distributed language with metalevel architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP)*, pages 482–501, Kaiserlautern, Germany, 1993.

[Chr91] P.K. Chrysanthis. *ACTA, A framework for modeling and reasoning about extended transactions.* PhD thesis, University of Massachusetts, Amherst, MA, 1991.

[CR91a] P.K. Chrysanthis and K. Ramamritham. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 103–112, Barcelon, Spain, 1991.

[CR91b] P.K. Chrysanthis and K. Ramamritham. A unifying framework for transactions in competitive and cooperative environments. *IEEE Bulletin of the Technical Committee on Data Engineering*, 4(1):3–21, 1991.

[CR92] P.K. Chrysanthis and K. Ramamritham. ACTA: The Saga continues. In Ahmed Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 349–398. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[CR94] P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[DE89] W. Du and A.K. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in interbase. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 347–355, Amsterdam, The Netherlands, 1989.

[dRS84] J. des Rivières and B. Smith. The implementation of procedurally reflective languages. Technical Report ISL-4, Xerox PARC, 1984.

[EPT95] D. Edmond, M. Papzoglou, and Z. Tari. R-OK: A reflective model for distributed object management. In *Proceedings of the RIDE '95 Workshop (Research Issues in Data Engineering)*, pages 34–41, Taipei, Taiwan, 1995.

[FO89] A. Farrag and T. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, 1989.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[GMGK+91] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modelling long-running activities as nested sagas. *IEEE Bulletin of the Technical Committee on Data Engineering*, 14(1):14–18, 1991.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of 1987 ACM SIGMOD International Conference on Management of Data*, pages 462–473, San Francisco, CA, 1987.

[GHKM94] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the 1994 IEEE Conference on Data Engineering*, pages 462–473, 1994.

[GHS95a] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From processing modeling to workflow automation infrastructure. *Distributed and Parallel Database*, 3(2):119–152, 1995.

[GHS95b] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From processing modeling to workflow automation infrastructure. *Distributed and Parallel Database*, 3(2):119–152, 1995.

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

[HC93] M. Hammer and J. Champy. *Reengineering the Corporation.* New York Publishing Co., New York, NY, 1993.

[Hei97] G.T. Heineman. *A Transaction Manager Component for Cooperative Transaction Models.* PhD thesis, Columbia University, New York, NY, 1997.

[HK93] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. Technical Report SMLI TR-93-14, Sun Microsystems Laboratories, Inc., 1993.

[HK97] G.T. Heineman and G.E. Kaiser. The cord approach to extensible concurrency control. In *Proceedings of the 1997 IEEE Conference on Data Engineering*, pages 562–571, Birmingham, UK, 1997.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[HS91] M. Hsu and A. Silberschatz. Unilateral commit: A new paradigm for reliable distributed transaction processing. In *Proceedings of the 1991 IEEE Conference on Data Engineering*, pages 286–293, Kobe, Japan, 1991.

[IB94] T. Imielinski and B.R. Badrinath. Wireless mobile computing: Challenges in data management. *Communications of the ACM*, 37(10):144–153, 1994.

[JS92] H.V. Jagadish and O. Shmueli. A proclamation-based model for cooperating transactions. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 265–276, Vancouver, British Columbia, Canada, 1992.

[KDLM95] G. Kiczales, R. DeLine, A. Lea, and C. Maeda. *Open Implementations Analysis and Design.* Tutorial Notes, ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1995.

[KdRB91] G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kic92] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, pages 1–11, Tokyo, Japan, 1992.

[Kle91] J. Klein. Advanced rule driven transaction management. In *Proceedings of the 36th Computer Society International Conference (CompCon)*, pages 562–567, Santa Clara, CA, 1991.

[KN93] Y.A. Khalidi and M.N. Nelson. The spring virtual memory system. Technical Report SMLI TR-93-09, Sun Microsystems Laboratories, Inc., 1993.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 95–106, Brisbane, Australia, 1990.

[KP92] G.E. Kaiser and C. Pu. Dynamic restructuring of transactions. In Ahmed Elmagarmid, editor, *Database Transaction Models for Advanced Transactions*, pages 265–296. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[KPng] G. Kiczales and A. Paepcke. *Open Implementations and Metaobject Protocols.* The MIT Press, 1998 (forthcoming).

[Kor83] H.F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, 1983.

[KS88] H.F. Korth and G. Speegle. Formal models of correctness without serializability. In *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data*, pages 379–386, Chicago, IL, 1988.

[Kru92] C.W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[Lab93] Unix System Labs. *TUXEDO system product overview.* Unix System Labs, Summit, N.J., 1993.

[LHP94] K-Y Lam, S-L Hung, and C. Pu. Two locking protocols based on epsilon serializability for real-time database systems. In *Proceedings of the International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, pages 63–72, Hong Kong, 1994.

[LKS91] E. Levy, H.F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of 1991 ACM SIGMOD*, pages 88–97, Denver, Colorado, 1991.

[LP98a] L. Liu and C. Pu. A Transactional Activity Model for Organizing Open-ended Cooperative Activities. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS-31)*, pages 380–387, Big Island of Hawai'i, HI, 1998.

[LP98b] L. Liu and C. Pu. Methodical Restructuring of Complex Workflow Activities. In *Proceedings of the 1998 IEEE Conference on Data Engineering*, pages 342–350, Orlando, Florida, 1998.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155, Orlando, FL, 1987.

[MHG+92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. Distributed object management. *Int. J.of Intelligent and Cooperative Information Systems*, 1(1):18–25, 1992.

[ML83] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of 2nd ACM SIGACT/SIGOPS Symposium on PODC*, pages 76–88, Montreal, Canada, 1983.

[MLO86] C. Mohan, B. Lindsay, and R. Obermark. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.

[MNB+94] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 25(10):59–70, May 1994.

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

[Mos85] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology, 1985.

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

[NKM93] M.N. Nelson, Y.A. Khalidi, and P.W. Madany. The spring file system. Technical Report SMLI TR-93-10, Sun Microsystems Laboratories, Inc., 1993.

[NZ90] M. Nodine and S. Zdonik. Cooperative transaction hierarchies: a transaction model to support design applications. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 83–94, Brisbane, Australia, 1990.

[O'N86] P.E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

[PAB+95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, pages 314–324, Copper Mountain, Colorado, 1995.

[PKH88] C. Pu, G.E. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 26–37, Los Angeles, CA, 1988.

[Rao91] R. Rao. Implementational reflection in silica. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 251–266, Geneva, Switzerland, 1991.

[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 Publishers, San Mateo, CA, 1992.

[RC97] K. Ramamritham and P. Chrysanthis. *Executive Briefing: Advances in Concurrency Control and Transaction Processing.* IEEE Computer Society Press, Los Alamitos, CA., 1997.

[RJY+88] R. Rashid, A. Tevanian Jr, M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[RKT+95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wasch, and P.Muth. Towards a cooperative activity model. In *Proceedings of the 21st International Conference on Very Large Data Bases,* pages 194–205, Zurich, Switzerland, 1995.

[RP90] K. Rothermel and S. Pappe. Open commit protocols for the tree of processes model. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 236–244, Paris, France, 1990.

[RP95] K. Ramamrithan and C. Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, December 1995.

[SBCM93] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the 1993 IEEE Conference on Data Engineering*, pages 325–333, Vienna, Austria, 1993.

[SBCM95] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases*, 3(4):325–360, 1995.

[Sch93] F. Schwenkreis. APRICOTS: A prototype implementation of a ConTract system. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 12–22, San Jose, CA, 1993.

[Ser91] O. Serlin. The TPC benchmarks. In J. Gray, editor, *Database and Transaction Processing Systems Performance Handbook*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[SGMS94] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM Transactions on Database Systems*, 19(1):117–165, March 1994.

[Smi82] B. C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, 1982.

[SMK+94] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994.

[SN92] K. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, 1992.

[SS84] P.M. Schwarz and A.Z. Spector. Synchronizing shared abstract data types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[SSU96] A. Silberschatz, M. Stonebraker, and J. Ullman. Database research: Achievements and opportunities into the 21st century. *ACM SIGMOD Record*, 25(1):52–63, 1996.

[Str93] R.J. Stroud. Transparency and reflection in distributed systems. *ACM Operating Systems Review*, 22(2):99–103, April 1993.

[SW95] R.J. Stroud and Z. Wu. Using metaobject protocols to implement atomic data objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 168–189, Aarhus, Denmark, 1995.

[Tra94a] Transarc Corporation, Pittsburgh, PA. 15219. *Encina Product Overview*, 1994.

[Tra94b] Transarc Corporation, Pittsburgh, PA. 15219. *Encina Toolkit Server Core Programmer's Reference*, 1994.

[VRS86] S. Vinter, K. Ramamritham, and D. Stemple. Recoverable actions in gutenberg. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 242–249, Hong Kong, China, 1986.

[WBT92] D.L. Wells, J. Blakeley, and C.W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 37(5):74–82, October 1992.

[Wei88] W.E. Weihl. Commutativity-based concurrency control for abstract data types. In *21st Annual Hawaii International Conference on System Sciences*, volume II Software Track, pages 205–214, Kona, HI, January 1988.

[WR92] H. Wachter and A. Reuter. The contract model. In Ahmed Elmagarmid, editor, *Database Transaction Models for Advanced Transactions*, pages 219–264. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[WYP92] K.L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proceedings of Eighth International Conference on Data Engineering*, pages 506–515, Phoenix, AZ, 1992. IEEE/Computer Society.

[Yok92] Y. Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–434, Vancouver, British Columbia, Canada, 1992.

[ZPL96] T. Zhou, C. Pu, and L. Liu. Adaptable, efficient, and modular coordination of distributed extended transactions. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, pages 262—273, Miami Beach, Florida, 1996.

Biographical Note

Roger Barga received the B.S. degree in Mathematics from Boise State University, Boise, ID, in 1985, and the M.S. degree in Computer Science from the University of Idaho, Moscow, ID, in 1987. Roger entered the doctoral program in Computer Science at the Oregon Graduate Institute, Portland, OR, in 1992. In 1996, Roger received an *Intel Graduate Fellowship Award* from the Intel Foundation.

Roger Barga began his professional career as a software engineer for Ore-Ida Foods while working towards his B.S. degree, and continued working as a software engineer and consultant throughout his M.S. degree. From 1987 to 1992, he was a research scientist at Battelle Pacific Northwest National Laboratory (PNNL), Richland, WA, where he was involved in research on machine learning and adaptive pattern recognition. From 1989 to 1992, he was an adjunct faculty member in the Department of Computer Science at Washington State University, Richland, WA, where he was responsible for teaching courses in artificial intelligence, pattern recognition, and machine learning. Since 1997, he has been with the Database Research Group, Microsoft Corporation, Redmond, WA. His current research interests are in application recovery and advanced transaction processing.