December 1996

Reflection on a legacy transaction processing monitor

Roger Barga

Calton Pu

To appear in proceedings of the Reflection'96 Conference San Francisco, California April 1996

Reflection on a Legacy Transaction Processing Monitor

Roger Barga and Calton Pu

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
email: {barga,calton}@cse.ogi.edu)

December 7, 1996

Abstract

In this paper we describe our experience applying the concepts of reflection to a legacy transaction processing (TP) monitor in order to support the implementation of extended transaction models. In the past ten years, numerous extended transaction models have been proposed to address the challenges posed by new advanced applications. Few practical implementations of these extended transaction models exist and none are being used in a commercial TP monitor. We believe the reason for this state of affairs is that the interface and functionality of commercial TP monitors is "locked in" to a fixed transaction model. We have developed the Reflective Transaction Framework as a practical method to implement extended transaction models on a commercial TP monitor. The design of our framework is based on the principles of computational reflection, and in particular open implementation. The implementation of our framework introduces transaction adapters, which are reflective software modules that provide a meta interface to the underlying TP monitor, allowing application developers the flexibility to adjust both the application programming interface and system functionality. Unlike classic reflective systems, the Reflective Transaction Framework applies reflection to a legacy TP monitor written in a non-reflective programming language. This paper focuses on the ability of the Reflective Transaction Framework to perform reflective computation and on the extent to which the legacy TP monitor supports this computation.

1 Introduction

In recent years, numerous extended transaction models have been proposed to address the requirements posed by advanced applications [Elm93]. A few of these models have been implemented as prototypes, but most remain mere theoretical constructs with no practical implementation [Moh94] and none are being used in a commercial product. We believe the main reason for this state of affairs is that the interface and functionality of commercial transaction processing (TP) monitors is *"locked in"* to a fixed transaction model, making it difficult, if not impossible, to readily use them to implement new extended transaction models. We have found [BP95], the problem is not due to an inherent inadequacy of modern TP monitor functionality, but rather lies with the visible aspects of the underlying implementation not permitting access to the required functionality. In this paper, we describe an approach to apply reflection to a commercial TP monitor to *"open up"* the interface and underlying functionality so that extended transaction models can be implemented.

We have introduced the *Reflective Transaction Framework* [BP95] as a practical method to implement a wide range of extended transaction models on a commercial TP monitor. The design of the Reflective Transaction Framework is based on the principles of *computational reflection* [Mae87], and in particular the *Open Implementation* approach [Kic92]. In the Reflective Transaction Framework, reflection is realized using *transaction adapters*, which are reflective software modules built on top of the TP monitor. Transaction adapters provide a *meta interface* to adjust aspects of the legacy TP monitor that are often hidden, effectively opening up the implementation so that programmers can explore alternative transaction model implementations and application interfaces. To demonstrate the practicality of our approach, we have implemented the Reflective Transaction Framework on top of Encina [Cor91], a commercial TP monitor distributed and supported by Transarc.

We believe the Reflective Transaction Framework represents a new application of reflective concepts. Unlike classic reflective systems, such as CLOS [BGW93] and 3-LISP [dRS84], the Reflective Transaction Framework applies reflection to a legacy TP monitor that was written in a non-reflective programming language. Concretely, the Reflective Transaction Framework demonstrates the practicality and usefulness of this new application of reflection to incrementally extend a legacy TP monitor. This approach also enables the *reuse* of available legacy functionality, which has obvious economic advantages. Another, more pragmatic contribution, is that the Reflective Transaction Framework presents the first practical method to implement a wide range of extended transaction models on an industrial-grade TP monitor.

This paper is organized as follows: Section 2 provides background and motivation for our approach. Section 3 presents the Reflective Transaction Framework, and discusses how reflection is implemented on a legacy TP monitor. Section 4 illustrates the implementation of an extended transaction model using our framework. Section 5 presents a comparison with related work, and Section 6 concludes the paper.

2 Background and Motivation

In order to open up the interface and functionality of a legacy TP monitor, we draw upon results stemming from work on *computational reflection*. The concept of reflection was first put forth explicitly by Brian Smith in his PhD thesis [Smi82] concerning the structure and organization of self-modifying procedures and functions, which was later summarized in a shorter paper [Smi84]. The essence of computational reflection is that a program is able to reason about its own behavior and then change it. While the concept of reflection originated in work on flexible programming language design, the notion has proven to be quite general. Over the years, researchers have incorporated the principles of reflection into a variety of application domains and software frameworks in a general move to make systems flexible [Str93, Yok92, EPT95]. This flexibility allows the systems to be made compatible with other hardware, function efficiently in a range of different circumstances and clients, and support application requirements that were not considered during development. Our motivation, then, is to investigate how we might apply reflection to a modular legacy TP monitor, to attack similar challenges of flexibility in the interface and implementation.

In the remainder of this section, we first present background information on transactions and the implementation challenges presented by extended transaction models, followed by a high-level description of the TP monitor. Finally, we return to discuss how results from reflection can be applied to a legacy TP monitor.

2.1 Transaction Models, Past and Present

The transaction concept has been used effectively in traditional database systems to synchronize concurrent access to a shared database and provide reliable access in the face of failures. A transaction is an atomic unit of work against the database, consisting of a *sequence* of data manipulation operations, typically `read` and `write`, along with *control operations* that affect the execution of the transaction. The control operations found in the traditional database transaction model are `Begin_Transaction`, `Commit_Transaction`, and `Abort_Transaction`. ACID properties of transactions (atomicity, consistency, isolation, and durability) guarantee correct concurrent execution as well as reliability [HR83]. The flexibility of a given transaction model depends on the way these four ACID properties are combined, along with the control operations available to the transaction.

Although powerful, it is commonly accepted that the traditional transaction model found in conventional database systems is lacking in *functionality* and *performance* when used for advanced application domains [Elm93]. Some of the considerations are that transactions in these domains are longer in duration and often require the release of intermediate results to other transactions during execution, requiring a relaxed notion of atomicity. This is the case, for example, in computer-aided design and manufacturing (CAD/CAM) applications. In some application domains, the property of isolation between transaction has to be relaxed to allow cooperation among transactions in order to accomplish a common task. This is the case in computer-supported co-

operative work environments. Hence, over the last five years a number of *extended transaction models* have been introduced to accommodate the diverse requirements of advanced application domains [PKH88, MP92, RC92, CR92, Moh94, WR93]. These models introduce new transaction control operations, such as the operation `Split` and `Join` operations introduced by the split/join transaction model [PKH88], and they associate "broader" interpretations of the ACID transaction properties to provide enhanced functionality while increasing the potential for improved performance.

While the potential benefits of these new extended transaction models have been known for some time, few practical implementations of these models have been developed [Moh94]. One reason for the lack of practical implementations is that conventional TP monitors have a closed interface, accepting only traditional database transaction control operations and an apparently rigid implementation, making it difficult, if not impossible, to implement extended transaction models. Systems programmers attempting to implement extended transaction behaviors are often frustrated by the complexity of the task, and, even if they succeed, the closed interface locks in the extended functionality, preventing application programmers from using the new extended transaction behaviors. As a result, to date most extended transaction models remain mere theoretical constructs with no practical implementation [Moh94].

2.2 TP Monitors in the Real World

In contrast to extended transaction models, TP monitors supporting ACID transactions are a well established technology that has been around for almost 20 years. They provide a general framework for transaction processing, supplying the "glue" to bind together the many functional components of a transaction processing system through services like multithreaded processes, interprocess communication, queue management, and system management [Ber90]. Early TP monitors, e.g. IMS/DC, CICS, were single monolithic systems used in proprietary mainframe environments to achieve high transaction rates in large-scale online transaction processing (OLTP) systems, such as airline reservation, electronic banking, or securities trading.

While early TP monitors were proprietary and constructed from single monolithic proprietary systems, modern TP monitors, such as Transarc's Encina, DEC's ACMS, and IBM's CICS/6000, are more modular and constructed from open transaction middleware. These middleware modules provide the basic functional building blocks that a TP monitor requires for transaction processing, such as a *Transaction Manager*, a *Lock Manager*, a *Log Manager* and a *Resource Manager* (i.e., DBMS). Each module provides its transaction services through a relatively simple and uniform application programming interface (API). The relationships among a transactional application and these basic functional components are depicted in Figure 1. In a commercial setting, we might find a TP Monitor such as Transarc Encina or DEC ACMS providing access to various resource managers, such as an Oracle or Informix relational DBMS.

Although the functional components of modern TP monitors allow users access to system behaviors and transaction services via the API, the size and complexity of the system forms

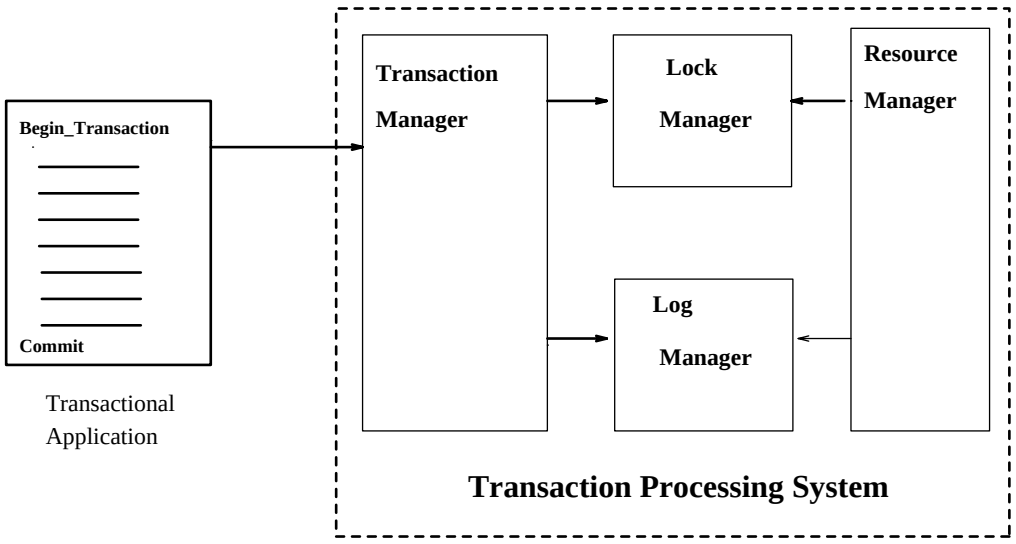

Figure 1: Modular Functional Components of a TP Monitor

a barrier to users wishing to extend the behavior of the TP monitor. Rather than a smooth progression in the level of complexity and degree of expertise required to perform customization with the available API, there is often a steep learning curve involved in learning all the functions and understanding how they can be combined.

2.3 A Practical Approach Based on Reflection

The challenge of opening up a legacy TP monitor, is therefore, characterized by two inter-related problems. The first problem is the *interface for customization*. Conventional TP monitors have a standard application-level interface that accepts only traditional transaction control operations, such as `Begin_Transaction` and `Commit_Transaction`. Extending the TP monitor through the available API would require application programmers to learn intricate details of the TP monitor architecture and the available API. The second problem is the *level of customization*. TP monitor system-level code functions 'underneath' the application code and is not subject to the same abstractions as the application. Indeed, with the API approach the TP monitor must be customized outside the application, rather than within it. An application, then, has no means to specify its requirements for extended transaction behaviors at runtime. At best, systems programmer could hope to anticipate the requirements of application programmers and adjust the TP monitor behavior through the API to implement an extended transaction model *a priori*, but this is at the cost of reusability of the TP monitor by applications with other requirements. Both of these problems combine to give users no easy way to reach in to the TP monitor and customize the way in which transaction functionality and interface are provided.

Computational reflection gives us a conceptual tool, the notion of a reflective module, to address these two problems. A reflective module contains a *representation* of the system, and has a *causal connection* between that representation and the actual behavior of the system. The

causal connection is two-way; not only are changes in the system reflected in equivalent changes to the representation, but changes in the representation will also cause changes in the behavior of the system. A reflective module presents the application developer with a base interface that provides access to the default system behavior. In addition to the base interface, a reflective module presents a separate *meta interface* which a programmer can use to "reach in" and adjust the representation to add new behaviors to the system. In the next section we describe the Reflective Transaction Framework, demonstrating how reflection and the notion of a reflective module can used to enhance a legacy TP monitor, providing the flexibility to open the interface and implementation.

3 A Reflective Framework for Implementing ETMs

We now turn our attention to the design of the Reflective Transaction Framework. To open up the functionality of a legacy TP monitor we introduce *transaction adapters*, which are add-on reflective software modules that provide a meta interface to the underlying TP monitor. In addition, to open up the application interface to the TP monitor we introduce a *separation of programming interfaces*. In the framework, transaction adapters provide an interface for default transaction control operations, an extensible interface for extended transaction control operations, and the meta interface to control implementation level concerns.

Although the Reflective Transaction Framework applies reflection in a non traditional sense, to a legacy TP monitor written in a non-reflective programming language, it nevertheless incorporates many of the ideas from the traditional notion of computational reflection. An important difference is that reflecting on a legacy system implies a strict separation between base and meta objects, as opposed to the metacircular interpreters from traditional computational reflection. In our framework this base/meta object separation in implemented by introducing transaction adapters. Adapters do not have a full model of the entire monitor, but only a partial model of selected aspects of the underlying TP Monitor. However, reflection is still very much present and, as we will demonstrate, it does enable transaction adapters to open up the legacy TP monitor functionality and extend it to implement extended transaction models.

3.1 Transaction Adapters

In the Reflective Transaction Framework, reflection is realized through *transaction adapters*, which are software modules built on top of the legacy TP monitor. Each adapter corresponds to a particular aspect (functional component) of the TP monitor, such as transaction execution, lock management, conflict detection, and log management. Transaction adapters contain a number of *meta objects* which represent selected behaviors of the underlying functional components, and a *meta interface* (what some might call a Meta-Object Protocol or MOP) to control the behavior of that component. This is illustrated in Figure 2. The behavior of the TP monitor functional components can be changed by explicitly updating these meta objects through the adapter meta

interface, allowing users to "reach in" and make changes to the behavior of the TP monitor. As such, adapters provide access to aspects of a legacy TP monitor that are often hidden.

Transaction adapters are application level objects, which means that applications have access to them through the meta interface. Changes or modifications made to an adapter by an application change the behavior of the TP monitor component *for only that application.* For example, if an application would like to change the isolation for a transaction, it can use the appropriate meta interface operations to change the conflict detection method for that transaction. Therefore, transaction adapters allow an application to enhance the underlying mechanisms of a legacy TP monitor incrementally, and dynamically, in a modular manner at the granularity of each (extended) transaction execution.

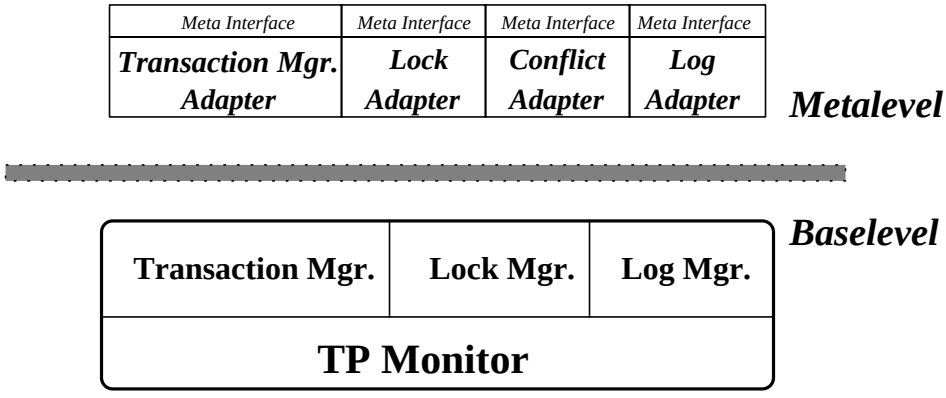

Figure 2: Transaction Adapters in the Reflective Transaction Framework.

In the context of a legacy TP monitor, *reflection* is the ability of an executing TP monitor to make selected aspects of the underlying functional components, such as transaction operation dispatch, lock management, conflict detection and log file updates, to be themselves the subject of computation. The steps involved in this reflection consist of: *reification* of structures and behaviors of the underlying TP monitor component into objects that represent or model aspects of the underlying TP monitor component, *reflective computation* using the reified aspects as data, and *reflective update* that modifies the actual computational state of the underlying TP monitor component.

In the Reflective Transaction Framework, *reification* is the representation of structural and computational aspects of the underlying TP monitor component as an object within the corresponding transaction adapter. Reification in our framework is based on *callbacks*, also known as *upcalls* [Cla85]. Upcalls support efficient cross-layer communications and enable the functional components in the TP monitor to pass relevant state information to a transaction adapter where it is reified, as illustrated in Figure 3. The most important decisions to be made in designing a transaction adapter are what aspects of the underlying TP monitor component should be reified. As an example, for the *Lock Adapter* depicted in Figure 3, such aspects include the locks being held by a transaction, pending lock requests, the procedure used to grant lock requests, and the structure of the lock table. Depending on the transaction model, however, several other aspects could also be reified, for example the operations being performed on a locked data object, or the mode in which a lock has been granted to a transaction. To make adapters as flexible as possible,

they are designed to be extensible. Should the need arise, additional aspects of the underlying functional component can be reified by adding appropriate upcalls to the TP monitor and associated reification methods to the transaction adapter. Reifying selected aspects of the underlying TP monitor component into metalevel objects that are dynamically accessible and modifiable enables *reflective computation* and *reflective update*.

In the Reflective Transaction Framework, the shift in computation from the TP monitor functional component to *reflective computation* in the transaction adapter occurs in an event-driven manner. A transaction *significant event* is raised whenever a transaction attempts to change state, e.g. the transaction aborts or commits, or when a transaction requests a service from the TP monitor. For each transaction event there is an adapter assigned to process the event, and when that event is raised, control is passed to the assigned transaction adapter along with all information relating to the event. For example, when the LOCK MANAGER detects a lock conflict between two transactions during a lock request, control is passed to the *Lock Adapter* through an upcall, along with all information pertaining to the conflicting request. The *Lock Adapter* can then apply operation or application-specific semantic information to determine if the request should be granted according to the semantics of the transaction model. The *Lock Adapter* can then grant the lock request, or it by simply returning control back to the LOCK MANAGER, effectively implementing semantics-based concurrency control [BPZH95]. As this example illustrates, reflective computation not only allows transaction adapters to expose default behaviors of the underlying TP monitor, but also to augment legacy functionality with new extended transaction model behaviors.

If the reflective computation updates the reified data, the modifications are *reflected* down to the actual computational state of the underlying TP monitor component in what is called a *reflective update*. In the Reflective Transaction Framework, reflective update is implemented through calls to the API provided by each TP monitor functional component. Through the API the transaction adapter can update the structures and computational state of the underlying functional component. The most challenging issue when implementing an adapter is to identify the appropriate API calls in order to implement each reflective update. Ideally, this task is performed only once, by the designer of the Reflective Transaction Framework, who is familiar with the inner workings of the monitor functional components. When an adapter needs to perform a reflective update, it issues the appropriate sequence of API calls, as illustrated in Figure 3. Thus, each transaction adapter not only reifies aspects of the TP monitor functional component, enabling reflective computation, but also provides the means to affect that state and control the component's behavior through reflective update. This is termed *causal-connection* [Mae87], and is satisfied by transaction adapters in the Reflective Transaction Framework.

To design each transaction adapter, we followed systematic procedure to select aspects of the TP monitor for reification, to determine how they would be represented, and the ways they could be manipulated through the adapter meta interface. This was an important process in the development of the framework, because it determines what extended behaviors can be realized on the underlying TP monitor and, correspondingly, what extended transaction models can be implemented by the Reflective Transaction Framework. Details of the systematic procedure we introduced to design the adapters in the framework are not appropriate for this discussion, and can

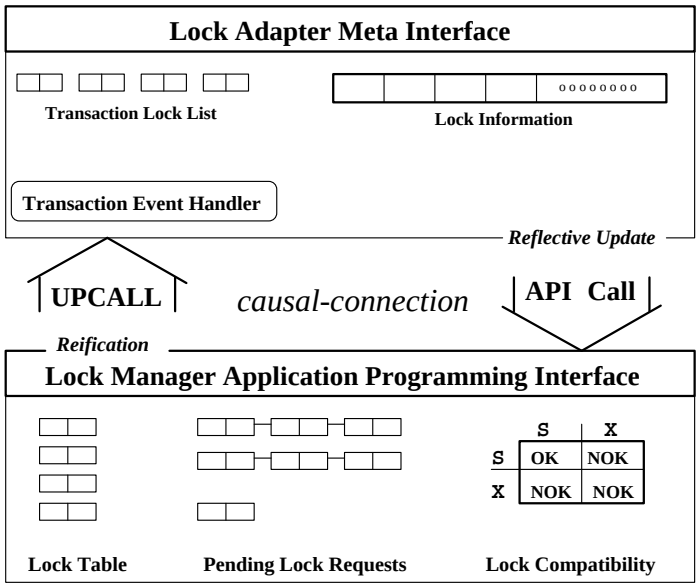

Figure 3: Reflective Update and Reification

be found elsewhere [BP95]. It is important to note, however, following the systematic procedure we designed a generic framework to describe extended transaction behaviors within which a wide range of extended transaction models can be implemented [BP95]. For practicality, the functional design of each transaction adapter was based on the commands and functionality of the well-documented TP monitor reference architecture [GR93]. The TP monitor reference architecture is general enough to allow us to make observations on TP monitors in general, and yet concrete enough to make implementation details obvious in a modern commercial TP monitors. Further, our implementation of the adapters relied only on a small, widely supported, set of commands found in the API of the TP monitor reference architecture.

In the remainder of this section, we will highlight some of the objects reified by transaction adapters, and describe selected commands from the adapter meta interface.

Transaction Management Adapter — reifies transactions executing extended behaviors, and provides a meta interface to control these transactions and adjust the behavior of the underlying TRANSACTION MANAGER functional component. Commands in the *Transaction Management Adapter* meta interface include: `Instantiate`, `Reflect`, `Delegate_Ops`, `Form_Dependency`, `Create_Group`, `Create_Tran`, `Terminate_Tran`, and `Wait`. Primary meta objects reified by the *Transaction Management Adapter* include a metatransaction descriptor for each extended transaction, see Figure 4, a reflective transaction table, and a transaction dependency graph.

An extended transaction is entered into the reflective transaction framework through the command `instantiate`, at which time a metatransaction descriptor is created for the transaction. All information relating to the execution and current status of the transaction will be reified in this metatransaction descriptor. Afterwards, an instantiated transaction can be assigned a set of extended transaction control operations (semantics) through the command `reflect`. When an extended transaction invokes a control operation, the actual code executed is determined by

```
metatransaction Descriptor{
  self: <TRID>;
  groupID: [<TRID>];
  execMode:  <state>;
  dependsOn: [<TRID>, <TRID>,...];
  waitOn: [<event>];
  lockList: [<lockID>, <lockID>,...];
  opList: [<opName>, <opName>,...];
  initiateOperations: {<Begin, atomicBegin>};
  processOperations:   nil;
  terminateOperations:{<Commit, atomicCommit>,
                       <Abort, atomicAbort>}};
```

Figure 4: Metatransaction Descriptor for an Extended Transaction.

its metatransaction. For example, if the transaction were to invoke the `Commit` operation this would result in a transaction significant event being raised and control would be passed to the *Transaction Management Adapter* Processing the command involves first verifying this control operation is permitted for the transaction; a simple comparison with the metatransaction descriptor is performed. Once the *Transaction Management Adapter* has verified the `commit` operation is valid then the function identified in the metatransaction descriptor is executed. Afterwards, any results from the transaction control operation are returned to the base level transaction as if for a normal operation call.

Conflict Adapter — reifies information on the conflicts that occur between transactions attempting to acquire shared resources, and provides a meta interface to control the definition of conflict and appropriately adjust the behavior of the underlying LOCK MANAGER. Commands in the *Conflict Adapter* meta interface include: `Relax_Conflict`, `No_Conflict`, `Allow`, `Wait` and `Revoke`. Primary meta objects reified by the *Conflict Adapter* include a a compatibility table defining conflict relationships between operations, and a no-conflict table that records all conflicts explicitly *relaxed* between extended transactions.

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

Lock Adapter — reifies information on locks held by transactions and on the lock table, and provides a meta interface to control these locks and adjust the behavior of the underlying LOCK MANAGER functional component. Commands in the *Lock Adapter* meta interface include: `Release_Lock`, `Acquire_Lock`, `Delegate_Lock`, `Share`, `Wait`, `Peak` and `Upgrade_Mode`. Primary meta objects reified by the *Lock Adapter* include a transaction lock list, lock mode table, and an active locks list.

Locks on data objects can restrict the ability of a transaction to see the effects of other transactions on data objects while they are executing. The *Lock Adapter* allows greater control over the visibility of data objects by enabling a transaction to grant other transactions access to data objects on which they hold locks. The *Lock adapter* also enables an extended transaction to delegate ownership of its locks to another transaction prior to termination through the `delegate_lock` command. The `delegate_lock` command allows the transaction to specify whether it wishes to delegate all the locks it currently holds or only those for specified data objects. The conflict adapter records access rights granted between extended transactions in the no-conflict table, while the lock adapter provides the transactions access to the locked data object(s).

3.2 A Separation of Programming Interfaces

Application programmers develop transactional applications using a set of transaction model-specific verbs, or *transaction control operations*. For example, classic database transactions are initiated by the control operation `Begin-Transaction`, and terminated by either a `Commit-Transaction` or `Abort-Transaction` control operation. Extended transactions, on the other hand, often introduce additional operations to control their execution, such as the operation `Split-Transaction` introduced by the split/join transaction model [PKH88], or the operation `Join-Group` introduced in the cooperative group model [MP92, RC92]. Indeed, a transaction model defines both the control operations available to a transaction as well as the semantics of these operations. For example, whereas the `Commit-Transaction` operation of the classic database transaction model implies the transaction is terminating successfully and that its effects on data objects should be made permanent in the database, the `Commit-Transaction` operation of a member transaction in a cooperative transaction group implies only that its effects on data objects be made persistent and visible to transactions that belong to the same group.

To accommodate the diversity between different extended transaction models, we introduce a separation of programming interfaces to the TP monitor. This separation follows the open implementation approach [Kic92], pioneered in the meta-object protocol [KdRB91], in which the *functional interface* is separated from the *meta interface*. The purpose of the meta interface is to modify the behavior, or semantics, of the functional interface. In our separation of interfaces, presented figuratively in Figure 5 and described below, both the transaction demarcation interface and extended transaction interface are functional, subdivided for clarity only.

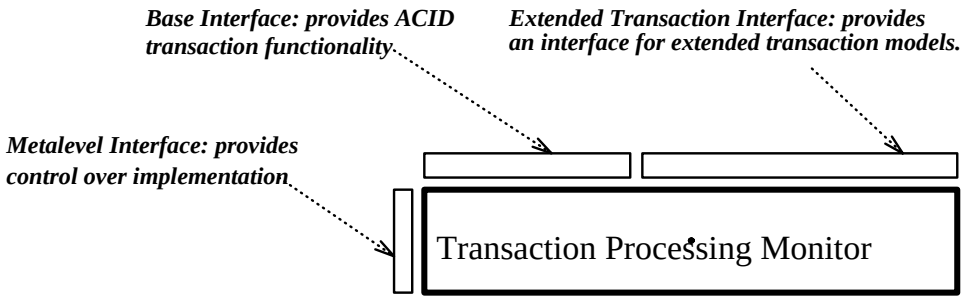

Figure 5: Separation of interfaces to the Reflective Transaction Framework.

- **Transaction demarcation interface** — presents the standard transaction interface offered by the TP monitor and when used alone provides *default* transaction behavior of ACID transaction semantics. Control operations in the transaction demarcation interface include: `begin-transaction`, `commit-transaction`, and `abort-transaction`.

- **Extended transaction interface** — presents an extensible interface to new *extended* behaviors added to the TP monitor and used when applications require extended transaction functionality and semantics. Operations in the extended transaction interface include transaction control operations defined by specific extended transaction models, such as the operations `Split` and `Join` for the split/join transaction model [PKH88].

- **Meta interface** — allows applications to view aspects of the underlying TP monitor functionality and to make modifications. The meta interface provides commands for programmers to "reach in" to the implementation and adapt it to the needs of a specific extended transaction model. Some of the operations in the meta interface include: `instantiate`, `reflect`, `delegateOp`, `delegateLock`, `formDependency`, and `noConflict`.

The separation of programming interfaces to the TP monitor provides a way not just to talk about existing transaction models, but to also introduce new extended transaction behaviors and interfaces. Default transaction behaviors remain available through the standard *transaction demarcation interface*. New extended transaction behaviors can be defined using the *meta interface*, and made available to to application through the introduction of new extended transaction control operations in the *extended transaction interface*. The extended transaction interface, then, *augments* the transaction demarcation interface with new extended control operations.

Using the Reflective Transaction Framework, TP systems programmers don't need to "second guess" the specific needs of application developers, or restrict the applicability of the TP monitor to a subset of extended transaction models, but can use the meta interface to add new extended transaction models as necessary. Application programmers are not constrained to a fixed set of control operations or a fixed transaction model, but are free to select the appropriate transaction model to meet the needs of the application. Moreover, they can access these new extended transaction behaviors using programming skills they already posses, namely through transaction control operations.

3.3 An Encina Implementation

For our current implementation of the Reflective Transaction Framework we are using Encina, a commercial TP Monitor distributed by the Transarc Corp. Transaction services for Encina are provided by the Encina Toolkit [Tra], which is composed of a small number of transaction middleware service modules, including: Transaction Service Module (TRAN) that provides transaction execution control and default transaction control operations (begin, commit, abort), Lock Service Module (LOCK) that provides a logical locking package to guarantee transaction isolation and, Log Service Module (LOG) that provides write-ahead log support

for transaction updates and crash recovery. The transaction middleware service modules of the Encina Toolkit provide the basic building blocks of the TP monitor reference architecture [GR93, pp. 21], and have been used by a number of computer system providers to implement various TP monitors, including IBM's CICS/6000 TP monitor, DEC's ACMSxp TP monitor [BCDW95] and, of course, the Encina TP monitor [Cor91].

Each module in the Encina Toolkit provides access to its transaction services and behaviors through a relatively simple and uniform API. In addition, each module provides a set of *transaction callbacks* that allows the user to register a function that is to be called during a transaction event. In Encina, transaction events include a transaction changing execution state or requesting a resource. Callbacks not only pass data to the registered function, but allow the function to modify default system handling of the event. In our implementation we use these existing API calls and transaction callbacks to form the *causal-connection* between the modules in the Encina Toolkit and transaction adapters in the Reflective Transaction Framework. Transaction adapters use the callback facility for reification, and the standard Toolkit API for reflective updates.

While we had access to Encina source code, the implementation did not require any changes to the fundamental structures or functions of the Encina Toolkit. In a few cases, however, it was necessary to modify the callback function arguments to pass additional information to the adapter for reification. Because of the availability of transaction callbacks and rich API command set in the Encina Toolkit, we have been able to fully integrate the Reflective Transaction Framework into Encina. An extended transaction running on Encina behaves just like a standard ACID transaction. Having opened the implementation of Encina, we can adjust the extended transaction's behavior as needed. Extended transactions, generally speaking, retain most of their base-level semantics provided by Encina and simply gain some additional behavior, such as semantic notions of conflict or new extended transaction control operations; looking back on the split/join example, the model *added* the operations `split` and `join`, while the operations `begin`, `commit`, and `abort` retain their original meaning (implementation).

Our Encina implementation has demonstrated the *practicality* of the Reflective Transaction Framework. A question which naturally arises is how *portable* the framework is to TP monitors not constructed from the Encina Toolkit, and to transaction services found in relational and object-oriented database servers. As pointed out in Section 3.1, transaction adapters were designed in the context of the TP monitor reference architecture to use only a small, widely supported, set of API commands. However, to answer this question concretely, we are compiling a list of the required transaction functionality and API commands for reflective update, along with the necessary upcalls for reification. When complete, we will compare this list against other transaction facilities to assess the portability of the Reflective Transaction Framework to other systems.

4 Example of Implementing Extended Transactions

In this section we present an example to illustrate how extended transaction functionality can be implemented using the Reflective Transaction Framework. This example is based on the split/join transaction model [PKH88], in particular the transaction control operations `split` and `join`. We first informally define the extended transaction model, demonstrate how the meta interface is used to synthesize new extended transaction functionality, illustrate how an application can use this new functionality and, finally, demonstrate *how* transaction adapters function to implement the extended transaction behaviors. Readers interested in additional examples of implementing extended transaction models, are referred to our previous paper [BP95].

4.1 Split/Join Transaction Model

The split/join transaction model was proposed for open-ended activities such as computer-aided design and manufacturing (CAD/CAM). Open-ended activities are characterized by uncertain duration, uncertain developments and interaction with other concurrent activities. Due to these characteristics, sometimes it is desirable to release earlier modified data of a transaction to other transactions. The split/join transaction model provides two operations to dynamically restructure transactions, namely `split` and `join`. A transaction T may *split* into two transactions T_a and T_b, providing applications with a mechanism to release data objects that are no longer needed and, hence, release intermediate results to other transactions. Two transactions can also *join* together to become one transaction, or use combinations of split and join to allow transfer of resources from one transaction to another.

4.1.1 synthesizing the extended functionality of `split` and `join`

When a transaction T_1 splits, by executing the transaction control operation `split(`T_2`)`, it must first create a new transaction (T_2) and then delegate responsibility for executing some of its operations to this new transaction. To be more precise, T_1 transfers to T_2 responsibility for all uncommitted operations on a particular set of data objects, referred to as the *DelegateSet*. In practice, users define the DelegateSet by selecting the objects to split from the re-structured transaction. At the time of the split, a new transaction is created, instantiated, and then operations invoked on objects in the DelegateSet by T_1 are delegated to T_2. The transactions T_1 and T_2 can then commit or abort independently. The following code segment illustrates how the `split` transaction control operation is synthesized using commands in the meta interface:

```
split(NewTran, DelegateSet){
  // instantiate new transaction.
  instantiate(NewTran);
  // add split/join transaction interface through reflection.
  reflect(NewTran, SplitJoin);
  // delegate locks related to objects in delegate set.
  delegate_lock(NewTran, DelegateSet);
  // delegate ops related to objects in delegate set.
  delegate_op(NewTran, DelegateSet);
  // initiate execution of the newly created transaction.
  begin(NewTran);
  // return execution control to base-level transaction
  return;
}
```

Figure 6: `Split` transaction control operation.

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

```
join(DestTran,DelegateSet){
  // delegate locks related to objects in DelegateSet.
  delegate_lock(DestTran, DelegateSet);
  // delegate operations related to objects in DelegateSet.
  delegate_op(DestTran, DelegateSet);
  // terminate execution of T1.
  commit(self);
  // return control to invoking transaction.
  return;
}
```

Figure 7: `Join` transaction control operation.

Once the extended functionality of the `split` and `join` transaction control operations have been defined using the *meta interface*, the can then be added to the *extended transaction interface* where they will be available for applications to use.

4.1.2 Application Programming Using the `split` Operation

In order to motivate the need for the `split` and `join` operations, 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, the principal engineer might like to split off responsibility for the design of specific subsystems to component engineers who can either join their results 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 an easy and straightforward manner but can be easily satisfied by the split/join transaction model. The code fragment below outlines how an application programmer could use the split and join operations to dynamically restructure a transaction in order to release subsystem data objects and operations to a separate transaction and, later, join with a separate transaction:

```
Begin_Transaction PE_Tran                    (1)
begin
    instantiate(PE_Tran)                     (2)
    reflect(PE_Tran, SplitJoin)              (3)
    ...
    ...{ data manipulation }
    ...
    split(CE_Tran, Subsystem)                (4)
    ...
    ...{ data manipulation }
    ...
    join(QA_Tran,*)                          (5)
end
Commit_Transaction {CAD_design}              (6)
```

Line 1 declares the beginning of the principal engineer's transaction using the `Begin_Transaction` command found in the the *base 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` *meta interface* command in line 2 is to notify the Reflective Transaction Framework of the programmers intention to "renegotiate" the base transaction model. The `reflect` *meta interface* command in line 3 details the terms of the renegotiation, selecting the split/join model for the transaction. The importance of the reflect 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, it 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 new

transaction created for the component engineer and *Subsystem* is 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 program, *QA_Tran*.

One can see from this example that there is no description of creating the new transaction for the component engineer, no explicit delegation of the locks held on data objects in *Subsystem*, and no explicit delegation of the data manipulation operations pertaining to *Subsystem* when the application is written. With the exception of the instantiate and reflect operations, the programmer simply uses familiar transaction control operations to write the application.

4.1.3 Transaction Adapters Behind the Scenes

Continuing with our example, we now examine how transaction adapters work behind the scenes to support extended transaction behavior on a legacy TP monitor. We begin with the `instantiate` meta interface command in line 2. During execution, the instantiate command causes control to be passed to the *Transaction Management Adapter*, which first creates a *metatransaction descriptor* and reifies information for the transaction *PE_Tran*, including the transaction identifier (TRID), current execution status of the transaction, and control operations available to the transaction. Next, the *Transaction Management Adapter* directs the other adapters to create initial entries for objects will be reified for this transaction during its execution, and then it returns control back to the base transaction for processing. The `reflect` command in line 3 also causes control to be passed to the *Transaction Management Adapter*, which updates the metatransaction descriptor, as illustrated below, to contain the transaction control operations `split` and `join`, specified by the split/join extended transaction model.

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

Processing resumes on the base TP monitor, until the transaction control operation `split(CE_Tran, Subsystem)` is processed in line 4. Split is a transaction control operation defined the *extended transaction interface* for the transaction *PE_Tran*. When the transaction invokes a control operation, the actual code executed is determined by its *metatransaction* (see Figure 8). When the `split` operation is invoked by the transaction, processing involves first verifying this control operation is permitted for the transaction, and once it has been verified then the function is executed, as illustrated in Figure 8. For the execution of the split operation, as defined in Figure 6, the first meta interface command directs the *Transaction Management Adapter* to create a metatransaction descriptor for the new transaction *CE_Tran*. This change is reflected down

onto the TRANSACTION MANAGER, resulting in the creation of a new base level transaction. The commands instantiate and reflect are then processed by the *Transaction Management Adapter* to initialize the meta objects for the transaction *CE_Tran*. Next, the *Lock Adapter* delegates locks on all data objects in the delegate set *Subsystem* from the transaction *PE_Tran* to the transaction *CE_Tran*. This change is first made first to the meta object *lockTable*, and through causal connection the change is reflected down to the LOCK MANAGER through the API commands `releaseLock` and `acquireLock`. Once the delegate_lock command is complete, the *Transaction Management Adapter* processes the delegate_op command. Finally, the begin command is processed by the *Transaction Management Adapter*, which sets the execution mode of the transaction *CE_Tran* to active and returns control to the TP monitor to begin base level processing.

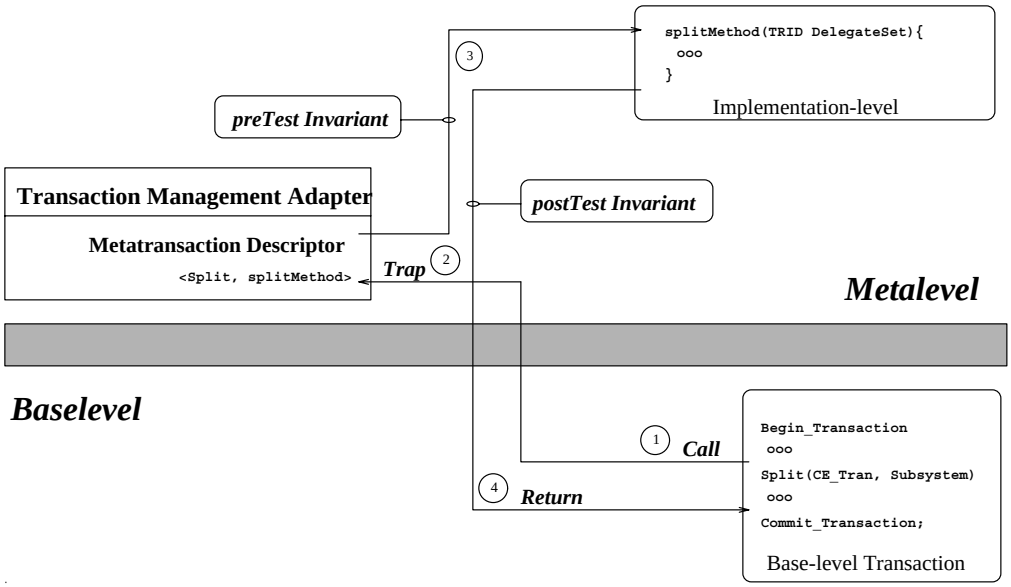

Figure 8: Transaction control operation redirection

5 Comparison to Related Work

In this section we compare the Reflective Transaction Framework to related efforts in implementing extended transaction models and reflective systems.

There exist only a small number of research efforts on implementing extended transaction models, similar in spirit to the Reflective Transaction Framework. Two noteworthy systems are ASSET [BDG+94] and TSME [GHKM94]. Similar to our framework, these systems are designed to facilitate the implementation of extended transaction models. However, they simply present the user with a closed application interface and a fixed selection of mechanisms from which a predetermined set of extended transaction models can be implemented. In our approach, the user is presented with a flexible framework in which the functionality and interface for an extended transaction model can be *created*, rather than a fixed selection of mechanisms from which particular extended transaction models can be *selected*.

ASSET [BDG+94] provides a set of new language primitives that enable the realization of various extended transaction models in an object-oriented database setting. In addition to the standard database control operations `Begin`, `Commit` and `Abort`, ASSET provides three new primitives: `form-dependency` to establish structure-related inter-transaction dependencies, `permit` to allow for data sharing without forming inter-transaction dependencies, and `delegate` which allows a transaction to transfer responsibility for an operation to another transaction. Using these new primitives, it is possible to synthesize certain extended transaction control operations within the program. However, the task of synthesizing new control operations is a skill that each programmer would necessary have to learn and the task must be repeated for each transaction that requires the operation.

TSME [GHKM94] consists of a transaction specification facility that understands TSME's transaction specification language, and drives the transaction management mechanism to configure a run-time system in order to support specific extended transaction models. The transaction management mechanism is programmable, but uses templates to describe existing extended transaction models. TSME is a toolkit based system, in which certain expressions in the specification language are mapped to certain configurations of *pre-built* components in the transaction management mechanism. If the needs of the application fall outside of this pre-built set, there is no recourse for the programmer — even though the transaction facility may be fully capable of implementing the required behavior.

As for reflective systems, most of the work on reflection has been on procedural reflection, where the meta-level directly implements the base-level. Some notable exceptions are Rok Sosic's work on Dynascope [Sos92b, Sos92a] and the Synthetix project [PAB+95]. Dynascope is a programming environment for directing the execution of traditional compiled languages. Program directing involves two processes, an *executor* and a *director*. When a program is executed in the Dynascope environment, the executor process generates an execution stream which the director process monitors. When selected events occur, similar to *breakpoints*, computation is shifted from the application to the Director where event-specific processing is performed on behalf of the application. This approach is similar to that of the Reflective Transaction Framework, in that adapters respond to selected transaction significant events and perform processing on behalf of the transaction. However, unlike transaction adapters, the Dynascope director does not maintain a causal connection with underlying application, and it requires a special-purpose programming environment for the necessary instrumentation.

Also, the Synthetix project [PAB+95] is studying specialization of operating systems, which can be viewed as a form of reflection. The Synthetix notion of specialization classes [CPW] provides a declarative meta interface through which an application can specify it's particular specification needs. Software tools then apply these specializations to the operating system to achieve the desired performance and functionality goals of the application.

6 Conclusions and Future Research

In this paper we have described an approach to apply reflection to a legacy TP monitor in order to support the implementation of extended transaction models. We described the Reflective Transaction Framework, in which reflection is manifested in a small number of add-on software modules called transaction adapters. Transaction adapters *open up* the functional components of the legacy TP monitor and present a meta interface through which users can adjust the behavior of the functional component according to their requirements. The Reflective Transaction Framework provides application programmers who find the default transaction model insufficient for their applications, the means to reach in to a conventional legacy TP monitor and implement new extended transaction models.

The Reflective Transaction Framework represents a new application of reflective concepts. While there have been papers that discuss various aspects of reflection and classify metalevel reflective architectures from different viewpoints, there are few previous works that apply reflection to legacy systems. One contribution of the Reflective Transaction Framework is that it demonstrates the practicality and usefulness of this new application of reflection to incrementally extend a legacy TP monitor. In general, this requires very little change at the underlying TP monitor. A distinct advantage of this approach is that of *reusability*. A second, more pragmatic contribution of the Reflective Transaction Framework, is that it provides the first practical method to implement a wide range of extended transaction models on an industrial-grade TP monitor. By doing so, we hope this will enables application developers to draw conclusions from direct experience in applying extended transaction models in real, working environments.

Our current implementation of the Reflective Transaction Framework is implemented on the commercial TP monitor Encina. We are currently in the process of measuring and optimizing the performance of this implementation. In addition, we are working to extend the ideas of the framework to other TP monitors, and to other research challenges in advanced transaction processing, such as semantics-based concurrency control protocols [BPZH95]. It is our hope that this work will not only provide solutions of practical value to these challenging problems, but provide insights into the general application of the notions of reflection and open implementation to legacy systems.

References

[BCDW95] R.K. Baaif, J.I. Carrie, W.B. Drury, and O.L. Wiesler. ACMSxp open distributed transaction processing. *Digital Technical Journal*, 7(1):23–33, 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*, pages 44–53, May 1994.

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

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

[BP95] Roger S. Barga and Calton Pu. A practical and modular method to implement extended transaction models. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995.

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

[Cla85] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 171–180, Orcas Island, Washington, December 1-4 1985.

[Cor91] Transarc Corp. *Encina Product Overview*. Transarc Corp, Pittsburgh, PA., 1991.

[CPW] Crispin Cowan, Calton Pu, and Jonathan Walpole. Specialization Objects: A Reflective Interface for Specialization. Submitted for review.

[CR92] P.K. Chrysanthis and K. Ramamritham. *ACTA: The SAGA Continues*, chapter 10. Morgan Kaufmann, 1992.

[dRS84] Jim des Rivières and Brian Smith. The implementation of procedurally reflective languages. Technical Report ISL-4, Xerox PARC, June 1984.

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

[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*, 1995.

[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, Feb 1994.

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

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. See `http://www.xerox.com/PARC/spl/eca/oi.html` for updates.

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

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

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

[PAB+95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

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

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

[Smi82] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, 1982.

[Smi84] B.C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[Sos92a] Rok Sosic. Dynascope: A tool for program directing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, volume 27, pages 12–21, July 1992.

[Sos92b] Rok Sosic. *The Many Faces of Introspection.* PhD thesis, University of Utah, 1992.

[Str93] R. J. Stroud. Transparency and reflection in distributed systems. *ACM Operating Systems Review*, 22(2):99–103, April 1993.

[Tra] Transarc Corporation, Pittsburgh, PA. 15219. *Encina Toolkit Server Core Programmer's Reference*.

[WR93] H. Wachter and A. Reuter. *Database Transaction Models for Advanced Transactions*, chapter The ConTract Model. In Elmagarmid [Elm93], 1993.

[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)*, October 1992.