Advanced Transaction Models and Architectures

Advanced Transaction Models and Architectures

Edited by

Sushil Jajodia and Larry Kerschberg
George Mason University
Fairfax, VA

Springer Science+Business Media, LLC

Library of Congress Cataloging-in-Publication Data

Jajodia, Sushil and Kerschberg, Larry
Advanced transaction models and architectures / edited by Sushil Jajodia and Larry Kerschberg.
P. Cm. --
Includes bibliographical references and index
ISBN 978-1-4613-7851-8 ISBN 978-1-4615-6217-7 (eBook)
DOI 10.1007/978-1-4615-6217-7
1. Reliability (Engineering) 2. Electronic digital computers--Reliability. 3. Computer software--Reliability. 4. Fuzzy sets. 5. Fuzzy logic.
QA76.545.A38 1997
005'.74--dc21 96-16276
CIP

Contents

Preface

Motivation

Modern enterprises rely on database management systems (DBMS) to collect, store and manage corporate data, which is considered a *strategic corporate resource*. Recently, with the proliferation of personal computers and departmental computing, the trend has been towards the decentralization and distribution of the computing infrastructure, with autonomy and responsibility for data now residing at the departmental and workgroup level of the organization.

Users want their data delivered to their desktops, allowing them to incorporate data into their personal databases, spreadsheets, word processing documents, and most importantly, into their daily tasks and activities. They want to be able to share their information while retaining control over its access and distribution.

There are also pressures from corporate leaders who wish to use information technology as a strategic resource in offering specialized value-added services to customers. Database technology is being used to manage the data associated with corporate processes and activities. Increasingly, the data being managed are not simply formatted tables in relational databases, but all types of objects, including unstructured text, images, audio, and video. Thus, the database management providers are being asked to extend the capabilities of DBMS to include object-relational models as well as full object-oriented database management systems. Corporations are also using the World Wide Web and the Internet to distribute information, conduct electronic commerce, and form virtual corporations where services are provided by a collection of companies, each specializing in a certain portion of the market. This implies that organizations will form federations in which they will share information for the good of the virtual enterprise.

Rather than viewing a database as a passive repository of information, users, managers, and database system providers want to endow databases with *active* properties, so that corporate databases can become an *active participants* in corporate processes, activities and workflows. Thus, there is a real need for active databases that can deliver timely information to users based on their needs, as expressed in profiles and subscriptions. Further, active databases must deal with important events and critical conditions in real-time, that is, as they happen, and take appropriate actions to ensure the correctness and quality of data. Finally, organizations are extracting historical data from on-line transaction processing databases, loading it into data warehouses for on-line analytical processing, and mining it for important patterns and knowledge. These patterns drive decision-making processes to improve corporate workflow, enhance customer satisfaction, and attain competitive advantage.

Clearly, the trends discussed above pose new requirements and challenges for the design and implementation of next-generation database management systems. For example, we cannot rely on traditional transaction processing models with their stringent locking protocols because many corporate activities require support for long-running transactions. In federated systems one cannot impose a processing protocol on a federation partner, rather one must rely on negotiated contracts and commitments for specified levels of service.

New workflow models are required to define computer- and database-supported activities to cooperate in the integration and sharing of information among functional units in the organization. The reengineering of processes and activities can benefit from these new workflow models. These concepts may find their way into the new database management systems or into "middle-ware" products that work in conjunction with the DBMS.

Advanced Transaction Models and Architectures

It is in the context of evolving requirements, uses and expectations for database management systems that we have assembled this important collection of papers authored by world-renowned thinkers, designers and implementors of database systems to address the issues associated with advanced transaction models and architectures. The issues discussed in the book include: 1) workflow models, 2) new transaction models, protocols and architectures, 3) semantic decomposition of transactions, 4) distributed processing, 5) real-time transaction processing, 6) active databases, and 6) new concurrency models for transactional workflows.

We have divided the book into sections and have grouped the papers into topic areas. Part I deals with Workflow Transactions. D. Worah and A. Sheth discuss the role of transactions in workflows, including such topics as recovery and error handling for long-running workflows. G. Alonso and C. Mohan address architectures for workflow management systems, and discuss the challenges facing designers of such systems.

Part II deals with tool-kit approaches to transaction processing. R. Barga and C. Pu present a Reflective Transaction Framework for implementing advanced transaction models as well as semantics-based concurrency control. L. Mancini, I. Ray, S. Jajodia and E. Bertino address flexible commit protocols and show how a general framework can address specific issues such as sagas, workflows, long-lived activities and transactions, and transaction dependencies.

Part III addresses semantic issues associated with transactions, specifically within the context of the ConTracts Model, and also in the semantic decomposition of transactions. A. Reuter, K. Schneider and F. Schwenkreis provide a survey of the ConTracts model, and show how it can be used for handling workflows and properties dealing with the correctness of long-running transactions. P. Ammann, S. Jajodia, and I. Ray focus on the semantics-based decomposition of transactions, introduce concepts such as compensating steps and semantic histories, and prove useful properties of valid decompositions and the processing of such decomposed transactions.

Part IV deals with concurrency control and recovery of transactions. L. Daynès, M. Atkinson, and P. Valduriez discuss how one can customize concurrency control for "persistent" Java. They present a programming model and a transaction shell to support user trade-off analysis and design decisions. C. Martin and K. Ramamritham provide a formal model for recovery of advanced transactions. They couch their model in the form of requirements, assurances and rules to ensure failure atomicity, transaction durability, and recovery. They discuss the model and framework within the context of the ARIES and ARIES/RH recovery protocols.

Part V focuses on transaction optimization techniques. A. Helal, Y-S. Kim, M. Nodine, A. Elmagarmid, and A. Heddaya discuss the failings of current architectures, propose a novel approach based on pre- and post-optimization, and discuss the role of query optimization as it relates to query decomposition, site assignment and replication strategies.

Part VI discusses how the Event-Condition-Action (ECA) paradigm from active databases can be used to implement transaction models. A. Anwar, S. Chakravarthy, M. Viveros present this approach within the Zeitgeist object-oriented database management system.

Part VII discusses the role of inter- and intra-transaction parallelism in the context of both on-line transaction processing (OLTP) and on-line analytical processing (OLAP). C. Hasse and G. Weikum present these concepts within the framework of the PLENTY system which supports both kinds of transaction processing. This is quite different from the current approach in which OLAP is done separately in a data warehouse which is constructed by extracting data from corporate on-line transaction processing systems.

Part VIII is devoted to Real-Time Data Management and P. Jensen, N. Soparkar and M. Tayara discuss real-time concurrency and coordination control in the context of distributed systems.

Part IX completes our collection with a focus on Mobile Computing. J. Shanmugasundaram, A. Nithrakashyap, J. Padhye, R. Sivasankaran, M. Xiong, and K. Ramamritham discuss transaction models in the context of mobile systems in which low bandwidth, low storage capacity and insufficient power impose new challenges for client-server communication and transactions.

We would like to extend our sincerest thanks to Mr. Indrajit Ray who assisted with every aspect of preparing this book, from collection of manuscripts from the authors to dealing with the Kluwer staff regarding LaTeX-related issues. Thanks are also due to our publisher, Mr. Alex Greene, whose enthusiasm and support for this project was most helpful.

SUSHIL JAJODIA AND LARRY KERSCHBERG

This book is dedicated to our loving wives Kamal and Nicole

I Workflow Transactions

1 TRANSACTIONS IN TRANSACTIONAL WORKFLOWS

Devashish Worah and Amit Sheth

Abstract: Workflow management systems (WFMSs) are finding wide applicability in small and large organizational settings. Advanced transaction models (ATMs) focus on maintaining data consistency and have provided solutions to many problems such as correctness, consistency, and reliability in transaction processing and database management environments. While such concepts have yet to be solved in the domain of workflow systems, database researchers have proposed to use, or attempted to use ATMs to model workflows. In this paper we survey the work done in the area of transactional workflow systems. We then argue that *workflow requirements in large-scale enterprise-wide applications involving heterogeneous and distributed environments either differ or exceed the modeling and functionality support provided by ATMs. We propose that an ATM is unlikely to provide the primary basis for modeling of workflow applications, and subsequently workflow management.* We discuss a framework for error handling and recovery in the METEOR$_2$ WFMS that borrows from relevant work in ATMs, distributed systems, software engineering, and organizational sciences. We have also presented various connotations of *transactions* in real-world organizational processes today. Finally, we point out the need for looking beyond ATMs and using a multi-disciplinary approach for modeling large-scale workflow applications of the future.

1.1 INTRODUCTION

A *workflow* is an activity involving the coordinated execution of multiple *tasks* performed by different processing entities [Krishnakumar and Sheth, 1995]. A *workflow process* is an automated organizational process involving both human and automated tasks. *Workflow management* is the automated coordination, control and communication of work as is required to satisfy workflow processes [Sheth et al., 1996a]. There has been a growing acceptance of workflow technology in numerous application domains such as telecommunications, software engineering, manufacturing, production, finance and banking, health care, shipping and office automation [Smith, 1993, Joosten et al., 1994, Georgakopoulos et al., 1995, Fischer, 1995, Tang and Veijalainen, 1995, Sheth et al., 1996b, Palaniswami et al., 1996, Bonner et al., 1996, Perry et al., 1996]. Workflow Management Systems (WFMSs) are being used in inter- and intra-enterprise environments to re-engineer, streamline, automate, and track organizational processes involving humans and automated information systems.

In spite of the proliferation of commercial products for workflow management (including modeling and system supported enactment), workflow technology is relatively immature to be able to address the myriad complexities associated with real-world applications. The current state-of-the-art is dictated by the commercial market which is focused toward providing automation within the office environment with emphasis on coordinating human activities, and facilitating document routing, imaging, and reporting. However, the requirements for workflows in large-scale multi-system applications executing in heterogeneous, autonomous, distributed (HAD) environments involving multiple communication paradigms, humans and legacy application systems far exceeds the capabilities provided by products today [Sheth, 1995].

Some of the apparent weaknesses of workflow models that need to be addressed by the workflow community include the lack of a clear theoretical basis, undefined correctness criteria, limited support for synchronization of concurrent workflows, lack of interoperability, scalability and availability, and lack of support for reliability in the presence of failures and exceptions [Breitbart et al., 1993, Jin et al., 1993, Georgakopoulos et al., 1995, Mohan et al., 1995, Alonso and Schek, 1996b, Kamath and Ramamritham, 1996a, Leymann et al., 1996, Alonso et al., 1996a]. In addition, a successful workflow-enabled solution should address many of the growing user needs that have resulted from:

- emerging and maturing infrastructure technologies and standards for distributed computing such as the World Wide Web, Common Object Request Broker Architecture [OMG, 1995b], Distributed Common Object Model (DCOM), ActiveX, Lotus Notes, and Java.

- increasing need for electronic commerce using standard protocols such as Electronic Data Interchange (EDI) (e.g., ANSI X.12 and HL7),

- additional organizational requirements in terms of security and authentication,

- demands for integrated collaboration (not just coordination) support,

- increasing use of heterogeneous multimedia data, and
- requirements to support dynamic workflows to respond to the fast changing environment (e.g., defense planning), or for supporting today's dynamic and virtual enterprises.

Workflow technology has emerged as a multi-disciplinary field with significant contributions from the areas of software engineering, software process management, database management, and distributed systems [Sheth et al., 1996a]. In spite of the standardization efforts of the Workflow Management Coalition [Coalition, 1994], a consensus on many broader aspects have not yet been achieved.

Work in the areas of transaction processing [Gray and Reuter, 1993] and database systems, and many (but not all) efforts related to ATMs [Elmagarmid, 1992, Chrysanthis and Ramamritham, 1991, Georgakopoulos et al., 1994], are based on a strong theoretical basis. They have proposed or documented solutions (although many of which have yet to be implemented) to problems such as correctness, consistency, and recovery when the constituent tasks are transactional, or the processing entities provide a transactional interface. There exists a strong school of thought, primarily comprised of researchers from the database community, which views a workflow model as an extension of ATMs [Georgakopoulos and Hornick, 1994, Georgakopoulos et al., 1994, Chen and Dayal, 1996, Biliris et al., 1994, Weikum, 1993, Waechter and Reuter, 1992]. However, it has also been observed [Breitbart et al., 1993, Alonso et al., 1996b, Worah and Sheth, 1996] that ATMs have limited applicability in the context of workflows due to their inability to model the rich requirements of today's organizational processes adequately.

Traditional database transactions provide properties such as failure atomicity and concurrency control. These are very useful concepts that could be applicable in workflows. For example, failure atomicity can be supported for a task that interacts with a DBMS, or a group of tasks using the two-phase commit protocol. There is a potential need for concurrency control and synchronization of workflow processes for addressing correctness concerns during workflow execution [Jin et al., 1993, Alonso et al., 1996a]. Based on our review of requirements of existing applications using workflows [Worah and Sheth, 1996], we feel that transactional features form only a small part of a large-scale workflow application. Workflow requirements either exceed, or significantly differ from those of ATMs in terms of modeling, coordination and run-time requirements. It would definitely be useful to incorporate transactional semantics such as recovery, relaxed atomicity and isolation to ensure reliable workflow executions. Nevertheless, to view a workflow as an ATM, or to use existing ATMs to completely model workflows would be inappropriate. We do not think that existing ATMs provide a comprehensive or sufficient framework for modeling large-scale enterprise-wide workflows.

Our observations in this chapter reflect our experience in modeling and development efforts for a real-world workflow application for immunization tracking [Sheth et al., 1996b, Palaniswami et al., 1996], experience in trying to use flexible transactions in multi-system telecommunication applications [Ansari et al., 1992], and our understanding of the current state of the workflow technology and its real-world or realistic applications [Sheth et al., 1996b, Medina-Mora and Cartron, 1996, Bonner et al., 1996, Ansari et al., 1992, Vivier et al., 1996, Sheth and Joosten, 1996].

We emphasize the need for looking beyond the framework of ATMs for modeling and executing workflow applications. The term *transaction* as it is used in business processes today has multiple connotations, database transactions being only one of them. For example, EDI transactions are used for defining interfaces and data formats for exchange of data between organizations and Health Level 7 (HL7) transactions are used to transfer patient data between health care organizations. We discuss other uses of this term in section 1.7. Workflow systems should evolve with the needs of the business and scientific user communities, both in terms of modeling and run-time support. Of course, it is possible that in some specific domains, ATM based workflow models may be sufficient, however, we believe, such cases would be very few.

The organization of this chapter is as follows. Sections 2 through 5 are tutorial in nature. In section 2 we review the research in the domain of ATMs. The next section discusses the characteristics of transactional workflows and significant research in this area. One of the primary focus of transactional workflows is recovery. In section 4 we highlight the issues involved in recovery for workflow systems. Section 5 discusses the types of errors that could occur during workflow execution. In section 6 we discuss a practical implementation of error handling and recovery in a large-scale WFMS. Section 6 provides a perspective into the characteristics and interpretation of *transactions* as they exist in workflow applications today. Finally, we conclude the paper with our observations regarding the role of transactions in transactional workflows.

1.2 ADVANCED TRANSACTION MODELS

In this section we will briefly describe some of the ATMs discussed in the literature [Gray and Reuter, 1993, Elmagarmid, 1992]. These models can be classified according to various characteristics that include transaction structure, intra-transaction concurrency, execution dependencies, visibility, durability, isolation requirements, and failure atomicity. ATMs permit grouping of their operations into hierarchical structures, and in most cases relax (some of) the ACID requirements of classical transactions. In this section, we discuss some of the ATMs that we feel are relevant in the context of workflow systems.

1.2.1 Nested Transactions

An important step in the evolution of a basic transaction model was the extension of the flat (single level) transaction structure to multi-level structures. A *Nested Transaction* [Moss, 1982] is a set of subtransactions that may recursively contain other subtransactions, thus forming a *transaction tree.* A child transaction may start after its parent has started and a parent transaction may terminate only after all its children terminate. If a parent transaction is aborted, all its children are aborted. However, when a child fails, the parent may choose its own way of recovery, for example the parent may execute another subtransaction that performs an alternative action (a *contingency subtransaction*). Nested transactions provide full isolation at the global level, but they permit increased modularity, finer granularity of failure handling, and a higher degree of intra-transaction concurrency than the traditional transactions.

1.2.2 Open Nested Transactions

Open Nested Transactions [Weikum and Schek, 1992] relax the isolation requirements by making the results of committed subtransactions visible to other concurrently executing nested transactions. This way, a higher degree of concurrency is achieved. To avoid inconsistent use of the results of committed subtransactions, only those subtransactions that *commute* with the committed ones are allowed to use their results. Two transactions (or, in general, two operations) are said to commute if their effects, i.e., their output and the final state of the database, are the same regardless of the order in which they were executed. In conventional systems, only *read* operations commute. Based on their semantics, however, one can also define update operations as commutative (for example increment operations of a counter).

1.2.3 Sagas

A *Saga* [Garcia-Molina and Salem, 1987] can deal with long-lived transactions. A Saga consists of a set of ACID subtransactions $T_1, \ldots, T_n$ with a predefined order of execution, and a set of *compensating subtransactions* $CT_1, \ldots,$ CT_{n-1}, corresponding to $T_1, \ldots, T_{n-1}$. A saga completes successfully, if the subtransactions $T_1, \ldots, T_n$ have committed. If one of the subtransactions, say T_k, fails, then committed subtransactions $T_1, \ldots, T_{k-1}$ are undone by executing compensating subtransactions $CT_{k-1}, \ldots, CT_1$. Sagas relax the full isolation requirements and increase inter-transaction concurrency. An extension allows the nesting of Sagas [Garcia-Molina et al., 1991]. *Nested Sagas* provide useful mechanisms to structure steps involved within a long running transaction into hierarchical transaction structures. This model promotes a relaxed notion of atomicity whereby forward recovery is used in the form of *compensating transactions* to undo the effects of a failed transaction.

1.2.4 Multi-Level Transactions

Multi-Level Transactions are more generalized versions of nested transactions [Weikum and Schek, 1992, Gray and Reuter, 1993]. Subtransactions of a multi-level transactions can commit and release their resources before the (global) transaction successfully completes and commits. If a global transaction aborts, its failure atomicity may require that the effects of already committed subtransactions be undone by executing *compensating subtransactions*. A compensating subtransaction t^- semantically *undoes* effects of a committed subtransaction t, so that the state of the database before and after executing a sequence $t\ t^-$ is the same. However, an inconsistency may occur if other transaction s observe the effects of subtransactions that will be compensated later [Gray and Reuter, 1993, Garcia-Molina and Salem, 1987, Korth et al., 1990b]. Open nested transactions use the commutativity to solve this problem. Since only subtransactions that commute with the committed ones are allowed to access the results, the execution sequence $t\ s\ t^-$ is equivalent to $s\ t\ t^-$ and, according to definition of compensation, to s, and therefore is consistent. A somewhat more general solution in the form of a *horizon of compensation*, has been proposed in [Krychniak et al., 1996] in the context of multi-level activities.

1.2.5 Flexible Transactions

Flexible Transactions [Elmagarmid et al., 1990, Zhang et al., 1994a] have been proposed as a transaction model suitable for a multidatabase environment. A flexible transaction is a set of tasks, with a set of functionally equivalent subtransactions for each and a set of execution dependencies on the subtransactions, including failure dependencies, success dependencies, or external dependencies. To relax the isolation requirements, flexible transactions use compensation and relax global atomicity requirements by allowing the transaction designer to specify acceptable states for termination of the flexible transaction, in which some subtransactions may be aborted. IPL [Chen et al., 1993] is a language proposed for the specification of flexible transactions with user-defined atomicity and isolation. It includes features of traditional programming languages, such as type specification to support specific data formats that are accepted or produced by subtransactions executing on different software systems, and preference descriptors with logical and algebraic formulae used for controlling commitments of transactions. Because flexible transactions share some more of the features of a workflow model, it was perhaps the first ATM to have been tried to prototype workflow applications [Ansari et al., 1992].

1.2.6 ACTA and its derivatives

Reasoning about various transaction models can be simplified using the *ACTA metamodel* [Chrysanthis and Ramamritham, 1992]. ACTA captures the important characteristics of transaction models and can be used to decide whether a particular transaction execution history obeys a given set of dependencies. However, defining a transaction with a particular set of properties and assuring that an execution history will preserve these properties remains a difficult problem.

In [Biliris et al., 1994], the authors propose a relaxed transaction facility called *ASSET.* It is based on transaction primitives derived from the ACTA framework that can be used at a programming level to specify customized, application specific transaction models that allow cooperation and interaction. The transaction primitives include a basic and an extended set of constructs that can be used in an application that needs to support custom transactional semantics at the application level. These can be used to support very limited forms of workflows that involve transaction-like components. In some sense, this demonstrates the limitations one may face when trying to use an ATM as a primary basis for workflow modeling.

1.3 TRANSACTIONAL WORKFLOWS

The term *transactional workflows* [Sheth and Rusinkiewicz, 1993] was introduced to clearly recognize the relevance of transactions to workflows. It has been subsequently used by a number of researchers [Breitbart et al., 1993, Rusinkiewicz and Sheth, 1995, Krishnakumar and Sheth, 1995, Georgakopoulos et al., 1995, Tang and Veijalainen, 1995, Leymann et al., 1996]. Transactional workflows involve the coordinated execution of multiple related tasks that require access to HAD systems and support selective use of transactional properties for individual tasks or entire workflows. They use ATMs to specify workflow correctness, data-consistency and reliability. Transactional workflows provide functionality required by each workflow process (e.g., allow task collaboration and support the workflow structure) which is usually not available in typical DBMS and TP-monitor transactions. Furthermore, they address issues related to reliable execution of workflows (both single and multiple) in the presence of concurrency and failures.

Transactional workflows do not imply that workflows are similar or equivalent to database transactions, or support all the ACID transaction properties. They might not strictly support some of the important transaction features supported by TP monitors (e.g., concurrency control, backward recovery, and consistency of data). Nevertheless, such workflows share the objectives of some of the ATMs in terms of being able to enforce relaxed transaction semantics to a set of activities.

In a somewhat conservative view, transactional workflows are workflows supported by an ATM that defines workflow correctness and reliability criteria [Georgakopoulos et al., 1995]. In such a workflow, the tasks are mapped to constituent transactions of an advanced transaction supported by an ATM

[Georgakopoulos et al., 1994], and control flow is defined as dependencies between transactional steps. Similarly, in [Weikum, 1993] an extra *control* layer in terms of dependencies is added to ATM to provide functionality to the transactions running in a large-scale distributed information systems environment.

A WFMS may provide transactional properties to support forward recovery, and use system and application semantics to support semantic based *correct* multi-system application execution [Sheth, 1995, Krishnakumar and Sheth, 1995]. These could include transaction management techniques such as logging, compensation, etc. to enable forward recovery and failure atomicity. In addition, the workflow could exhibit transactional properties for parts of its execution. It might use transaction management technology such as transactional-RPC between two components of a WFMS (e.g, scheduler and task manager), an extended commit coordinator [Miller et al., 1996], or a transactional protocol (XA) between a task manager and a processing entity.

In our view, the scope of transactional workflows extends beyond the purview of database transactions and ATMs. Workflow executions include tasks that might involve database transactions; however, large-scale workflow applications typically extend beyond the data-centric domains of databases and infrastructures that inherently support transaction semantics (e.g., TP-monitors), to more heterogeneous, distributed and non-transactional execution environments.

1.3.1 Previous Research on using Transactions for Workflows

Two major approaches have been used to study and define transactional workflows. The first one utilize a workflow model that is based on supporting organizational processes (also called business process modeling) as its basis, and complements it with transactional features to add reliability, consistency, and other transaction semantics. In the second approach, ATMs are enhanced to incorporate workflow related concepts to increase functionality and applicability in real-world settings. The degree to which each of the models incorporates transactional features varies, and depends largely on the requirements (such as flexibility, atomicity and isolation of individual task executions and multiple workflow instances, etc.) of the organizational processes it tries to model. In the remainder of this section, we discuss some of the research that has been done using ATMs and workflows.

ConTracts [Waechter and Reuter, 1992] were proposed as a mechanism for grouping transactions into a multitransaction activity. A ConTract consists of a set of predefined actions (with ACID properties) called *steps*, and an explicitly specified execution plan called a *script*. An execution of a ConTract must be *forward-recoverable*, that is, in the case of a failure the state of the ConTract must be restored and its execution may continue. In addition to the relaxed isolation, ConTracts provide relaxed atomicity so that a ConTract may be interrupted and re-instantiated.

Workflow applications are typically long-lived compared to database transactions. A workflow is seen as a *Long-Running Activity* in [Dayal et al., 1990, Dayal et al., 1991]. A Long-Running Activity is modeled as a set of execution units that may consist recursively of other activities, or top-level transactions (i.e., transactions that may spawn nested transactions). Control flow and data flow of an activity may be specified statically in the activity's *script*, or dynamically by Event-Condition-Action (ECA) rules. This model includes compensation, communication between execution units, querying the status of an activity, and exception handling.

Motivated by advanced application requirements, several ATMs have been proposed (refer to [Chrysanthis and Ramamritham, 1991, Georgakopoulos and Hornick, 1994] for frameworks for defining and comparing ATMs, [Elmagarmid, 1992] for several representative ATMs, for a representative model and specification that support application specific transaction properties, and [Breitbart et al., 1993, Hsu, 1993, Rusinkiewicz and Sheth, 1995] for earlier views on relationships between workflows and ATMs). ATMs extend the traditional (ACID) transaction model typically supported by DBMSs to allow advanced application functionality (e.g., permit task collaboration and coordination as it is required by ad hoc workflows) and improve throughput (e.g., reduce transaction blocking and abortion caused by transaction synchronization). However, many of these extensions have resulted in application-specific ATMs that offer adequate correctness guarantees in a particular application, but not in others. Furthermore, an ATM may impose restrictions that are unacceptable in one application, yet required by another. If no existing ATM satisfies the requirements of an application, a new ATM is defined to do so.

In [Georgakopoulos et al., 1994], the authors define an extended (advanced) transaction framework for execution of workflows called the *Transaction Specification and Management Environment* (TSME). A workflow in this framework consists of constituent transactions corresponding to workflow tasks. In addition, workflows have an execution structure that is defined by an ATM; the ATM defines the correctness criteria for the workflow. The TSME claims to support various ATMs (extended transaction models) to ensure correctness and reliability of various types of workflow processes. Extended transactions consist of a set of constituent transactions and a set of dependencies between them. These transaction dependencies specify the transaction execution structures or correctness criteria. A programmable transaction management mechanism based on the ECA rules [Dayal et al., 1990] is used to enforce transaction state dependencies.

Semantic transaction models aim to improve performance and data consistency by executing a group of interacting steps within a single transaction and relaxing the ACID properties of this transaction in a controlled manner. In [Weikum, 1993], the author suggests that semantic transaction concepts be merged with workflow concepts to promote workflow systems that are consistent and reliable. The author defines a transactional workflow to be a control sphere that binds these transactions by using dependencies to enforce as much behavioral consistency as possible thereby enforcing reasonable amount of data consistency.

The *METEOR*[1] [Krishnakumar and Sheth, 1995] workflow model is an integration of many of the approaches described above. A workflow in METEOR is a collection of multiple tasks. Each of the tasks could be heterogeneous in nature. The execution behavior of the tasks are captured using well-defined task structures. This model supports tasks that have both transactional and non-transactional semantics. Groups of tasks along with their inter-task dependencies can be modeled as compound tasks. The compound tasks have their task structures too. Transactional workflows can be modeled using transactional tasks and transactional compound tasks as the basis of the workflow model. The METEOR_2 WFMS [Miller et al., 1996, Sheth et al., 1996b] is based on the METEOR model. It extends the model in terms of providing better support for failure recovery and error handling in heterogeneous and distributed workflow environments (see section 1.6.1 for additional details).

The *Exotica* project [Alonso et al., 1995a, Alonso et al., 1996b] explores the role of advanced transaction management concepts in the context of workflows. A stated objective of this research is to develop workflow systems that are capable enough (in terms of reliability, scalability, and availability) to deal with very large, heterogeneous, distributed and legacy applications. One of the directions of this project is to research the synergy between workflow systems and advanced transaction models; the results that follow point in the direction that workflow systems are a superset of advanced transaction models [Alonso et al., 1996b] since workflow systems incorporate process and user oriented concepts that are beyond the purview of most ATMs. Partial backward recovery has been addressed in the context of the FlowMark WFMS [Leymann, 1995] by generalizing the transactional notions of compensation.

One of the projects in which transactional semantics have been applied to a group of steps define a logical construct called a *Consistency unit* (C-unit) [Tang and Veijalainen, 1995]. A C-unit is a collection of workflow steps and enforced dependencies between them. C-units relax the isolation and atomicity properties of transactional models. The authors also discuss how C-units can be used to develop transactional workflows that guarantee correctness of data in the view of integrity constraints that might exist across workflow processing entities.

The *INformation CArrier* (INCA) [Barbara et al., 1996a] workflow model was proposed as a basis for developing dynamic workflows in distributed environments where the processing entities are relatively autonomous in nature. In this model, the INCA is an object that is associated with each workflow and encapsulates workflow data, history and processing rules. The transactional semantics of INCA procedures (or steps) are limited by the transaction support guaranteed by the underlying processing entity. The INCA itself is neither atomic nor isolated in the traditional sense of the terms. However, transactional and extended transactional concepts such as *redo of steps*, *compensating steps*and *contingency steps* have been included in the INCA rules to account for failures and forward recovery.

In the *Nested Process Management* environment [Chen and Dayal, 1996] a workflow process is defined using a hierarchical collection of transactions. Failure handling is supported using a two-phase approach. During the first phase of recovery, a bottom-up lookup along the task tree is performed to determine the oldest parent transaction that does not need to be compensated. The next phase involves compensation of all the children of this parent. In this model, failure atomicity of the workflow is relaxed in terms of compensating only parts of the workflow hierarchy.

The *Workflow Activity Model*(WAMO) [Eder and Liebhart, 1995] defines a workflow model that enables the workflow designer in modeling reliable workflows [Eder and Liebhart, 1996]. It uses an underlying relaxed transaction model that is characterized by relaxing i) failure atomicity of tasks, ii) serializability of concurrent and interleaved workflow instance executions, and iii) relaxing isolation in terms of externalization of task results.

Thus we see that transaction concepts have been applied to various degrees in the context of workflows. They have been used to define application specific and user-defined correctness, reliability and functional requirements within workflow executions. In the next section, we discuss features specific to transactions and ATMs that would be useful for implementing recovery in a WFMS.

1.4 WORKFLOW RECOVERY

Reliability is of critical importance to workflow systems [Georgakopoulos et al., 1995, Georgakopoulos, 1994, Jin et al., 1993]. WFMS should not only be functionally correct, but should also be robust in the view of failures. Workflow systems (both commercial and research prototypes) in their current state, lack adequate support for handling errors and failures in large-scale, heterogeneous, distributed computing environments [Georgakopoulos et al., 1995, Alonso and Schek, 1996b, Kamath and Ramamritham, 1996a, Sheth et al., 1996a, Leymann et al., 1996]. Failures could occur at various points and stages within the lifetime of the workflow enactment process. They could involve failures associated with the workflow tasks (such as unavailability of resources, incorrect input formats, internal application failures, etc.), failures within the workflow system components (such as schedulers, databases, etc.), and failures in the underlying infrastructure (such as hardware and network failures). Reliability in the context of workflows requires that tasks, their associated data, and the WMFS itself be recoverable in the event of failure, and that a well defined method exists for recovery.

A workflow process is heavily dependent on the organizational structure, and business policies within an organization. Workflows are activities that are *horizontal* in nature and are spread across the organizational spectrum as compared to transaction processing activities (e.g., database transactions) that are more *vertical* or *hierarchical* in nature and might form only part of the workflow process. In other words, hierarchical decomposition used for complex advanced transaction models is not sufficient for modeling workflows. A WFMS needs to support recovery of its tasks, associated data and the workflow process as a whole. The heterogeneous nature of workflow tasks and processing entities might preclude any transactional semantics that are required for assuring transactional behavior of the workflow or the constituent tasks themselves. A viable recovery mechanism should be consistent with and should support the overall goal of the business process concerned.

Valuable research addressing recovery has been done in transaction management and ATMs [Bernstein et al., 1987, Gray and Reuter, 1993, Korth et al., 1990b, Moss, 1987, Waechter and Reuter, 1992, Chen and Dayal, 1996] (see sections 1.2 and 1.3.1). A strictly data-centric approach has been used to address recovery issues in transaction processing. The problem domain of recovery in a WFMS is broader than that of transaction systems and ATMs due to its process-oriented focus, and diverse multi-system execution requirements. Although the ideas proposed in ATMs are limited in terms of the domains and environments they apply to, they are valuable in terms of their semantics and overall objectives. In the next section, we discuss the value and applicability of transaction concepts in the context of workflow recovery.

1.4.1 Transaction Concepts in Modeling Workflow Recovery

Earlier, we have discussed some of the ATMs that have been proposed in the literature. Recovery involves restoration of state - a concept which is voiced by transactional systems also. Later, we also reviewed some of the work in transactional workflows, and different approaches for incorporating transactional semantics into workflow models. We feel that transaction concepts are necessary for a recovery mechanism to be in place; however, basing a workflow recovery framework on a transactional (or advanced) transactional model would be naive.

As discussed in section 1.2.1, the hierarchical model in nested transactions [Moss, 1982] allows finer grained recovery, and provides more flexibility in terms of transaction execution. In addition to database systems, nested transactions can been used to model reliable distributed systems [Moss, 1987]. There is a lot to learn from work done in nested transactions. It provides a model for partitioning an application system into recoverable units; transaction failure is often localized within such models using retries and alternative actions. Workflow systems can borrow these ideas to a great extent, and tasks can be retried in the case of certain failures (e.g., failures related to unavailability of input data, or inadequacy of resources for executing a task at a processing entity), or alternate tasks can be scheduled to handle other more serious errors (e.g., when a certain number of retries fail, or when a task cannot be activated due to unavailability of a processing entity) that might cause a task to fail.

In the work on nested process management systems [Chen and Dayal, 1996] (discussed in section 1.3.1), the authors present a formal model of recovery that utilizes relaxed notions of isolation and atomicity within a nested transaction structure. Although, this model is more relaxed in terms of recovery requirements as compared to nested transactions, it is strict for heterogeneous workflow environments that involve tasks that are non-transactional in nature. Moreover, the recovery model uses backward recovery of some of the child transactions for undoing the effects of a failed global transaction. The backward recovery approach has limited applicability in workflow environments in which it is either not possible to strictly reverse some actions, or is not feasible (from the business perspective) to undo them since this might involve an additional overhead or conflict with a business policy (e.g., in a banking application).

The notion of compensation is important in workflow systems. Undoing of incomplete transactions (or backward recovery) is an accepted repair mechanism for aborted transactions. However, this concept is not directly applicable to most real-world workflow tasks which are governed by actions that are in general permanent (e.g, human actions and legacy system processing). One can define a semantically inverse task (commonly referred to as *compensating tasks*), or a chain of tasks that could effectively undo or repair the damage incurred by a failed task within a workflow. In addition to Sagas, semantic transaction models have been proposed to address many such issues in which failure atomicity requirements have been relaxed. Compensation has been applied to tasks and groups of tasks (*spheres*) to support partial backward recovery in the context of the FlowMark WFMS [Leymann, 1995].

Work on flexible transactions[Elmagarmid et al., 1990, Zhang et al., 1994a] discusses the role of *alternate transactions* that can be executed without sacrificing the atomicity of the overall global transaction. This provides a very flexible and natural model for dealing with failures. These concepts are applicable in workflow environments also. A prototype workflow system that implments a flexible transaction model has been discussed in [Alonso et al., 1996b].

In transactional models, the unit of recovery is a transaction. Each transaction has a predefined set of semantics that are compliant with the transaction processing system. The model for recovery in a workflow system is more involved since the recovery process should not only restore the state of the workflow system, but should proceed forward in a manner that is compliant with the overall organizational process.

Recovery of Workflow Tasks A task (activity or step) forms a basic unit of execution within a workflow model. A task is a logical unit of work that is used to satisfy the requirements of the business process that defines the workflow concerned. In database systems, it is sufficient to maintain *before* and *after* images of the data affected by a transaction to guarantee enough information needed to recover that transaction in case of its failure. Recovery of tasks, therefore, should be addressed from a broader perspective; in addition to focusing on data-centric issues, one must focus on the overall business model associated with the actions within a task.

The tasks within a workflow could be arbitrarily complex and heterogeneous (i.e., transactional and non-transactional) in nature. A workflow model proposed in [Georgakopoulos et al., 1994] compares database transactions to tasks within a workflow, thereby regarding a workflow task to be the unit of recovery. This parallelism is valid when the tasks are relatively simple, obey transactional semantics and are executing within an environment that can enforce the transactional behavior of a group of tasks. Most real-world workflow applications and run-time environments are far more complex in nature and may be spread across arbitrary autonomous systems. Hence, a uniform recovery model based solely on transactional assumptions is inapplicable to commercial workflow systems.

Many task models have been defined for workflow systems [Attie et al., 1993, Krishnakumar and Sheth, 1995, Rusinkiewicz and Sheth, 1995]. In spite of this fact, it is difficult to determine the exact execution state of a task since these task models do not model detailed task execution. One could implement a workflow system involving special tasks that reveal their internal state to the WFMS layer; however, this workflow solution is not general enough to handle tasks that are diverse and arbitrarily complex in nature. Guaranteeing strict failure atomicity akin to that in database transactions is therefore difficult for workflow tasks. Hence, recovery of tasks should be addressed from a broader perspective. One should focus on the overall business process model when trying to decide the next action to be performed when resolving task failures.

In the case of non-transactional tasks, it is difficult to monitor the exact state of the task once it has been submitted for execution. This lack of control could leave the system in an undeterministic state in view of failures. In such a scenario, automatic recovery of a failed task becomes impossible due to lack of run-time feedback or transactional guarantees from the processing entities. The role of the human (e.g., workflow administrator) is important for recovery in such situations for determining the state of the failed task based on information that is external to the workflow system. In the METEOR$_2$ system [Worah, 1997], a special task is used to *cleanup* the remnants of such failures and to restore the workflow system to a consistent state. It could involve the role of a human or an application that is programmed to be able to reconfigure the data and applications associated with a task to restore it to a consistent state.

Recovery of Workflow Data Data plays an important role in workflow systems, as is in the case of a DBMS and a TP-system. Data recovery issues have been studied extensively in the context of database systems. *Logging* and *shadow paging* are common mechanisms used in transaction processing to record state of critical data persistently. Several *checkpointing* mechanisms have been discussed in literature [Bernstein et al., 1987] to enhance the performance of the recovery process. These principles can be applied to workflow systems in situations related to making the state of the workflow components persistent and the recovery process more efficient. In the case of distributed WFMSs, it is also important to replicate data across machines to enhance data availability in the view of hardware and network failures. This problem, once again, has been studied extensively in the area of distributed databases; its applicability has also been studied in workflow systems [Alonso et al., 1995b] to enhance their availability.

1.5 WORKFLOW ERROR HANDLING

Error handling is another critical area of workflow research that has not received adequate attention [Georgakopoulos et al., 1995, Alonso and Schek, 1996b]. The cause of errors in workflow systems could be multifarious. Errors are logical in nature; they could be caused due to failures within the workflow system, or failures occurring at the task level.

Error handling in database systems has typically been achieved by aborting transactions that result in an error [Gray and Reuter, 1993]. Aborting or canceling a workflow task, would not always be appropriate or necessary in a workflow environment. Tasks could encapsulate more operations than a database transaction, or the nature of the business process could be forgiving to the error thereby not requiring an undo operation. Therefore, the error handling semantics of traditional transactional processing systems are too rigid for workflow systems.

A mechanism for dealing with errors in an ATM for long running activities was proposed in [Dayal et al., 1990, Dayal et al., 1991]. It supported forward error recovery, so that errors occurring in non-fatal transactions could be overcome by executing alternative transactions. Although, this model provides well defined constructs for defining alternative flow of execution in the event of errors, it is restrictive in terms of the types of activities (relaxed transactions) and the operating environment (a database) that form the long running process and therefore, it does not provide the error modeling capabilities of capturing workflow errors.

We can characterize the types of errors arising in a WFMS into three *broad* categories:

- *Infrastructure errors*: these errors result from the malfunctioning of the underlying infrastructure that supports the WFMS. These include communication errors such as loss of information, and hardware errors such as computer system crashes and network partitioning.

- *System errors*: these errors result from faults within the WFMS software. This could be caused due to faults in the hardware, or operating system. An example is the crash of a workflow scheduler.

- *Application and user errors*: these errors are closely tied to each of the tasks, or groups of tasks within the workflow. Due to its dependency on application level semantics, these errors are also termed as *logical* errors [Krishnakumar and Sheth, 1995]. For example, one such error could involve database login errors that might be returned to a workflow task that tries to execute a transaction without having permission to do so at a particular DBMS. A failure in enforcing inter-task dependencies between tasks is another example of an application error.

The above categorization is a descriptive model for categorizing errors within WFMSs. Large-scale WFMSs typically span across heterogeneous operating environments; each task could be arbitrarily complex in nature. To be able to detect and handle errors in such a diverse environment, we need a well-defined error model that would help us specify, detect and handle the errors in a systematic fashion. In 1.6.1.3 we define a hierarchical error model that forms the basis for handling errors in the METEOR$_2$ WFMS.

In the previous sections, we have discussed research done in the area of ATMs, transactional workflows, and the problem of error handling and recovery in WFMSs. In the next section we outline issues that are important for implementing a reliable WFMS. In doing so, we discuss a specific example of a WFMS that exploits many of the concepts from transactional systems and ATMs to include support for error handling and recovery.

1.6 TRANSACTIONS, ATMS AND RECOVERY IN LARGE-SCALE WFMSS

Pervasive network connectivity, coupled with the explosive growth of the Internet has changed our computational landscape. Centralized, homogeneous, and desktop-oriented technologies have given way to distributed, heterogeneous and network-centric ones. Workflow systems are no exceptions. They would typically be required to operate in such diverse environments in a *reliable* manner. Implementation of error handling and recovery in a WFMS is affected by numerous factors ranging from the underlying infrastructure (e.g., DBMS, TP-monitor, Lotus Notes, CORBA, Web), architecture of the supporting framework (e.g., centralized vs. distributed), nature of the processing entities (e.g., open vs. closed, transactional vs. non-transactional, human vs. computer system), type of tasks (user vs. system, transactional vs. non-transactional), and the nature of the workflow application (e.g., ad-hoc vs. administrative vs. production). Most of these issues are beyond the purview of transaction-based systems, and therefore have not been adequately tackled by them.

A single recovery mechanism cannot be applied to all workflow applications due to the diversity of their business logic. Also, the variations in WFMS run-time architectures and execution environments would dictate the choice of suitable recovery mechanisms. A workflow is a collection of tasks; the tasks could be arbitrary in nature. It is impossible to include task specific semantics within a generalized recovery framework since task behavior is orthogonal to that of the workflow process. Nevertheless, a WFMS should provide the necessary infrastructure to support error handling and recovery as needed by the task. It should also provide tools to allow users to specify failure handling semantics that are conformant with the governing business process model. This is an important characteristic that differentiates failure handling in workflow systems from that in transaction processing where it suffices to satisfy the ACID properties for transactions.

ATMs provide techniques for handling failures (see Section 1.2). However, most of these ATMs do not discuss any aspects of implementation. Implementation of processes in workflow systems require support for business level *details* such as groups, roles, policies, etc. ATMs are weak in this aspect, since they define models that are focused towards the tasks themselves (in this case advanced transactions). Therefore, workflow systems are implemented at a higher level of granularity than ATMs. In fact, in [Alonso et al., 1996b] *sagas* and *flexible transactions* have been implemented using a WFMS.

WFMSs in distributed environments are dependent on inter-process communication across possibly heterogeneous computing infrastructures. In such systems, it is important that communication between processes is reliable. Transactional RPC mechanisms have been used in distributed transaction processing to guarantee reliable messaging between distributed processes. They can also be incorporated into workflow systems [Wodtke et al., 1996] to guarantee transactional messaging between the workflow components thereby increasing the level of fault-tolerance of the WFMS infrastructure.

TP-monitors have been used extensively to guarantee transactional semantics across distributed process spaces. They are, therefore, a viable middleware technology for implementing workflow systems. However, their use within a workflow environment comes with a lot of cost: 1) it is not feasible to impose infrastructural homogeneity (e.g., use of TP-monitors) across autonomous organizations, and 2) it is very expensive to maintain and administer especially when workflow process span multiple organizations. Emerging infrastructure technologies such as Web, CORBA, and DCOM, on the other hand, provide more open and cost effective solutions for implementing large-scale distributed workflow applications [Sheth et al., 1996b, Palaniswami et al., 1996]. In particular, the CORBA standard [OMG, 1995b] includes specifications for services [OMG, 1995a] such as the Object Transaction Service (OTS), the Concurrency Control Service, and the Persistence Service that can be combined to form a framework for achieving TP-monitor-like functionality in a HAD environments.

1.6.1 Error Handling and Recovery in the METEOR$_2$ WFMS

The study of workflow systems is inter-disciplinary, and stems from areas such as distributed systems, database management, software process management, software engineering, and organizational sciences [Sheth et al., 1996a]. Error handling and recovery are equally critical in these domains, and numerous solutions have been suggested to address these problems [Bhargava, 1987, Bernstein et al., 1987, Cristian, 1991, Saastamoinen, 1995].

In this section, we present an error handling and recovery framework that we have implemented for the distributed run-time of the METEOR$_2$ WFMS. This solution has been based on principles and implementation ideas that we have borrowed from related research in databases, advanced transaction models, software engineering and distributed systems. Due to lack of space, brevity is key in our discussions (for additional details, see [Worah, 1997]).

1.6.1.1 Overview of METEOR$_2$ Workflow Model. The METEOR$_2$ workflow model is an extension of the METEOR [Krishnakumar and Sheth, 1995] model, and is focused towards supporting large-scale multi-system workflow applications in heterogeneous and distributed operating environments. The primary components of the workflow model include 1) processing entities and their interfaces, 2) tasks, 3) task managers, and 4) the workflow scheduler.

- *Processing Entity*: A processing entity is any user, application system, computing device, or a combination thereof that is responsible for completion of a task during workflow execution. Examples of processing entities include word processors, DBMSs, script interpreters, image processing systems, auto-dialers, or humans that could in turn be using application software for performing their tasks.

- *Interface*: The interface denotes the access mechanism that is used by the WFMS to interact with the processing entity. For example, a task that involves a database transaction could be submitted for execution using a command line interface to the DBMS server, or by using an application programming interface from within another application. In the case of a user task that requires user-input for data processing, the interface could be a Web browser containing an HTML form.

- *Task*: A task represents the basic unit of computation within an instance of the workflow enactment process. It could be either transactional or non-transactional in nature. Each of these categories can be further divided based on whether the task is an application, or a user-oriented task. *Application tasks* are typically computer programs or scripts that could be arbitrarily complex in nature. A *user task* involves a human performing certain actions that might entail interaction with a GUI-capable terminal. The human interacts with the workflow process by providing the necessary input for activating a user task. Tasks are modeled in the workflow system using well-defined task structures [Attie et al., 1993, Rusinkiewicz and Sheth, 1995, Krishnakumar and Sheth, 1995] that export the execution semantics of the task to the workflow level. A task structure is modeled as a set of states (e.g., initial, executing, fail, done), and the permissible transitions between those states. Several task structures have been defined - transactional, non-transactional, simple, compound, and two-phase commit [Krishnakumar and Sheth, 1995, Wang, 1995].

- *Task Manager*: A task manager is associated with every task within the workflow execution environment. The task manager acts as an intermediary between the task and the workflow scheduler. It is responsible for making the inputs to the task available in the desired format, for submitting the task for execution at the processing entity, and for collecting the outputs (if any) from the task. In addition, the task manager communicates the status of the task to the workflow scheduler.

- *Workflow Scheduler*: The workflow scheduler is responsible for coordinating the execution of various tasks within a workflow instance by enforcing inter-task dependencies defined by the underlying business process. Various scheduling mechanisms have been designed and implemented [Wang, 1995, Miller et al., 1996, Das, 1997, Palaniswami, 1997], ranging from highly centralized ones in which the scheduler and task managers reside within a single process, to a fully distributed one in which scheduling components are distributed within each of the distributed task manager processes.

We will focus our discussions on a run-time implementation of a distributed architecture for the METEOR$_2$ WFMS. A recovery framework has been defined for this architecture. The basic distributed model has been enhanced with additional functionality to 1) handle various forms of errors, 2) use transaction semantics at run-time, 3) monitor active workflow components, 4) recover failed components, and 5) log critical data that is necessary to restore the state of a failed workflow.

1.6.1.2 ORBWork: A Distributed Implementation of the METEOR$_2$ WFMS. ORBWork is a distributed run-time engine for METEOR$_2$ WFMS. It has been implemented using CORBA [OMG, 1995b] and Web infrastructure technologies [Sheth et al., 1996b, Das, 1997, Worah, 1997]. The former provides the necessary distribution and communication capabilities for the workflow components, and the latter makes it possible for humans to interact with the Object Request Broker (ORB)[2] based workflow layer. The main components of ORBWork are shown in Figure 1.1. In this implementation, task managers, recovery units, data objects, monitors, and clean-up tasks are implemented as CORBA objects.

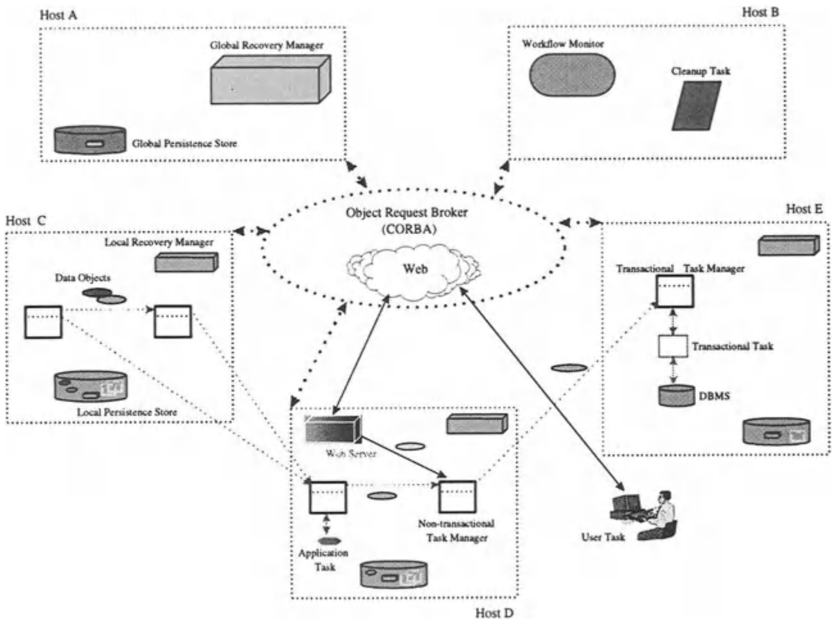


Figure 1.1 System Schematic for the Recovery Framework in ORBWork

In METEOR$_2$, the workflow process that defines the overall organizational process is captured in the form of a *workflow map* that is specified by a workflow designer. This determines the data and control dependencies that need to be enforced as part of the workflow scheduling process. Due to the distributed nature of the workflow engine, ORBWork does not have a *centralized* scheduling entity. The scheduling mechanism is embedded in each of the task managers.

Each task managers performs four primary functions: 1) task activation, 2) error handling and recovery of task and its own errors, 3) logging of task inputs, outputs, and its internal state, and 4) scheduling of dependent task managers as defined by the workflow process. Task managers communicate with each via the ORB using object method invocations. Due to the location transparency offered by CORBA, they are able to communicate seamlessly irrespective of the host they execute on.

Input and output data elements to the tasks are represented as CORBA objects internal to ORBWork. These CORBA objects are *wrappers* around the actual data elements. This allows workflow data objects to be distributed within the ORB environment. Task managers logically enforce workflow data dependencies and pass data by exchanging references to these data objects.

User tasks have associated "to-do" *worklists* (not shown in the figure) that provide a list of pending tasks for the user. User inputs form one of the implicit dependencies for a *user task manager*. User (human) tasks communicate with the task managers using HTML forms and Common Gateway Interface (CGI) functionality provided by Web servers. In our current implementation, CGI scripts are implemented as CORBA clients to user task manager objects. References to CORBA objects that encapsulate the user provided data are passed as inputs to the task manager.

ORBWork is subject to numerous errors and failures. The architecture of ORBWork, as described above, does not provide support for error handling, other than what is already inherent to the components themselves. The distributed nature of our workflow architecture alleviates problems associated with a single point of failure. This allows scope for incorporating fault-tolerant features into the framework. However, distribution adds to the complexity of the system in terms of management of the various components and detection of failures. This problem is compounded due to the asynchronous communication paradigm used in workflow communication models. Moreover, the communication infrastructure is subject to failures, and could adversely affect workflow enactment. In the following two sections, we describe the error model that we have used to capture such errors, and the failure handling components that form our recovery framework. For a detailed discussion on ORBWork, see [Das, 1997].

1.6.1.3 Modeling Errors in METEOR$_2$. The METEOR$_2$ error model has been defined in a hierarchical manner. We have based it on the layered nature of the METEOR$_2$ workflow model. It enables us to describe and classify the various errors that occur during workflow execution. This, in effect, makes it possible to modularize our error handling algorithms during workflow execution. Errors are detected and masked as close to the point of occurrence as possible to prevent them from propagating to other, unrelated components of the WFMS. We use a three-tiered approach to classify errors within the METEOR_2 workflow model:

1. *Task and Workflow Errors*: this class forms the lowest level within our hierarchy and includes all errors that are specific to tasks, and their inter-task dependencies. Application and user errors (as discussed in Section 1.5) are defined and modeled at this level. The workflow designer is responsible for defining these errors during the workflow definition process. The workflow system does not preclude a task from handling its errors on its own; in such cases, only unhandled errors would be categorized as task errors within the WFMS. Some of these errors may have implications on the whole workflow process. A task error that cannot be resolved is eventually reported to its task manager; such an error falls into the category of task manager errors.

2. *Task Manager Errors*: this class of errors involves all task errors that could not be resolved at the task level (as described earlier), and errors that are specific to the task manager itself. For example, the latter includes errors such as

   - not being able to prepare the inputs for the task,
   - not being able to submit a task for execution,
   - not being able to recover the state of task during failure recovery, and
   - not being able to handle a task error that might have occurred.

   A task manager error that remains unhandled is reported as a workflow error to the scheduler.

3. *WFMS Errors*: These are the highest level of errors within our model and include

   - system errors that affect the task scheduling mechanism,
   - communication errors between the scheduler and the task managers,
   - other failures in workflow components that are common to all instances of a workflow type (e.g., failure recovery units, log managers, etc.), and
   - errors that could not be handled at the level of the task manager.

In our model, task and workflow errors are logical in nature. Error handling at this level is achieved by retries, aborts, cancellations, and by trying alternate tasks. Task manager and WFMS errors are system errors caused due to failures within the WFMS software. Task manager errors are either handled at the level of the task manager itself (e.g., retrying task submission for a task that cannot be submitted). WFMS errors are handled by the recovery components within the WFMS, or by a human that would be provided with information necessary to handle the error.

Principles relating to classification of errors, and handling them in a modular fashion have been commonplace in computer architecture, programming languages, and software engineering. We have mapped the ideas to our workflow model, and have defined the error-handling semantics so that they are in synchrony with the overall business process that defines the workflow. Although, this model has been applied within the METEOR$_2$ WFMS, in principle, it is applicable to any workflow model that has a well-defined modular architecture. The error handling capabilities in ORBWork, are developed on the basis of this error model.

1.6.1.4 Recovery Framework in ORBWork. In this section, we describe the recovery framework for ORBWork (see Figure 1.1). In defining the recovery framework, we have extended the ORBWork workflow engine in terms of being able to handle failures ranging from the task level to the level of the workflow system components. The recovery model assumes a distributed, component-based architecture for the WFMS, and a communication mechanism (in this case CORBA) that makes it possible to interact with components across host boundaries.

Persistence is an essential part of our recovery framework. We have used an object-oriented approach wherein the various workflow components are responsible for logging their respective states to stable storage. This approach is very similar to the notion of recoverable objects in the distributed object-oriented framework of *Arjuna*[Shrivastava et al., 1991]. In our model, data objects inherit from a base interface that attributes it with capabilities to *save* and *restore* its state at runtime from stable storage. A *Local Persistence Store* (LPS) is used as the stable storage mechanism for logging local data critical for recovery purposes. We have used a DBMS as the basis for our LPS. A DBMS provides transactional capabilities to log data. A *Global Persistence Store* is used for logging at the level of the GRM. Logging is done at various stages within the workflow enactment process. For example, 1) task Managers log the state of their tasks (including error codes returned by the task, for future debugging and error recovery), inputs that they receive from other task managers, and outputs that they send out to dependant task managers; 2) data objects log the state of their data they encapsulate.

Failures in distributed systems are hard to detect, unless there is a fault-tolerant detection mechanism in place. This problem is compounded especially when most of the components communicate in an asynchronous mode. Distributed workflow systems fall into this category due to their asynchronous coordination model. In ORBWork, we have provided additional services for *monitoring* distributed components, to address the issue of failure detection. In this regard, we have borrowed ideas from other work done in reliable distributed systems [Birman and Renesse, 1994, Maffeis, 1996].

Task Managers and data objects on each host are monitored by a *Local Recovery Manager* (LRM) process executing on the same machine. On startup, the task managers and data objects, *register* with the LRM on their host. Once, these components are no longer required within the workflow process, they *deregister* from the LRM. The LRM maintains a *watch-list* of currently registered components that are supposed to be executing as part of the workflow process instance on its host. When an object registers with the LRM, the LRM logs this message and appends it to the list. On deregistration, these objects are removed from the list. The LRM contains a *watchdog* that periodically, polls each of the components on the watch-list to ensure their liveliness. When a failed component is detected, the LRM reactivates the component, which in turn, restores its own state from local logs. The LRM checkpoints it logical view of the local system to the local log to enable its own recovery. In addition to the LRM, each host contains a daemon process called the *Local Activation Daemon* (LAD) (not shown in the figure) that is endowed with the ability to create processes (for the various CORBA objects) on the various hosts.

A *Global Recovery Manager* (GRM) executing on a reliable host in the workflow execution environment monitors the liveliness of all the LRMs and is responsible for reactivating any failed LRMs. On recovery, the failed LRMs synchronize the state of their respective local systems based on their local logs and create any task managers that might have failed in the interim. Due to the infancy of the CORBA standard, and unavailability of many of its object services, we had to rely on programmatic efforts to implement many of the features that we would have otherwise liked to have been supplied by the ORB vendor. The implementation of error handling is achieved via the use of exceptions and *try-catch* blocks that help to isolate the normal flow of execution from the abnormal case during run-time.

Local *configuration files* (not shown in the figure) are used on each host by the LAD. These files are used for directory lookup for the various components (i.e., task manager, data object, LRM, GRM) during activation or recovery of the processes.

During the definition of the workflow design, it might not be feasible to capture all errors and causes of failures that might occur during the enactment process. Also, especially in the case of non-transactional tasks, it is not always possible to *undo* the effects of a task that might have completed partially. We therefore feel that the role of a human is indispensable within the workflow recovery framework. In our model, we have allocated a special human-performed task, called the *cleanup task* to serve the functionality of bringing the system to a consistent state after such irrecoverable failure. This mode of restoration is used only when the WFMS is unable to handle the recovery process automatically.

Let us summarize the main characteristics of our recovery framework.

- Workflow recovery is implemented in a distributed CORBA and Web based execution environment.
- A notion of hierarchical monitoring of workflow components has been used to detect failures, and to initiate the recovery process (i.e., GRM monitors the LRMs; LRMs monitor task managers and data objects; task managers monitor tasks). This allows failures to be localized, and their effects to be masked as close to the point of occurrence as possible.
- The recovery model ensures that there is no single point of failure. Therefore, the failure of a host does not significantly affect the performance of tasks within another (unless they are directly dependant on each other).
- The performance of the workflow system would degrade *progressively* in the case of failures; however, once the failure has been restored, the WFMS would execute normally.
- Each workflow component is responsible for logging its own state. The persistence mechanism used is also local to the component itself.
- The workflow components are responsible for managing their own recovery actions once they have been *recreated.*
- The recovery mechanism is semi-automated. The role of the human is crucial both during the workflow design process and the enactment. The workflow designer specifies the run-time behavior of the error handling and forward recovery mechanism. The workflow administrator is responsible for fixing drastic system failures (e.g., machine crash, network partitioning), and for *cleanup* of failed tasks that cannot be handled by the WFMS.
- The distribution and hierarchical nature of the recovery mechanism makes the system scaleable and manageable.

In this section we have briefly described the design and implementation of error handling and recovery in the distributed run-time of the METEOR$_2$ WFMS (see [Worah, 1997] for more details). We have used this discussion to illustrate the applicability of concepts and basic mechanisms from traditional and ATMs within a practical workflow execution environment. Also, our discussion is suggestive of the need to look for solutions beyond ATMs for addressing reliability issues in WFMSs.

1.7 TYPES OF TRANSACTIONS IN THE REAL-WORLD: BEYOND DATABASE TRANSACTIONS

As practicing researchers, the idea of using related transaction models for modeling workflows was appealing to us. We felt that such a model could provide a rigor or structure that was lacking in the work on workflow management [Ansari et al., 1992, Breitbart et al., 1993]. There are few, if any, examples of successes in developing systems that implement ATMs for significant commercial, large-scale multi-system applications.

Requirements of such applications include:

1. capability to explicitly define the functionality and organizational structure of organizational process involved,

2. support of coordination and execution of tasks in heterogeneous intra- and inter-enterprise environments,

3. modeling and support for human involvement with the run-time system, and

4. error handling and failure recovery.

Workflow management is specifically defined to address these real-world challenges. It provides the tools to integrate humans, computer systems, information resources and organizational processes into a unified solution. Hence, the requirements of WFMSs are far more challenging than those faced by current database systems [Alonso and Schek, 1996b]. In workflow applications, database resources might comprise only a part of the entire solution. For a task that entirely interacts with a DBMS, executing it as a transaction is often a desirable choice. At the same time, workflows involve other user and application tasks (e.g., tasks that interact with legacy systems) that are non-transactional in nature.

Due to the wide acceptance and applicability of workflows to application domains that extend beyond transaction based (primarily database related) environments, the term *transaction* is being used in a more loose manner with various connotations. These interpretations are based on: 1) the type of tasks and processing entities that are part of the workflow process, 2) the application domain or semantics of the organizational process that is being modeled, 3) the communication infrastructure that is used to develop the WFMS, and 4) transactional or advanced transactional semantics (such as relaxed isolation and atomicity) that can be attributed to the tasks, sub-workflow, or the workflow as a whole. It is important to understand each of these interpretations to be able to appreciate the similarities and differences between transactions from the world of database systems and those involved in the realm of multi-system workflow management systems. Let us consider some of the frequently encountered interpretations for the term *transactions* in the context of real-world workflow applications and WFMS that support workflow applications:

1. **Task specific interpretation in databases and distributed transaction processing.** In general, a workflow task is considered to be a *black box* that is functional in nature, i.e., the functionality of the task is orthogonal to that of the workflow process [Alonso et al., 1995b]. The tasks themselves could be transactional or non-transactional in nature [Rusinkiewicz and Sheth, 1995, Krishnakumar and Sheth, 1995]. Transactional tasks are those that minimally support the atomicity property and maximally support all ACID properties of traditional transaction models [Miller et al., 1996, Krishnakumar and Sheth, 1995]. These tasks typically include those that interact with a DBMS by using *BEGIN_ TRANSACTION - END_TRANSACTION* semantics, contracts (stored procedures), and two-phase commit (2PC) tasks [Wang, 1995, Miller et al., 1996] for synchronizing transactions across multi-DBMSs. In addition, tasks that use the *XA-Protocol* [Gray and Reuter, 1993] based RPC to communicate with transactional processing entities such as a TP-monitor in a distributed environment [Wodtke et al., 1996] can also be included in this category. Non-transactional tasks are used to include applications that cannot ensure isolation or atomicity as a part of the workflow process. Such task types are commonplace in the real-world and involve activities requiring interaction with humans, legacy systems, and others that interface with other processing entities that do not provide transactional support (e.g., HTTP servers, Lotus Notes, file systems, word processors, spreadsheets and decision support systems).

2. **Domain specific interpretation.** The move from a paper-based society to a paper-less one, and the increasing popularity of electronic commerce have led to evolution of standards for electronic data exchange across organizations. Some of these include (EDI) standards such as ANSI Accredited Standards Committee (ASC) X12 that are used in numerous commercial settings (e.g., ANSI 270 and 271 transactions for healthcare eligibility inquiry and response used in [Sheth et al., 1996b]), and the ANSI HL7 standard that is used specifically in the medical domain. The term *transaction* in this setting refers to the exchange of sufficient data in a standard electronic format necessary to complete a particular business action often using domain specific information. This view of a transaction tends to focus more on business requirements and contracts rather than on the need for maintaining data consistency within a database or to support atomicity or other transactional property between communicating processes or for a RPC call. Workflow technology is being applied in various forms to application domains such as manufacturing, bank-

ing, healthcare and finance that use domain specific *transaction* formats extensively. One of the tasks within a workflow process could involve sending data from one information system to another using an EDI *transaction*. At the receiving end, another workflow task could write the data that it receives to a DBMS in a *transactional* (having ACID properties) manner. The semantics associated with each of these transactions are different. Hence, the WFMS would have to be designed so that it can deal with different *transaction* forms in an appropriate manner.

3. **Business-process specific interpretation.** Database transactions and transaction processing aim at preserving data consistency and ensuring reliability in case of faults and failures. These semantics cannot be applied directly to workflow systems since tasks within a workflow process are both transactional and non-transactional in nature. However, at the same time, workflow systems should be correct and reliable. Correctness and reliability in the case of workflow systems is more applicable from a broader perspective - that of the *organizational process* involved in addition to the data that forms a part of the process. According to [Eder and Liebhart, 1995], a workflow *transaction* should ensure consistency from the business process point of view. The notion of a workflow transaction according to this view, is broader as compared to that of traditional transactions. Implementation support for such a concept would require an additional layer of control than that provided in transaction processing since workflows include features (e.g., roles, worklists, error handling) that are not available in (advanced) transaction models and transaction processing systems.

4. **Infrastructure specific interpretation.** Workflow management systems are large-scale applications that can be implemented using various infrastructure technologies such as Customized Transaction Management (CTM) [Georgakopoulos et al., 1995], Distributed Object Management specifically using CORBA [Georgakopoulos et al., 1994, Miller et al., 1996, Sheth et al., 1996b, Wodtke et al., 1996, Schuster et al., 1994], World Wide Web [Palaniswami et al., 1996, Sheth et al., 1996b, Technologies, 1995], TP-monitors [Wodtke et al., 1996], Lotus Notes [Reinwald and Mohan, 1996] and security services (as in *secure transactions* supported in the electronic commerce and Web-based services). The concept of *transactions* has been addressed in many of these technologies to some extent. For example CORBA provides an Object Transaction Service as a part of the Common Object Services Specification [OMG, 1995a] that enables objects in distributed environments to take part in a *transactional context*; TP-monitors also provide transactional semantics in a distributed environment. The HTTP protocol used in the Web paradigm, on the other hand, does not provide any transactional semantics.

Hence, we see that different interpretations of transactions are supported by each of these infrastructures.

From the above discussion, it is important to observe that the notion of transactions in workflow management is more general compared to that in transaction processing and DBMSs.. Its interpretation could involve various variables associated with the factors mentioned above. *Unlike advanced transaction systems, WFMS interact with database systems if required as part of the organizational process, however, this is not their primary focus.*

1.8 CONCLUSION

We view workflow management as an attractive approach to *programming in the large* for enterprise applications. Tasks within a workflow are modeled at a higher degree of granularity than traditional database transactions (i.e., component transactions in a ATM or subtransactions in a distributed transaction). The tasks themselves could be either transactional (e.g., database transactions, and processes interacting with a TP-monitor) or non-transactional (e.g., human-oriented activities, and processes that do not observe one or more of the transaction properties). Also, most real-world workflow processes involve activities that are long running in nature and execute in distributed and heterogeneous environments. The processing entities that execute or carry out a task might not support the protocol for guaranteeing transaction behavior. At the same time, it is desirable that workflow systems be reliable and ensure correct execution of processes just as transactions guarantee such characteristics for ensuring data consistency. It has been accepted that strict ACID transactions do not have direct applicability in the workflow domain as workflow systems differ to a large degree from traditional database systems.

In our perspective, the role of ATMs in workflow systems is of a supportive nature. Advanced transaction modeling concepts are quite restricted in terms of being directly applicable in process-oriented, large-scale workflow applications that run in HAD computing environments. Workflow systems today are still weak in terms of characteristics such as fault-tolerance, consistency, and in their support for recovery in case of exceptions and failures. ATMs have addressed most of these problems in the domain of database systems. Research in the areas of workflow systems can benefit from these approaches from a conceptual point of view.

Transactional semantics such as atomicity and isolation in their strict sense are not practical in workflow systems since tasks in a workflow domain are generally long-lived and could themselves be non-transactional in nature. Many of the solutions for recovery in transaction processing systems can be used to address recovery issues in workflow systems, for example, advanced transaction concepts such as compensation can be mapped to the workflow domain in terms of a compensating task that could be used to *undo* (often partially) what was done by an incomplete task; logs similar to those in transaction processing could be maintained for recording the history of the workflow process, thereby aiding in the recovery process [Krishnakumar and Sheth, 1995, Alonso et al., 1995b, Eder and Liebhart, 1996].

To address many of these advanced issues, workflow systems should borrow ideas that have been used effectively in concurrent, large-scale distributed and database systems, but should not rely entirely on them as many of these systems have developed models for environments that are limited in scope as compared to that in workflow systems.

In conclusion, we summarize the observations we have made in this chapter:

- There are several interpretations for *transactions* in organizational processes today and all or most of them may need to be accommodated in a workflow technology that supports organizational processes.

- Features offered by ATMs meet a very restricted subset of requirements of large-scale enterprise-wide workflow systems (see the appendix for a normative comparison of ATMs and workflow systems).

- We do not see ATMs as being a primary basis for modeling and executing workflow systems that have real-world commercial applicability. However these models provide useful features (e.g., relaxed atomicity, relaxed isolation, concurrency control and recovery) which can be used in components (e.g., tasks) that form a part of a WFMS. Traditional transaction processing and ATMs provide valuable concepts that can be applied towards partly solving the problem of error handling and recovery in WFMSs.

- Implementing reliable large-scale WFMSs involve requirements that are beyond the capabilities of transaction systems and ATMs (e.g., distribution of the workflow architecture, heterogeneity of the operating environment, business process governing the workflow, organizational structure of the enterprise, nature of the tasks, etc.). A lot of valuable research has been done on error handling and recovery in the areas of distributed systems, software engineering, and organizational sciences. Research and development in the domain of reliable WFMS should leverage these efforts to supplement the limitations of traditional transaction and ATM based systems.

There is a need for multi-disciplinary research to address the challenging issues raised by emerging workflow technology. Humans are an essential part of any organizational process, and human work involves many diverse issues. Therefore, research involving expertise from multiple disciplines is most likely to bring the highest return. Information is another critical asset of any organization, as discussed in [Sheth et al., 1996a]; we believe that more human-centric approaches with integral support for information management are needed for a successful workflow technology. We need to look beyond the capabilities provided by transaction processing systems and ATMs in modeling the complexities of large-scale, mission-critical workflow applications of the future.

Notes

1. METEOR refers to the project carried out at Bellcore. $METEOR_2$ is its follow on at the LSDIS Lab of the University of Georgia.

2. The Object Request Broker forms the core of the CORBA model; it is the middleware layer that makes it possible for distributed objects to communicate with each other. For details see [OMG, 1995b].

Acknowledgments

$METEOR_2$ is a group project, and our discussion related to it reflects contributions of numerous past and current members of the project (*http://lsdis.cs.uga.edu/workflow*). Current active members include, S. Das, Prof. K. Kochut, Prof. J. Miller, D. Palaniswami, Prof. A. Sheth, D. Worah and K. Zheng.

This research was partially done under a cooperative agreement between the National Institute of Standards and Technology Advanced Technology Program (under the HIIT contract, number 70NANB5H1011) and the Healthcare Open Systems and Trials, Inc. consortium. See URL:http://www.scra.org/hiit.html. Additional partial support and donations are provided by Post Modern Computing, Illustra Information Technology, and Hewlett-Packard Labs.

Appendix: A Normative Perspective

	Advanced Transaction Models	**Workflow Systems**
Theoretical Foundation	Usually good theoretical basis.	Weak dependency, except for scheduling components. Driven by practical considerations.
Granularity	Transactions.	Tasks, activities, or steps
Methodology	Data-centric. Emphasis on data consistency.	Process-centric. Emphasis on task coordination.
Correctness Criteria	Serializability.	Primitive, often limited to scheduling.
Failure Atomicity	Inherent.	Not part of most models.
Concurrency Control	Inherent.	Limited support.
Recovery	Well-defined. *Rollback* and *compensation.*	Insufficient support. *Forward recovery* when supported.
Error Handling	Limited.	Very limited.
Task/Activities	Supports transactions only.	Supports both human and application tasks.
Processing Entities	Usually DBMS.	Heterogeneous systems (e.g., DBMSs, TP monitors, legacy applications, humans)
Coordination Support	Limited.	Inherent.
Modeling Organizational Structure	Usually absent.	Varies significantly.
Worklists	No support.	Strong support.
Flexibility	Varied.	Good.
Implementation Status	Very few exist.	Numerous commercial products and few research prototypes.
Applicability to Non-DBMS applications	Very limited.	Extensive.

2 WORKFLOW MANAGEMENT: THE NEXT GENERATION OF DISTRIBUTED PROCESSING TOOLS

Gustavo Alonso and C. Mohan

Abstract: Workflow management systems have attracted a great deal of attention due to their ability to integrate heterogeneous, distributed applications into coherent business processing environments. In spite of their limitations, existing products are enjoying a considerable success but it would be a mistake not to try to see beyond current systems and applications. In today's computer environments, the trend towards using many small computers instead of a few big ones has revived the old dream of distributed computing. There is, however, a significant lack of tools for implementing, operating and maintaining such systems. In particular, there are no good programming paradigms for parallel architectures in which the basic building blocks are stand alone systems. Workflow management provides this key functionality, suggesting its potential as crucial component of any distributed environment. This chapter describes in detail such functionality and provides some insight on how it can be applied in environments other than business processing.

2.1 INTRODUCTION

One of the basic platforms in which to implement generic distributed systems is commodity hardware and software, usually in the form of clusters of workstations connected via a network. The continuous increase in computing power, storage capacity, and communication speed has made these share nothing configurations viable and cost effective alternatives to more tightly integrated multiprocessor architectures. There is also the added advantage of having most of the necessary infrastructure already in place, both in terms of hardware (clusters of personal computers connected by a Local Area Network) and software (the many existing applications). The only component missing in such environments is the necessary glue to make a coherent whole out of many autonomous, heterogeneous, loosely coupled building blocks. This problem has been addressed from many different perspectives, federated database systems [Schaad et al., 1995], TP-monitors [Gray and Reuter, 1993, Obermack, 1994], persistent queuing [Alonso et al., 1995a, Mohan and Dievendorff, 1994], CORBA [OMG, 1995b], process centered software engineering [Ben-Shaul and Kaiser, 1995] and workflow management systems [Hsu, 1995] being among the best examples.

From a practical point of view, these different approaches can be roughly divided in four categories: *interface definition*, *communication*, *execution guarantees*, and *development environment*. These four categories also correspond to the functionality needed in a distributed environment. In spite of this, existing products and research efforts tend to emphasize only one of the categories, e.g, TP-monitors for execution guarantees; CORBA as an interface definition; queuing systems as communication platforms; or workflow systems for developing distributed applications. Such narrow focus is one of the major limitations of these approaches. Users or designers interested in getting two or more of the four categories of functionality have to resort to combine several heavyweight solutions, which adversely affects performance and usability. Examples to prove this point abound, perhaps the most clear one being the transactional services described in the CORBA standard. These services can only be implemented using what today is known as a TP-monitor. In fact, current implementations do exactly just that: bundle together a CORBA implementation and a commercial TP-monitor. Since both were designed as stand-alone systems and, in practice, must solve many similar problems, the resulting system incorporates a great deal of redundancy and mismatches. As a result, performance and the overall functionality are adversely affected. A more reasonable approach would be to implement the CORBA standard with the transactional services included as part of the original design instead of as an orthogonal module. This would still not be enough, however, as the resulting system would lack, for instance, a development environment. To address this latter point, the OMG (Object Management Group) and the Workflow Management Coalition are joining efforts to define a CORBA Workflow Facility. But as with the transactional services, such facility will only be truly operational and useful when the design incorporates and integrates all these different technologies from the very beginning and not as separate tools.

This same example occurs in many other environments and products. The underlying problem is that no system incorporates the four categories of functionality in the design and, hence, it is not possible to rely on a truly integrated system. But building such system is only possible if the existing partial solutions are first generalized and their functionality becomes available in the form of open systems. It is possible to identify trends in industry that point clearly into this direction (the example of CORBA is one, the incorporation of transactional guarantees and queuing systems in workflow tools is another), but much remains to be done. The role workflow management systems will play in future computing environments is directly related to the idea of integrating the four categories of functionality. One of the factors that have made workflow management so successful is the support they provide for developing complex applications over distributed systems using already existing tools. This same concept can be generalized, turning workflow management into one of the basic technologies for developing large scale distributed applications based on autonomous components. Thus, workflow management should evolve as part of larger, tightly integrated architectures. In order for this to happen, workflow management needs to be reinterpreted from a perspective going beyond current products. This includes generalizing the notion of process, as has been suggested by several workflow designers [Emmrich, 1996, Leymann, 1995], instead of focusing solely on business processes reengineering. In this way, a workflow management system can become a very high level programming language linking, within a single control logic, heterogeneous applications residing over a wide geographic area. Additional technology such as CORBA, queuing systems or TP-monitors will then complete the integrated distributed system in which to exploit the coarse parallelism and distributed characteristics of workflow processes.

2.2 WORKFLOW MANAGEMENT SYSTEMS

2.2.1 Workflow Concepts

Workflow management is a relatively new term. The ideas and concepts associated with it, however, have been around for quite some time. The notion of workflow management can be traced back to prototypes and research carried out many years ago. Some [Swenson et al., 1994] propose as the earliest ancestors the SCOOP project [Zisman, 1978] and Office Talk [Ellis et al., 1991]. Others see the roots of workflow management in the work of imaging companies [Frye, 1994]. In the database community workflow ideas have been proposed under many disguises, mostly in the form of *advanced transaction models* [Elmagarmid, 1992, Waechter and Reuter, 1992, Garcia-Molina et al., 1991, Kreifelts et al., 1991, Nodine and Zdonik, 1990]. The Workflow Management Coalition [Hollinsworth, 1996] suggests no less than six areas that have had a direct influence on the development of workflow management as it is today: image processing, document management, electronic mail and directories, groupware, transactional systems, project support applications, business process re-engineering, and structured system design tools. Even one of the most popular workflow modeling paradigms [ActionTechnologies, 1993, Medina-

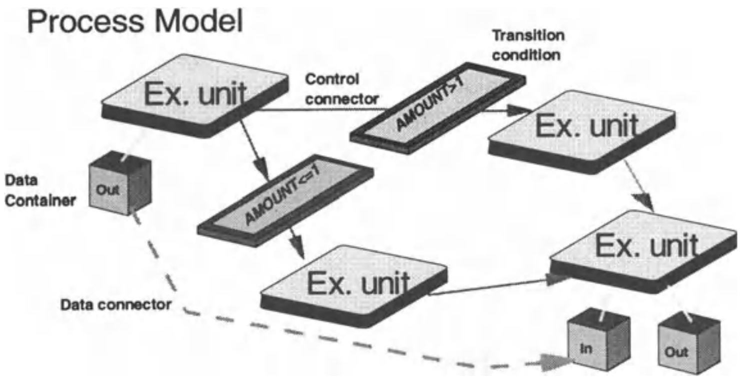


Figure 2.1 Basic components of a workflow process

Mora et al., 1993] can be traced back to early work on artificial intelligence and speech theory. In general, the need for workflow functionality was identified long ago by different communities as they realized the potential offered by computers and communications. For instance, just in the last decade, similar ideas were discussed in areas such as *paperless office* [Tsichritzis, 1982], *office automation* [Bracchi and Pernici, 1985], *groupware* [Ellis et al., 1991], or *computer supported cooperative work* [Kreifelts et al., 1991].

In spite of this early interest, the technology to develop full functional systems has become available only in the last few years. To certain extent, workflow management has found its window of opportunity in this decade thanks to organizational management trends such as business process reengineering [Hammer and Champy, 1993]. As a result, it is uncommon to find a product that it is not directly associated with the reengineering world. But this is likely to change in the future as workflow systems diversify and incorporate ideas from other areas.

2.2.2 Process Representation

The notion of process is central to any workflow system. A process is a complex sequence of computer programs and data exchanges controlled by a metaprogram. It is usually represented as an annotated directed graph in which nodes represent steps of execution, edges represent the flow of control and data among the different steps, and the annotations capture the execution logic. Other forms of representation are possible (for instance based on rules [Ben-Shaul and Kaiser, 1995]) but the underlying concepts are essentially the same regardless of the representation. These are shown in Figure 2.1 and can be described as follows:

Execution unit is the basic instruction of the workflow language. It can be compared with a procedure call in a programming language. Similarly to procedure calls, it can correspond to an internally defined procedure (a process), to a structured block of instructions (a block), or to a remote procedure call to an external application (an activity). Associated with each execution unit there is an input and an output data container used to store the inputs and outputs of the execution unit. A state is associated with each execution unit, as well as two conditions, one to determine when the execution unit can start and another to determine when it has been completed successfully.

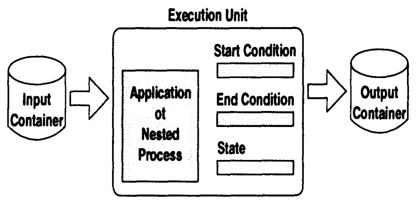


Figure 2.2 The execution unit as the basic building block of a workflow model

Process is the equivalent of a program. It specifies the execution logic by linking execution units via control and data connectors. To allow nesting, a process can be represented as an execution unit, in which case it becomes one more step within another process. The possible states of a process are shown in Figure 2.3.

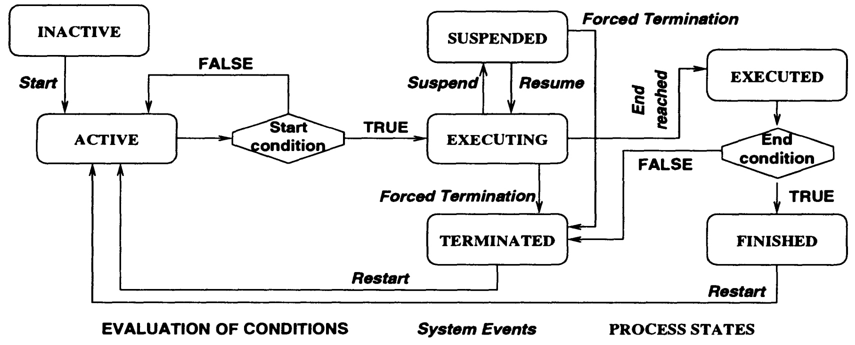


Figure 2.3 State diagram of a process

Blocks allow the modular decomposition of a process very much like in structured programming. A block is equivalent to a series of execution units bracketed by a *BEGIN* ... *END*. It is essentially another process except that it has no name and can not be reused. Contrary to sub-processes, which are bound to the parent process at run time, blocks are instantiated at compilation time. It is possible to associate certain semantics with blocks to denote specialized types of structures such as loops, case statements, and fork or parallel-do operations.

Activities correspond to the invocation of external applications. Processes and blocks are structuring constructs that have no effect outside the workflow system. Activities correspond instead to interactions with the external world. They can be *manual* if they require human intervention to be started, or *automatic* if they can be started without human intervention. In general, manual activities correspond to activities that also require user involvement to be completed (filling a form, providing some information, making a decision). Automatic activities, on the other hand, usually do not require user participation (transactions over a database, index calculations, statistical calculations, etc.). Associated with each activity there is an application and a set of eligible users indicating which application is to be invoked and the users allowed to execute it. Figure 2.4 shows the possible states of a manual activity (automatic activities have a similar but slightly simpler state graph).

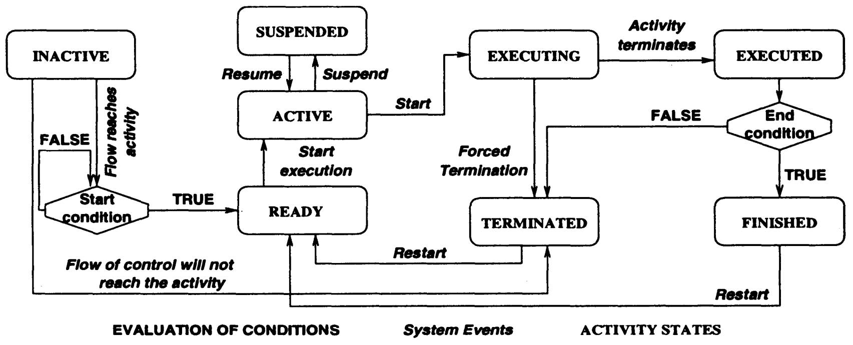


Figure 2.4 State diagram of an activity

Data containers provide a persistent repository for the input and output parameters of an execution unit. In the case of processes, the input data container collects input parameters for the entire process. When the process starts to be executed, these input parameters are distributed among the input containers of the execution units within the process. As these execution units terminate, their outputs are transferred from their own output data containers to the output container of the process. For activities, the input data container stores the parameters to use when invoking the application and the output data container stores the application's return values.

Data connectors are used to specify data flow between execution units. For instance, the input data container of a process is mapped to the different input data containers of the execution units within the process by indicating via data connectors which variable in the process container corresponds to which variable in an execution unit container. The same mechanism is used to pass the results produced by an activity as inputs to another activity. Together, data containers and data connectors eliminate the need for global variables and allow each execution unit to define its own parameters. The use of data connectors forces the workflow programmer to explicitly state the data flow within the process and helps to optimize data migration in applications distributed over a wide geographic area.

Control connectors indicate the flow of control among execution units. In general, control connectors can only be used between execution units at the same level of nesting, which strengthes the modularity of the language. That is, it is not possible to add a control connector between activities of two different blocks, or between an activity external to a process and an activity within the process. Each control connector has a condition attached to it, which is used to determine when the control connector is to be followed.

Conditions are boolean expressions over data in the data containers. They indicate when certain actions should take place. In the case of execution units, there are two types of conditions to be considered: *start* and *end* conditions. The former specifies when an execution unit can start to execute (the exact meaning varies depending on whether the execution unit is a block, a process or an activity). The latter is used to determine when an execution unit has terminated successfully, usually by checking the return code provided in the corresponding output data container. In the case of control connectors, conditions indicate whether the connector should be followed or not. If the condition of a connector is evaluated to true, the execution unit at its end is taken out of the inactive state (the exact action depends on the nature of the execution unit). If the condition associated to a connector evaluates to false, it indicates that the connector will not be followed. Marking a control connector as false triggers the procedure of *dead path elimination* which marks off all connectors and execution units that will never be executed. This helps to determine when a process has terminated its execution.

Applications represent the external programs to be invoked as part of the execution of an activity. Applications are registered with the workflow system very much like applications being installed in an operating system. The registration process allows the workflow system to establish in which network addresses a given application can be found, access permissions associated with it, under which operating system it runs, associated paths, input parameters, and any other additional information necessary to invoke the application remotely. Once registered, applications are invoked by linking them to activities.

Staff represents users and sets of users. Similarly to applications, users must be registered with the workflow system. Users must be registered individually and later on they can be grouped into more meaningful sets, usually known as *roles*. Roles allow the system to refer to groups (programmers, managers, engineers, sales representatives) when allocating work, instead of having to deal with individual users. When an activity or a process is defined, part of the information specified is the users or group of users that are eligible to execute the activity or to start the process.

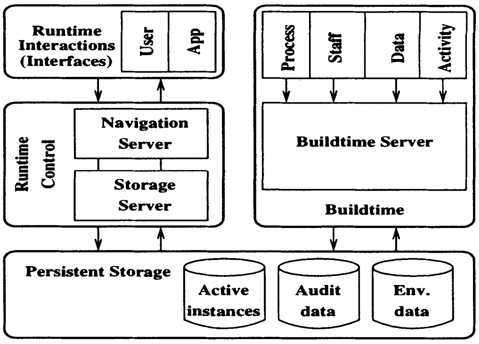


Figure 2.5 Functional architecture of a workflow management system

2.2.3 Architecture

Architectural details vary from product to product and are evolving very quickly as products try to cope with more demanding environments. It is possible, however, to distinguish a set of features common to most systems by looking at the functionality that needs to be provided.

2.2.3.1 Functional Description. The basic functionality of a workflow system can be divided in three major areas: *design and development*, *execution environment*, and *interfaces*. Usually, these three areas are also referred to as *Buildtime*, *Runtime control* and *Runtime interactions* respectively [Hollinsworth, 1996, WFMC, 1994].

For design and development, workflow systems provide a language along the lines described above as well as several tools to register users and applications. Programming, i.e., designing, a workflow process is usually done through a graphical interface in which execution units are represented as a variety of selectable icons and connectors as directed links between these icons. This approach is perhaps the most user friendly but it has several drawbacks, the main one being that it becomes rather cumbersome to visualize and manipulate large and complex processes. Current systems usually provide a more textual language in which to specify processes but, in most cases, these languages are not adequate for large scale programming. It is likely that, in the future, more sophisticated languages will be supported. Additional tools are also provided for debugging and compiling the process description into object code that can be used for execution. Current systems provide only a primitive development environment but, given the key role it plays, it is likely that the buildtime component of future systems will be significantly enhanced [Leymann, 1995, Silver, 1995].

The execution environment can be divided in two parts: *persistent storage* and *process navigation*. Persistent storage provides a repository where all the necessary information about the system can be kept and retrieved at any time. Persistent storage is managed via a *storage server*. Since the information involved is often complex and it is necessary to support complex queries over it, most systems use a database management system for this purpose. The advantage of relying on persistent storage is that it makes possible to recover from failures without losing data (forward recovery) and also provides the means to maintain a record of the execution of processes. These two features open up many interesting possibilities when programming distributed applications. For instance, the fact that the execution is persistent implies that failures will not require to repeat the entire process, execution can be resumed from the point where it was left when the failure occurred. It is possible to subdivide the persistent storage in several areas according to the data stored: *audit trail*, *active instances* , and *environment information*. The audit trail contains information about already executed processes. In business environments this provides the information necessary to evaluate the organization's performance, system evolution, potential bottlenecks as well as supporting data mining and analysis techniques. Active instances correspond to the persistent state of processes being executed, which can be queried through monitoring tools provided by the user interface. The environment information corresponds to the staff and applications. It is used to locate applications and to determine the invocation method as well as to locate users and to determine their access rights. Process navigation is performed by the *navigation server* or *WFM Engine*. It mainly involves evaluating the conditions specified for activities and control connectors, activating or deactivating control connectors and triggering status changes in execution units according to the events taking place in the system. Usually, all these operations are performed as transactions over the underlying storage server.

Finally, a workflow system supports two types of interfaces: users and application interfaces. Users interact with the workflow system through a *worklist* which acts as a repository for all the activities assigned to the user. This interface can be as simple as a list of manual activities waiting to be selected by the user or as sophisticated as a dynamic interface to the audit trail for querying information regarding already executed processes. The worklist is created when the user logs-in and updated every time a new activity becomes ready for execution (updates are sent using the environment information, which is also kept up-to-date regarding which users are connected to the system at any given time and from which location). Applications are handled on a location basis. Users will usually connect to the system from a PC or workstation. These locations will have an application interface so applications can be started when users decide to execute an activity. But it is also possible to have application interfaces in locations where no users are connected. This allows, for instance, to connect to mainframes, specialized workstations or execute automatic acti-

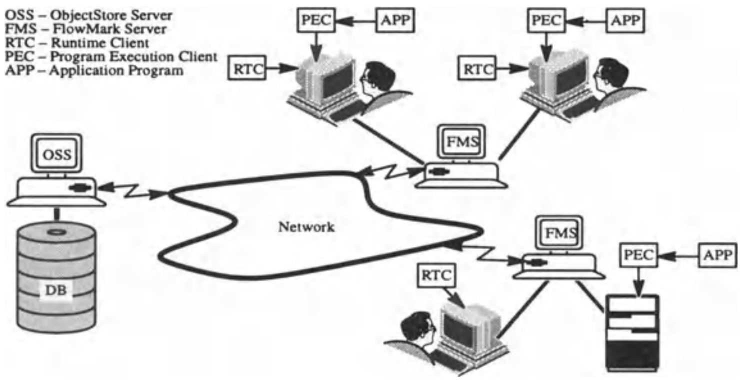


Figure 2.6 Runtime architecture of IBM FlowMark

vities across wide area networks. Which type of connections are allowed and supported depends largely on the intended use of the product, i.e., whether it is a collaboration tool to be used in a LAN environment or a production tool to be used in conjunction with OLTP (On Line Transaction Processing) and OLAP (On Line Analytical Processing) systems.

2.2.3.2 Runtime Architecture. Current workflow management systems serve as platforms for executing distributed applications designed according to business rules. The same functionality they provide for business processes can be used in generic distributed applications. Thus, very much like in the case of TP-monitors [Gray and Reuter, 1993], workflow systems are slowly evolving towards specialized, multi-platform distributed operating systems. As a generic example of existing architectures, Figure 2.6 shows the architecture of FlowMark [IBM, 1995, Leymann, 1995].

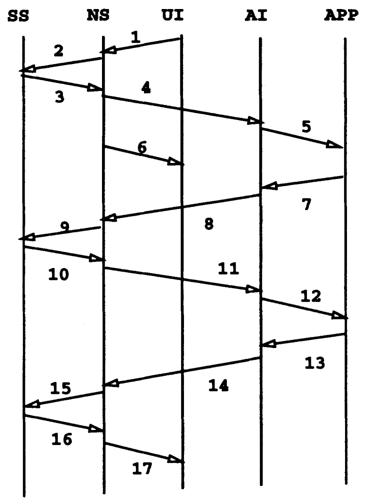
Most workflow systems are built on top of a database management system. In the case of FlowMark, the database is Object Store (represented in Figure 2.6 as *OSS* and *DB* which together act as the storage server). Most other systems are based on relational databases, for instance: ActionWorkflow is based on Microsoft SQL Server, WorkFlo of FileNet uses Oracle, and InConcert of XSoft can use Informix, Oracle or Sybase engines [Silver, 1995, Thé, 1994]. The navigation server, represented in Figure 2.6 by the *FMS* component, is usually implemented as a client of the database since most navigation steps involve getting information in and out of the database.

The rest of the system components used during the execution of a process are connected to the navigation servers, which can also be connected among themselves [Alonso et al., 1995b]. These connections do not need to be over a LAN, they can also take place through a WAN or even from mobile clients [Alonso et al., 1996c]. A common configuration is to have the application and user interface in the same location where the user accesses the system. This allows the user both to access the corresponding worklist and to execute activities locally (which, of course, also requires to have the application locally installed). In Figure 2.6 this is represented by the Runtime Client (*RTC*), the Program Execution Client (*PEC*), and the application (*APP*). These correspond to the user interface, application interface and application being invoked respectively. It is also possible to configure nodes to host only one application interface and specialized applications. Such configuration plays an important role when automatic activities are involved, for instance, when a series of transactions are executed over a database server.

2.2.4 Process Execution

The way execution proceeds in a workflow system is best illustrated with an example (this example follows the architecture and runtime interactions of FlowMark) [Alonso et al., 1996c]. An execution unit becomes ready for execution as a result of a navigation step. In the case of processes, when they reach the "active" state all of their starting activities are set to ready and any necessary input data transferred to the corresponding input data containers. In the case of activities, when they reach the "ready" state, the navigator performs role and staff resolution to determine all the users who are eligible to execute the activity and updates the worklists of all these users by including the activity as a new workitem. If the activity is an automatic activity, then it immediately changes to the "active" state during which the navigator locates a node where the activity can be executed. When the corresponding application is invoked, the activity then switches to "executing". Manual activities, on the other hand, must wait until a user selects the activity for execution. In this case, the exchange of messages between the different components is shown in Figure 2.7.

Manual activities appear in the worklist of all users eligible to execute it. When a user selects the activity, the user interface sends a *start activity* message to the navigator. The navigator reacts to this message by taking several steps. First, the activity is deleted from the worklists of all other users by sending a message to these worklists indicating that the activity is no longer available. Second, a transaction is started over the storage server to retrieve the information related to the corresponding application. This information allows the navigator to determine which application interface will be responsible for executing the activity. It is possible for an application to reside in many locations. If it requires interaction with the user, the application is usually invoked at the user's location, otherwise simple heuristics can be used to select the most appropriate location (load balancing, overhead, pre-established priorities, etc.). Once the application interface has been selected, a *start program* message is sent to it. As the third and final step, the navigator sends an *activity running* message to the user interface from where the activity was selected so the status



1. Start Activity
2. DB update
3. Commit Transaction
4. Start Program
5. Start Application
6. Activity Running
7. API call
8. Data Request
9. DB query
10. Query results
11. Requested Data
12. API call return
13. Application terminates
14. Program Terminated
15. DB update
16. Commit and Next Activities
17. Activity Terminated

SS= Storage Server
NS= Navigation Server
UI= User Interface
AI= Application Interface
APP= Application

Figure 2.7 Steps involved in the execution of an activity

of the activity can be updated and the progress of its execution monitored from the user interface.

Any communication between the application and the workflow system takes place through API calls to the application interface. Application interfaces are multi-threaded so as to be able to cope with several applications being executed simultaneously at the same location. Thus, upon receiving a *start program* message, the application interface spawns a thread for the particular application and this thread will start the application. Any initial parameters to be passed to the application when it is invoked are sent to the application interface along with the *start program* message. The application may, however, request additional information from its input data container by issuing API calls to the application interface. These calls are received by the application interface which will forward a *data request* message to the navigator. The navigator, upon receiving such request, executes a transaction over the storage server and forwards the *requested data* to the application interface. The application interface then completes the API call by returning the data to the application. When the application terminates, the application interface sends a *program terminated* message to the navigator, along with any values returned by the application. At the navigator, this message triggers the execution of a transaction that will store the values returned by the application in the appropriate output data container. The navigator then proceeds to perform the corresponding navigation steps: check the end condition of the activity, if it is false the status of the activity is set to "terminated", if it is true the activity status is set to "finished" and then the outgoing control connectors evaluated, and so forth. As a final step, the navigator sends an *activity terminated* message to the user interface indicating that the selected activity has completed its execution. This message results in the activity being deleted from the worklist.

2.3 FUNCTIONALITY AND LIMITATIONS OF WORKFLOW MANAGEMENT SYSTEMS

There are three key features in any successful workflow product: *availability, scalability* and *industrial strength design* [Alonso and Schek, 1996a, Mohan, 1996, Georgakopoulos et al., 1995]. Without availability, workflow systems will not be used for mission critical processes. Without scalability, they will not be used to support large organizations. Without industrial strength, their applicability is greatly reduced. The problem with these obvious requirements is that they exceed those of current database and transaction processing technology, which can be considered the state-of-the-art in corporate computing. As a consequence, the robustness and technological maturity reached in the transaction processing area is all but lacking in workflow systems [Gawlick, 1994]. In spite of their initial success, current systems still need to be further developed along these three areas:

2.3.1 Availability

The goal of current systems is to become the central tool for the coordination of mission critical processes. The most likely candidates to use current workflow systems are large corporations in which the number of potential users can be in the tens of thousands, the number of concurrent process in the hundreds of thousands, and the number of sites connected to the WFMS in the thousands, distributed over a wide geographic area and based on heterogeneous systems [Kamath et al., 1996]. In such environments, availability is a key feature. Fortunately, most failures in a workflow system can be masked using the redundancy inherent to the architecture. For instance, it is common to have the same application installed in several nodes. If one of them is not available, it may be possible to invoke the application at a different node. The same applies to all other components except to the storage server. A workflow system acts as an execution engine driven by the storage server, currently implemented in most systems as a centralized database. This centralized database becomes, sooner or later, a bottleneck and a single point of failure. It is certainly possible to rely on the underlying database to provide the necessary degree of availability. This approach has significant disadvantages, however. In the first place, database techniques are usually product based, i.e., the primary and the backup are the same database. In practice, this would tie the workflow architecture to a particular database and is in conflict with the distributed and heterogeneous nature of the system. It would also require either a backup for every individual system or a single remote backup for the entire system, which may be distributed over

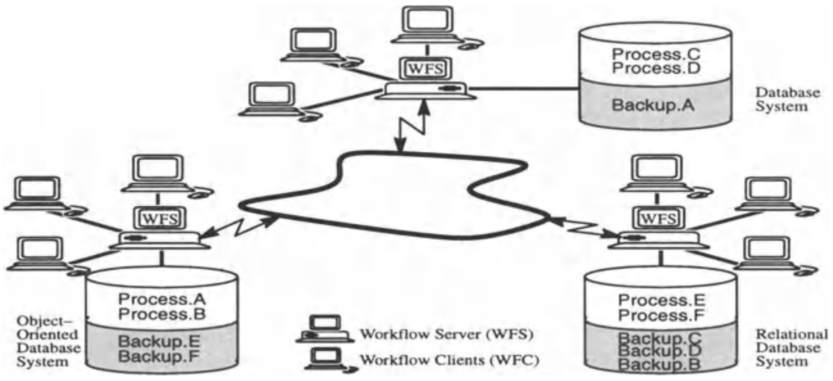


Figure 2.8 A flexible backup architecture for workflow systems

a wide area network. Such solution would be fairly expensive and it does not provide a good way to cope with the heterogeneity of the storage servers (it is not reasonable to expect that all "workflow clusters" will use the same database as storage server). In the second place, the granularity used in database solutions is very fine, mainly pages or log records [Mohan, 1993], and ignores the semantics of the application. The advantage of having a well defined application and a limited set of interactions would be lost. Finally, availability is always achieved at a price. When and how to pay this price should be an adjustable parameter so as to make the system as flexible as possible.

One way to address these concerns is to provide a backup architecture that is database independent, uses knowledge of the semantics of workflow operations to optimize the exchange of information between the primary and the backup, and allows to adjust the degree of availability in the system [Kamath et al., 1996]. For this purpose, standard database techniques such as hot-standby, cold-standby, 1-safe, and 2-safe, can be used [Gray and Reuter, 1993]. These approaches can be combined to provide a flexible mechanism for high availability on workflow systems. Three process categories are defined: *normal, important* and *critical.* Critical processes use a 2-safe, hot standby policy, i.e., critical processes can resume execution almost immediately after a failure. Important processes use a 2-safe, cold standby approach, i.e., execution can be resumed after failures but only after some delay necessary to update the backup. Normal processes do not use any backup strategy, i.e., execution can only be resumed after the failure has been repaired but, in exchange, normal processes do not create any extra overhead in the workflow system.

Since the degree of availability is set at the process instance level, it is no longer possible to predetermine the primary and backup locations for a process. For this reason and to achieve database independence, there is no single backup for the system. Each storage server will act as both primary and backup depending on the particular process instance, as shown in Figure 2.8. Thus, the backup mechanism can be implemented as part of the standard communications between storage servers. The only difficulty being that the primary and the backup may have different schemas (for instance, between a relational database and an object-oriented database). This problem can be solved by relying on semantic information about the workflow language, which is used to define a *canonical representation* in which each component of a workflow process is uniquely identified. When passing information between primary and backup, this is done using the canonical representation. In practice, this means that the primary only reports state changes to the components of a process, opening up the opportunity to optimize storage and communication overhead. In addition, this backup architecture also allows to perform load balancing in the system by moving the execution of a process from one location to another. For this it is enough to upgrade the copy at the backup so it acts as the primary copy. The mechanism is the same as if a failure would have occurred except in that the change to the backup is triggered by the system according to performance considerations. This provides an effective way to migrate processes and sets the basis for scalable architectures.

2.3.2 Scalability

Due in part to the emphasis placed on cooperation by the first workflow products, most of them were designed with small groups in mind. In many ways, workflow systems have been victims of their own success since once users realized the potential of workflows, these engines were applied in large scale environments for which they were not designed [Alonso and Schek, 1996a, Mohan, 1996, Georgakopoulos et al., 1995, Silver, 1995]. Other design issues aside, the main problem of current systems in terms of scalability is that they rely on a centralized database to implement the storage server, thereby introducing a serious bottleneck in the architecture. There are, of course, several advantages to the centralized approach: lightweight clients, centralized monitoring and auditing, simpler synchronization mechanisms, and overall design simplicity. But, in general, a centralized database results not only in scalability problems but also in performance limitations. The latter are not usually a concern in business processes but they are if the workflow system executes many automatic activities. Such problems can be addressed in several ways: using distributed execution instead of centralized control, and providing a way to tie together several workflow systems, each with its own storage server, into a bigger system. The former approach is still largely a research proposal, the latter a solution currently adopted by most products.

The idea of distributed execution was pioneered by the INCAS prototype [Barbara et al., 1996a]. In INCAS, the execution of a process takes place through an *Information Carrier*. The information carrier is an object that migrates from location to location as execution proceeds. It contains all the information relevant to the execution of the process so as to allow navigation to take place by consulting the data in the information carrier. A similar approach is followed by EXOTICA/FMQM, FlowMark on Message Queue Manager [Alonso et al., 1995a]. In Exotica/FMQM, each node functions independently, the only interaction between nodes being through persistent messages used to trigger the next step in the execution. The basic idea is to partition the process definition into independent subsets that are distributed to the nodes were execution may take place. In contrast to the information carrier of INCAS, where all the information moves from node to node as navigation takes place, in Exotica/FMQM each node stores locally all the information it needs to perform navigation on a given process. Such an approach has also been followed by other prototype systems [Wodtke et al., 1996]. This greatly reduces the communication overhead between nodes and solves some additional problems related to monitoring and state detection. Independently of the form in which navigation takes place, the advantage of the distributed approach is that the need for a centralized database is avoided, which eliminates the performance and scalability bottleneck. Moreover, the resulting architecture is more resilient to failures since the crash of a single node does not stop the execution of other active processes. It is also possible to combine this distributed approach with a backup mechanism such as the one described above to provide both scalability and availability.

An alternative to distributed execution is to use several identical, independent systems. One primitive form of this approach has been successfully used in environments that tolerate *load partition*. If all processes are entirely independent of each other and the shared resources (corporate databases, for instance) are capable of supporting the accumulated load, it is possible to use several identical systems, each one executing part of the total load. This approach allows linear growth but it does not really address more fundamental problems as there is no way for the independent systems to communicate with each other. A more sophisticated solution is based on the same mechanisms described above for increasing the availability of the system. Both critical processes and important processes are replicated somewhere else in the system. Instead of using the copy for backup purposes, it is possible to use it to migrate the execution of processes from the primary to other locations as the load at the primary increases. In this way, the scalability problem becomes just a matter of providing enough locations in which processes can be run. All these locations will share the environment information, which can be easily replicated at all sites since it does not change often. The links between the different locations (necessary for the backup architecture) can also be used for communication between navigation servers so as to allow a navigation server to invoke a subprocess at a different location [Alonso et al., 1995b].

The idea of process migration and remote invocation requires to have reliable communications between the different locations. As with many other distributed applications, workflow systems should rely on persistent queuing to provide some basic guarantees in the exchange of information [Alonso et al., 1995a]. These basic features, already in place in many distributed systems, are not present in current workflow products, limiting their ability to implement solutions to the existing problems. Thus, a first step in the evolution of any workflow system is, therefore, to provide the industrial strength of databases and TP-monitors.

2.3.3 Industrial Strength

Any new system needs some time to evolve and resolve the design inconsistencies, limitations and lack of flexibility of the initial versions. After this evolution period, products become more stable, their functionality well defined, reaching a degree of maturity that makes them reliable, understood and accepted by users. Workflow systems have not yet reached such a state. The demands placed on existing workflow systems go well beyond their capabilities and, in many cases, the *customer profile* designers had in mind was quite different from that of the actual users [Silver, 1995]. The limitations on scalability and availability discussed above are obvious examples, but there are many other glaring limitations. Some of them are product specific and related to the history behind the product (whether it evolved from a document management system, the tools available at the time it was designed, the position of the company in the market, etc.). Examples abound: inability to use subprocesses due to the way data is handled, scalability problems due to the underlying database, architectural limitations due to the communication system used, excessive emphasis on modeling philosophy, and so forth. These limitations are being quickly corrected as the products start to gain a wider customer base and experience with users provides the necessary feedback. There are, however, another set of limitations common to most systems that have no easy solution but need to be addressed before workflow systems can claim to have reached any reasonable degree of maturity.

Among these open questions, the one most often mentioned is exception handling. In environments where the number of concurrent instances may be in the hundreds of thousands with every instance taking several weeks to complete, exceptions affecting single processes are likely to occur. Moreover, it is also likely that, occasionally, the behavior of all active instances needs to be modified to accommodate changes in the organization. These two types of changes are currently not satisfactorily supported. The difficulty they pose derives from the way process instances are stored. There are two ways of doing it: as a compiled program or, more often, as a collection of database entries. Once created, modifying this implicit or explicit "script" is not an easy matter.

Any possible exception that may appear during the execution must be coded in as part of the behavior of the process. Otherwise, exceptions to the expected behavior can only be solved by aborting the entire process (or by invoking a subprocesses that hopefully can solve the situation, but this creates a considerable overhead for the end user). Ideally, exceptions should be handled in a more uniform way, allowing the user to access the process definition, do the necessary changes and resume the execution of the process. This requires a very flexible handling of the process definition: rescheduling activities that have been modified, reusing results that have not been affected by the modification, and mapping the state of the old process to the state of the new process. Existing systems are still too rigid to provide such capabilities.

Another important issue is the interaction with external applications. In current systems, it is usually not possible to suspend the execution of the external application when the corresponding activity is suspended or the entire process aborted. It is also not possible to control any side effects that the application may cause. As a result, failures and rollback of processes become a fairly complex issue for the user. Currently, these problems are solved via manual intervention (even detecting that there is a problem is left to the user in some systems). In the future, a tighter integration will be desirable. This may be achieved by using standard interfaces or by using persistent queues as a way of ensuring reliable asynchronous communication between autonomous systems [Alonso et al., 1995a].

A third issue related to industrial strength is the ability to express logical units within the workflow language. For this, transactional concepts could be used. There is an extensive literature on advanced transaction models [Elmagarmid, 1992] which has touched upon many areas related to workflow management [Alonso et al., 1996b, Breitbart et al., 1993, Waechter and Reuter, 1992, Garcia-Molina et al., 1991, Nodine and Zdonik, 1990]. Transactions are an excellent abstraction to encapsulate behavior (atomicity and isolation, for the most part) and have proven extremely useful in developing a widely accepted theory of transaction management. Current commercial workflow systems, however, do not incorporate transactional notions but there are many indications that this will change in the future [Alonso et al., 1996a, Alonso et al., 1996b, Chen and Dayal, 1996, Eder and Liebhart, 1996, Mohan, 1996, Hagen, 1996, Ben-Shaul and Kaiser, 1995, Leymann, 1995, Sheth and Rusinkiewicz, 1993]. In a workflow environment, transactions can play a significant role as a system component. Persistence in workflow systems is achieved by using a database, a feature that it is unlikely to change. Interactions with databases require to use transactions (as shown in Figure 2.7). The very nature of the environment requires to use transactions if execution guarantees have to be provided. The same problems of distributed commitment and atomicity that arise in any distributed environment also arise in a workflow system. These problems could be addressed using the concepts successfully implemented in TP-monitors [Gray and Reuter, 1993]. In addition, transactions may also play a significant role in the workflow language. As has already been pointed out [Alonso et al., 1996b], many of the ideas proposed in advanced transaction models can be used in workflow environments: compensation [Garcia-Molina et al., 1991], alternative execution [Nodine and Zdonik, 1990], spheres of control and atomicity [Leymann, 1995], to mention a few. Thus, workflow system can be seen as a ubiquitous programming environment for implementing the applications targeted by advanced transaction models [Alonso et al., 1996b, Georgakopoulos et al., 1996, Georgakopoulos and Hornick, 1994]. An example of how transaction may influence the workflow language is the use made of transactions in Encina, a TP-monitor that provides *transactional C* [Transarc, 1995]. Transactional C is an extension of C in which it is possible to bracket sets of instructions (usually service invocations) within a transaction and specify what to do in case the transaction commits or aborts. The same idea, as well as more sophisticated concepts, can be applied to the workflow language to allow the programmer of workflow processes to specify, for example, units of atomicity or compensation expanding several activities [Leymann, 1995] or alternative execution paths in case of exceptions [Alonso et al., 1996b].

2.4 EVOLUTION OF WORKFLOW MANAGEMENT SYSTEMS

2.4.1 Distributed Environments

As mentioned throughout the chapter, the future of workflow management is strongly tied to the evolution of distributed computing. As such, distributed environments require the four categories of functionality discussed in the introduction: interface definition, communication, execution guarantees, and development environments. While existing products are far from providing the four categories, they are slowly converging towards systems that do provide such functionality in an integrated an efficient manner. In such systems, workflow concepts could be one of the basic tools for programming distributed systems.

The characteristics of such distributed environment can be easily derived from the target architecture of existing systems. A quick look to the manuals of products such as implementations of the CORBA standard, TP-monitors, queuing systems, and workflow tools reveals striking similarities in their architecture. In all cases, the system can be succinctly described as shown in Figure 2.9.

In general, the *client* represents the user program invoking the services provided by the distributed system. The client usually resides outside the distributed system but interacts with it through a well defined set of APIs. The *service provider* determines the nature of the system since it plays the role of scheduler, navigator, and system controller. It provides core functionality such

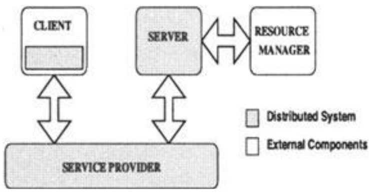


Figure 2.9 Generic architecture of a distributed system

as name services, registration facilities, protocol translations, and request routing. It also serves as the link between all other system components. The *server* is generally a simple proxy for the *resource manager*, acting as a common interface and implemented as a wrapper. Finally, the *resource manager* is the application that performs the operations requested by the client. In TP-monitors, for instance, the resource managers tend to be databases.

The advantage of such architecture is that the services provided can be distributed. Each server/resource manager pair can reside in a different location, with the service provider in charge of routing client requests to the appropriate node after locating a server adequate to the request submitted. Many issues are involved in this simple exchange: load balancing, replication, system configuration, name services, communication overhead, etc., all of which must be balanced in order to have a suitable system, regardless of the concrete application. The difference between CORBA implementations, TP-monitors, and workflow management systems lies on the assumptions made about the components shown in Figure 2.9. CORBA provides a standarized interface in order to have all servers looking alike. A TP-monitor provides similar normalization but with an emphasis on the transactional properties of the service provider. A workflow tool concentrates on the way the client concatenates service invocations and on facilitating the interaction with non-standarized resource managers (the server components being designed on an *ad-hoc* basis). Although these systems perform basically the same function, only workflow management pays sufficient attention to the concatenation of service invocations, i.e., to the programming aspects as seen from the client. CORBA relies on object oriented languages for this purpose, usually C++, TP-monitors have their own language, for instance, *transactional C* in Encina [Transarc, 1995], but none of them offers the flexibility and functionality provided by workflow management systems.

2.4.2 Process Support Systems

The most precise and simplest characterization of a *process* is as a complex sequence of computer programs and data exchanges controlled by a meta-program (the process itself). This characterization is useful in that it implicitly incorporates the goals of any process support system. It also covers a wide range of process types including business processes, virtual enterprises, software processes, manufacturing processes, scientific experiments, and geographic modeling. Such notion of process has proven to be very helpful in developing support tools for applications executing in a distributed fashion and over heterogeneous platforms, as it is the case in most process types. Existing workflow systems, however, target in most cases either business processes or imaging systems, with a few research prototypes addressing other areas [Meidanis et al., 1996, Ben-Shaul and Kaiser, 1995]. Such narrow purpose design along with the limitations mentioned in the previous section significantly restrict the applicability of current products. For instance, recent attempts to use commercial workflow products to support scientific applications have been rather disappointing [Meidanis et al., 1996, Bonner et al., 1996]. These results are not surprising, since the problems faced by current workflow systems are pervasive and appear in many application areas. Thus, the two main challenges of workflow management systems is to incorporate their functionality into a generic distributed system as the one described above, and generalizing the notion of process so as to provide support for any type of process based distributed computation, not just for business applications.

2.4.3 Programming in Heterogeneous, Distributed Environments

Regardless of whether the final system is seen as a distributed environment or as a process support system, the key aspect is the variety of computer tools available as basic building blocks. Workflow management provides the mechanisms to integrate these tools into a more meaningful system by combining them as necessary on a per process basis. Individual applications act as *resource managers*, while the workflow system acts as the language to specify the interactions between these service providers as well as serving as the execution environment in which those interactions take place.

Scientific data management offers a good example of the generalization of the concept of process and of the utilization of workflow tools in a distributed, heterogeneous environment. Scientific applications are known for the size and volume of the data involved [Hachem et al., 1993, Katz and et al., 1993]. Moreover, scientific data has the added problem of the multiple formats in which the information is represented and the multiple transformation to which it is subjected. Most existing research in scientific data management often overlooks the fact that scientific data is seldom used raw. In most cases, the data undergoes complex and successive transformations as part of sophisticated models of physical phenomena. Such transformations are a source of *derived data* which cannot be interpreted correctly without knowledge about how it was created. To make matters worse, the transformations and models themselves

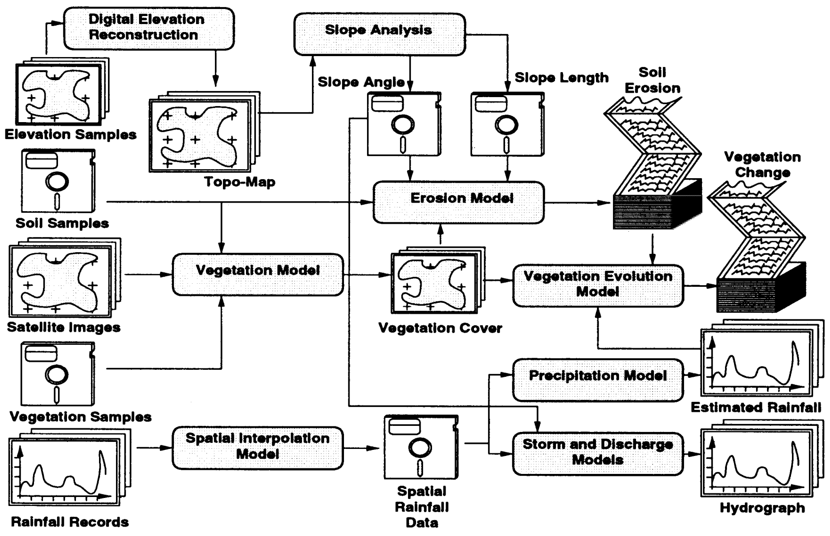


Figure 2.10 Scientific modeling as a workflow process

may evolve as more precise knowledge is available. Support for tracking these data dependencies and evolution is all but lacking in current systems.

Consider, for instance, the model shown in Figure 2.10 as a typical example of how scientific data is handled. The purpose of the model is to study the changes in the erosion patterns, vegetation and hydrographic characteristics of a given area. The model can be divided in three parts. The erosion model takes information about the slopes of the area, its soil characteristics, and vegetation cover to produce an estimate of the erosion of the terrain. Note that the soil information is obtained directly from available data. However, the slope information is not readily available and requires taking elevation samples and processing them to get the desired information. This is done by using two more models, the Digital Elevation Reconstruction and Slope Analysis. The data about vegetation changes is the result of a vegetation evolution model. This model takes several inputs, some of them primitive, i.e. raw data such as the soil map, and some of them derived (by applying other models). Finally the discharge model involves interpolating rainfall records, calculating the storm coverage and applying a flow analysis algorithm to define an hydrograph (showing the flow of water at a given point).

Workflow systems provide the tools necessary to capture such *modeling* activities. Figure 2.10 can be viewed as a workflow process in which the control flow follows the modeling logic and the data flow corresponds to the outputs of particular algorithms that are used as inputs to the next set of algorithms. Using a workflow system for such purpose helps to solve many of the problems posed by scientific data. To start with, the execution is persistent and can be distributed across many different nodes which, first, provides a considerable degree of reliability and, second, opens up the opportunity to parallelize and distribute expensive computations across a network of computers. Moreover, the auditing and monitoring tools of the workflow system keep track of every step of the execution and the data produced. Questions such as the lineage of a data set (how it was produced), data dependencies between data sets, and the algorithms involved in a given model can be easily answered by consulting the audit data of the workflow system. Moreover, complex tasks such as automatic change propagation (triggering the execution of a process when one of its inputs changes) and maintaining data consistency can be performed automatically by the system by using the information recorded about every process.

These ideas can be applied in a variety of scientific environments, once the workflow engine has been modified to support generic processes. The necessary enhancements are no different from those discussed in this chapter (scalability, availability, industrial strength, generalization of the modeling language) and some work is currently being done in this direction [Bonner et al., 1996, Meidanis et al., 1996, Alonso and El Abbadi, 1994]. A workflow system is, however, not just a repository for process dependencies. It can also play an important role in the usability of parallel and distributed environments such as clusters of workstations and PCs. Moreover, by not requiring to modify existing applications, workflow management systems may provide a straight forward solution to the problem of exploiting the parallelism inherent in such hardware clusters. A good example of this is the complex sequence of program invocations shown in Figure 2.10. Assuming the necessary hardware is available, each of the steps of the model depicted could be executed in a different machine, with the workflow tool acting as the scheduler for the overall computation. In this way, in a first stage, the vegetation model, the orographic data extraction and the spatial interpolation programs could be invoked in parallel at different sites. In a second stage, the storm discharge, and the erosion and precipitation models could be invoked in parallel, and so forth. In such scenarios, the workflow system takes on the role of distributed operating system facilitating the integration of independent systems into a single coherent whole. Ideally, workflow management systems should provide such functionality independently of the type of application process. Thus, future workflow systems may be constructed as tools over which concrete distributed applications (business oriented, scientific process support, virtual enterprises, etc.) are built.

2.5 CONCLUSIONS

Workflow management systems have had a considerable success as the first tools capable of both exploiting the coarse grained parallelism implicit in business processes and integrating heterogeneous systems into a coherent whole.

The notion of process, understood as a complex sequence of program invocations and data exchanges, has been widely accepted and applied in many areas. Unfortunately, existing workflow systems suffer from significant limitations that restrict their applicability. Among these limitations, one of the most relevant is their inability to support generic processes. This has lead to disappointing results when current products have been used in areas other than business processes. Other limitations arise from problems not very different in nature from those encountered in tools such as TP-monitors or CORBA environments. These similarities, as shown in the previous sections, are not surprising when taking into consideration the fact that all of these systems have basically the same goals. Solving these limitations requires to develop a new understanding of workflow management. In particular, workflow management systems should be incorporated as key functionality in tools supporting distributed applications, as well as be enhanced to support a more generic notion of process. From a functional point of view, the advances in communication and computing technology allow, and even require, to view workflow management systems as process support systems, i.e., meta-programming tools and execution environments for generic processes. The possibilities of such an approach have been clearly shown in the area of business process reengineering, where workflow management systems have provided an efficient way of designing very complex distributed applications reusing existing components. The example discussed above regarding scientific computing shows how these same ideas can be successfully applied in many other areas, turning workflow management into a key component of future distributed systems. In this regard, it is important to point out that none of the issues discussed in this chapter are tied to business processes, although the initial motivation to work on them may have been business applications. These issues are common to many distributed applications. Efforts like TP-monitors, CORBA, or queuing systems are addressing additional crucial aspects of distributed execution environments, and workflow management should be viewed as one more effort in this direction. The focus on business process has helped to create an initial market and allowed to gain important experience in the usage of workflow systems. The next step is to extrapolate these ideas to other areas and combine workflow technology with other ongoing efforts in distributed computing to arrive at the next generation of distributed processing tools.

Acknowledgments

Part of this work has been done in the context of the Exotica project ongoing at the IBM Almaden Research Center since 1994. We are grateful to all past and current members of the Exotica group, A. El Abbadi, D. Agrawal, R. Günthör, M. Kamath and B. Reinwald, for their contributions. In addition, many of the ideas discussed in this chapter have been developed as part of current research projects within the Database

Research Group of ETH Zürich. We are grateful to Claus Hagen and Hans-Jörg Schek for many helpful discussions about the future of workflow management systems. Even though we refer to specific IBM products in this chapter, no conclusions should be drawn about future IBM product plans based on the contents of this chapter. The opinions expressed here are our own.

II Tool-Kit Approaches

3 A REFLECTIVE FRAMEWORK FOR IMPLEMENTING EXTENDED TRANSACTIONS

Roger S. Barga and Calton Pu

Abstract: It is commonly accepted that the traditional transaction model used in database systems is not well-suited for advanced application domains, because it is lacking in *functionality* and *performance*. In recent years, numerous *extended transactions* have been proposed to address the requirements of advanced database applications. Extended transaction proposals can largely be categorized into two areas: *advanced transaction models* and *semantics-based concurrency control protocols*. Few extended transactions have been ever implemented, not even as research prototypes, and today most remain mere theoretical constructs. Thus, while the research literature bulges with papers there is no practical way to readily leverage these results for the advanced applications for which they were designed. As a consequence, extended transactions have had little impact on industry.

In this chapter we present the *Reflective Transaction Framework*, as a practical method to systematically extend both functionality and interface of a conventional TP monitor to implement extended transactions. The framework provides principled access to existing TP monitor functions and data structures, and carefully extends available transaction services to implement extended transactions. The design of the Reflective Transaction Framework is a synthesis of techniques: *computational reflection* for principled, effective access to TP monitor systems internals; *meta object protocols* to provide explicit descriptions of extended transaction behaviors; and, good software engineering practices for abstraction and modularity of the individual software modules that implement the framework. Using the framework, application developers can implement advanced transaction models and semantics-based concurrency control protocols on production quality TP monitor software, where they can be applied to real-

world applications. It is our hope that this work will help bring together research advances in transaction processing and commercial transaction processing systems, an interaction from which both sides may benefit.

3.1 INTRODUCTION

A vast majority of the ideas that have been proposed in the context of advanced transaction models and semantics-based concurrency control have remained, at least thus far, just that — proposed. In many cases, these *extended transactions* have been shown to have the potential to improve both performance and functionality of traditional transactions for emerging database applications [Elmagarmid, 1992]. However, few of these extended transactions have been implemented, not even as research prototypes, and most remain mere theoretical constructs [Mohan, 1994]. Today, extended transactions are on the critical path for a variety of advanced database applications [Silberschatz et al., 1996], and the time is ripe for their incorporation into commercial transaction processing (TP) systems where they can be applied to real-world applications.

We have introduced the Reflective Transaction Framework [Barga, 1997, Barga and Pu, 1995] to support the implementation of extended transactions on production quality TP monitor software. The insight behind our work is the observation that in most cases, the base functionality provided by a conventional TP monitor is *"almost right"* to implement advanced transaction models and semantics-based concurrency control protocols. While certain functions and data structures are missing, existing functions and data structures of the TP monitor software are basically correct. We do not propose that transaction systems should simply include more features to implement selected extended transaction models. There is no consensus as to which extended transactions a transaction system should include for advanced applications; most likely, there never will be, since each advanced transaction model and semantics-based concurrency control protocol is optimized for a particular application. Furthermore, as application requirements continue to evolve, transaction processing requirements will change and new models will be introduced. Instead, we present a software framework that opens the existing functionality of a TP monitor in such a way that allows programmers access and control over the system, and to tailor the framework to the needs of a particular application. This is called an *open implementation* [Kiczales, 1992]. The open implementation provided by the Reflective Transaction Framework gives the application programmer principled access to TP monitor functions and data structures, and carefully extends the TP monitor functionality with extended behaviors to implement extended transactions.

The design of the Reflective Transaction Framework draws from a variety of techniques to achieve the open implementation of a conventional TP monitor. The framework uses computational reflection [Maes, 1987] for principled, effective access to TP monitor systems internals. A meta level interface, or meta object protocol [Kiczales et al., 1991], is used to provide explicit descriptions of extended transaction behaviors. Good software engineering practices are followed for abstraction and modularity of the individual software modules that implement the framework.

The implementation of the Reflective Transaction Framework introduces *transaction adapters*, reflective software modules built on top of TP monitor software. Transaction adapters leverage existing transaction services of the underlying TP monitor, to the extent possible, as building blocks for constructing extended transaction functionality. Transaction adapters contain a *representation*, or meta-level description, of selected functional aspects of the underlying TP monitor, and maintain a *causal connection* [Maes, 1987] 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 to the representation, but changes in the representation will also cause changes in the behavior of the underlying TP monitor. Each extended transaction has meta-level representation, causally-connected with a transaction running on the TP monitor, that holds information about the transaction and how it is used; in essence this representation defines control and policy. The causal connection between the Reflective Transaction Framework and 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). The strengths of the Reflective Transaction Framework lie in:

1. **Incremental Design.** The Reflective Transaction Framework does not expose the entire TP monitor functionality, but only selected aspects of it. Access to TP monitor functionality and extended transaction behaviors is carefully organized through a well-documented interface, not random user hacking of internals. In addition, programmer access to extended transaction behaviors can be graduated to match application requirements. This is very different from letting an application programmer randomly change the implementation, as happens when a large number of more or less random "hooks" or callbacks into the implementation are provided. The framework preserves the original TP monitor application interface and functionality, enabling extended transactions to be gradually deployed without having to reimplement existing applications.

2. **Interoperability.** The Reflective Transaction Framework insulates transaction extensions from each other, so that multiple extended transactions can exist in one address space and be used in a single program, along with traditional transactions. They are *interoperable*. This addresses two fundamental problems with tailorability: (i) the framework allows the addition of transaction extensions to the TP monitor without requiring all programs to pay the additional cost, even if they do not make use of those

extensions. And (ii) the framework allows programmers to use in one address space different transaction operations that are alike but have been extended differently. For example, an application can select a `commit` operation designed for cooperative group transactions [Nodine and Zdonik, 1990], or a `commit` operation designed for nested transactions [Moss, 1985], etc. Systems that are only globally tailorable typically can not support multiple applications, because interface customizations diverge. By allowing programmers to explicitly control the scope of extensions, at the level of individual transactions, it is possible to customize the TP monitor to suit any number of different applications.

3. **Extensibility.** While the abstractions and extended functions provided by the Reflective Transaction Framework are sufficient to implement a wide range of existing transaction models and semantics-based concurrency control protocols, we anticipate the continued introduction of new extended transactions. Transaction adapters are designed for quick and easy extension. Each adapter encapsulates a set of extensions specific to a selected aspect of TP monitor functionality. This limits the scope of what is effected by an adapter and makes it easy to incrementally extend this functionality. As a result, the framework itself can be extended to implement new extended transactions for emerging applications.

4. **Practical Approach.** Transaction adapters do not duplicate existing transaction functionality, but instead implement extensions to the services provided by a TP monitor. These extensions leverage existing functionality and data structures, to the extent possible, for constructing extended transaction abstractions and services. This not only eliminates unnecessary infrastructure development by building on existing services, but provides efficient, robust base processing for extended transactions.

The contribution of the Reflective Transaction Framework, then, is a practical method to systematically extend the functionality of a conventional TP monitor to implement advanced transaction models and semantics-based concurrency control protocols. Using the framework, application developers will be able to apply extended transactions in real, working environments. It is our hope that this work will help bring together research advances in transaction processing and commercial transaction processing systems, an interaction from which both sides may benefit.

3.2 EXTENDING A CONVENTIONAL TP MONITOR

Transaction processing (TP) monitors supporting atomic transactions are a well established technology that have been around for almost 20 years. TP monitors 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 [Bernstein, 1990].

Early TP monitors, such as IBM's CICS, were proprietary and constructed from single monolithic proprietary systems, but 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 [Bernstein, 1996].

These middleware modules provide the basic functional building blocks required of any TP monitor for transaction processing, such as a *Transaction Manager*, *Lock Manager*, *Log Manager* and *Resource Manager*. Each module exports its transaction services through a relatively simple and uniform "application programming interface" (API). The relationships between an application and the modular functional components in a TP monitor are depicted in Figure 3.1.

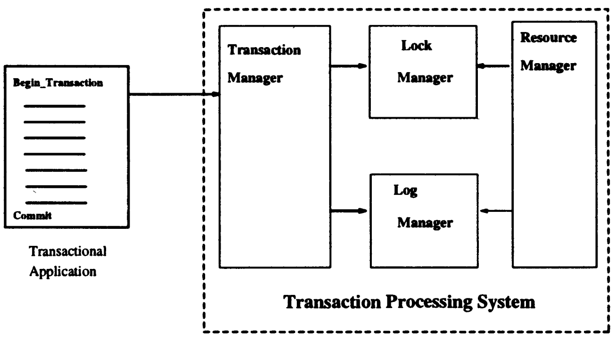


Figure 3.1 Modular Functional Components of a TP Monitor

One seemingly straightforward way to implement extended transactions would be to directly use the available functionality found in the functional components of a TP monitor. Two major impediments complicate this proposition. First, conventional TP monitors have a fixed application-level interface and a fixed implementation of system services. Application developers traditionally access transaction services through the atomic transaction control operations, such as `Begin_Transaction`, `Commit_Transaction`, and `Abort_Transaction`. Ideally, programmers would be able to define and then use similar transaction control operations for extended transactions, such as `Split_Transaction` or `Join_Transaction` for programming with the split/join transaction model [Pu et al., 1988]. However, the single, fixed interface of the TP monitor does not provide access to the underlying transaction services or permit extensions. The functional components of a TP monitor provide a rich set of transaction services, but require the application developer learn intricate details of the TP monitor and available API; the size and complexity of the API itself presents a formidable barrier to even the most accomplished programmers. Second, is the *level of customization* of the TP monitor. The transaction system-level code functions 'underneath' the code of an application program, is not subject to the same programming abstractions. This requires the TP monitor to be customized outside the application, rather than within it, making it impossible for an application to specify its requirements for extended transaction behaviors at runtime. At best, the TP system programmer could adjust the TP monitor functionality through the API to implement a selected extended transaction model *a priori*. Unfortunately, this approach is at the cost of reusability of the TP monitor by applications with other requirements. These issues, and others, combine to give users no convenient way to directly use a conventional TP monitor to define new application interfaces or leverage existing transaction services to implement extended transaction functionality. Efforts to provide implementation support for extended transactions have thus gravitated towards construction of entirely new transaction processing facilities. These efforts, though laudable, have limited practicality.

Computational reflection offers a conceptual tool, the notion of a reflective module, to address the challenges of extending a conventional TP monitor to implement extended transactions. Intuitively, a reflective module allows applications to observe and modify properties of their own behavior, especially properties that are typically observed from some external, meta-level point of view. Reflective modules contain a *representation* of selected aspects of the system, and maintain a *causal connection* between this 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 actual state and behavior of the system. An application can use this representation to both reason about selected aspects of the system, and adjust the representation to influence system behavior. Following the open implementation approach [Kiczales et al., 1991, Kiczales, 1992], a reflective module can be designed to provide a *meta interface* that allows applications to extend and control the implementation of the module's *primary interface*.

Thus, a reflective module with an open implementation enables an application to extend both interface and system services, and to participate in the modules implementation strategy in a principled way.

3.3 THE REFLECTIVE TRANSACTION FRAMEWORK

The Reflective Transaction Framework is a flexible software framework that supports the implementation of extended transactions on a conventional TP monitor. The framework is designed to be implemented as a thin software layer over an existing TP monitor. The implementation introduces *transaction adapters*, reflective software modules built on top of the individual functional components of the TP monitor. Each adapter provides a representation of selected aspects of the underlying functional component, and provides a primary interface to a set of extended transaction services and a meta interface to adjust these extended services. The Reflective Transaction Framework ties together the individual transaction adapters and provides a single, integrated interface for applications to systematically extend both application interface and functionality of a conventional TP monitor to implement extended transactions.

The extensions provided by transaction adapters leverage, to the extent possible, transaction functionality already provided by the underlying TP monitor. The additional transaction functionality provided by transaction adapters supplies the necessary building blocks for constructing a wide range of extended transactions; examples include structured relationships between individual transactions, transaction restructuring, recording and tracking inter-transaction dependencies, delegation of resources between transactions, specification of transaction management events and constraints on event occurrences, and relaxed notions of lock conflicts. The techniques used by the extensions in transaction adapters are not novel; for example, other systems using similar approaches are ASSET [Biliris et al., 1994], DOMS [Georgakopoulos et al., 1994], and the ACTA meta model [Chrysanthis and Ramamritham, 1990]. However, the techniques are applied in a unique way to the problem of carefully extending the existing functionality of a conventional TP monitor.

3.3.1 Extensions Through Transaction Events

One key to the Reflective Transaction Framework's ability to extend the functionality of a TP monitor is a mechanism that integrates extensions with the underlying TP monitor. The Reflective Transaction Framework uses *transaction events* to provide such a binding mechanism. In an event-based system, components announce some system occurrence by explicitly *raising* an event of a particular name. Other parties, interested in learning of the occurrence, register *event handlers* which execute in response to a raised event. Events are generally recognized as an effective technique for implementing loosely-coupled, flexible systems in which relationships between code components must be dynamically established [Sullivan and Notkin, 1992].

In the Reflective Transaction Framework, every transaction control operation represents a possible transaction event, such as `Begin_Transaction`, `Commit_Transaction`, or `Join_Transaction` a transaction changing state (to ACTIVE, ABORTED, COMMITTED, etc.), is a potential transaction event, or when a transaction requests a service (i.e., lock request) from the TP monitor. Consequently, all relationships between a transaction and the TP monitor are subject to change simply by changing the set of handlers associated with any given transaction event. Since the Reflective Transaction Framework allows an application to associate handlers with each transaction event, it is possible for an application to specify its requirements for extended transaction behaviors at runtime at the granularity of each (extended) transaction.

To see how this works, consider the processing of the `Commit_Transaction` control operation for an extended transaction. This transaction control operation raises an event that can be intercepted by a transaction adapter in the Reflective Transaction Framework. If there are inter-transaction dependencies, such as a commit dependency or abort dependency, the transaction adapter can take appropriate actions, possibly delaying the actual commit of the transaction, terminating abort-dependent transactions, or performing commit preprocessing. The Reflective Transaction Framework's use of *transaction events* as a mechanism to integrate transaction extensions is only part of the solution. Extended transaction processing often requires the ability to observe and reason about the state of active transactions, and to effect control over the underlying TP monitor. This is accomplished through reflection and causal connection.

3.3.2 Implementing Reflection and Causal Connection

In the Reflective Transaction Framework, reflection and causal-connection are implemented using *transaction adapters*. Each adapter corresponds to a particular functional aspect of the TP monitor, such as transaction execution, lock management, transaction conflict detection, log management, and transaction recovery. The relationship between transaction adapters and TP monitor functional components is illustrated in Figure 3.2. To expose, or *reify* the internal state of the TP monitor, each adapter contains a number of *meta objects* that represent or model selected structures and behaviors of the underlying functional component. Each adapter provides a *meta interface* that allows the state and behavior of these meta objects to be locally and incrementally adjusted. Furthermore, when the user modifies a meta object in an adapter, the modification is *reflected* to the actual computational state of the functional component in the TP monitor. Figure 3.4 outlines select transaction adapters, along with meta objects and meta interface commands each provides. Thus, transaction adapters provide access to aspects of a legacy TP monitor that are often hidden, enabling users to "reach in" and adjust or extend the behavior of the legacy system using the meta interface. This relationship between adapters at the *meta level* and legacy TP monitor at the *base level* is termed *causal-connection* [Maes, 1987], and is satisfied by all reflective systems.

Applications access transaction adapters using commands in the meta interface. Changes or modifications that an application makes to meta objects in an adapter, using the meta interface commands, affect the behavior of the TP monitor *for only that application.* For example, if an application would like to *relax* isolation properties of a transaction in order to facilitate cooperation with other concurrently running applications, it issues the appropriate meta interface commands to change the conflict detection method for that transaction. Therefore, adapters enable an application to extend the underlying mechanisms of the legacy TP monitor incrementally, dynamically, and in a modular manner at the granularity of each (extended) transaction execution. In the remainder of this section, we provide details on how *reification*, *reflective computation*, and *reflective update* are implemented in the Reflective Transaction Framework. Additional details on the design and implementation of transaction adapters can be found in companion articles [Barga and Pu, 1996, Barga and Pu, 1995].

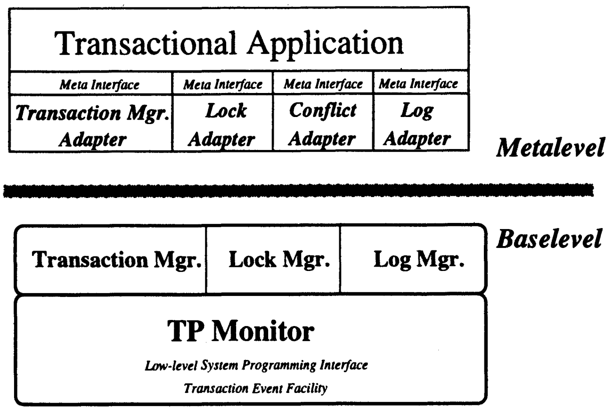


Figure 3.2 Transaction Adapters in the Reflective Transaction Framework.

Reification. In the Reflective Transaction Framework, *reification* is the representation of structural and computational state of the underlying TP monitor component as an object within the corresponding transaction adapter. This effectively provides a representation of the system at the meta level. Reification is implemented using *callbacks*, also commonly referred to as *upcalls*. Upcalls support efficient cross-layer communications and enable the functional components in the TP monitor to pass relevant state information to a transaction adapter in the meta level where it is reified, as illustrated in Figure 3.3. The most important decisions made in designing each transaction adapter were selecting those aspects of the underlying TP monitor component that should be reified. As an example, for the *Lock Adapter* depicted in Figure 3.3, such aspects include the locks being held by each extended transaction, pending lock requests, the procedure used to grant lock requests, and the structure of the lock table. Depending on the transaction model one wishes to implement, other aspects may 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. For generality, each adapter was designed based on the structure, function, and commands of the well-documented TP monitor reference architecture [Gray and Reuter, 1993]. The reference architecture was selected to allow observations on TP monitors in general, yet be concrete enough to reveal implementation details on modern commercial TP monitors. To identify the transaction and TP monitor structures and state that would be reified by the adapters, we referred to ACTA [Chrysanthis and Ramamritham, 1992], a logic-based formalism for defining and comparing transaction models. ACTA provides unifying abstractions for describing extended transaction functions and behaviors, and delineated dimensions of change for realizing extended transactions. A more detailed discussion of transaction adapter design can be found in a companion paper [Barga and Pu, 1995]. The essential point is that the meta objects in the transaction adapters and commands presented in the meta interface are not ad-hoc, but were defined within the context of general TP monitor functionality and extended transaction behaviors. Further, to ensure the flexibility of transaction adapters they were designed to be incrementally extensible. Should the need arise, additional aspects of the underlying TP monitor functional component can be reified as meta objects in the adapter by using the appropriate TP monitor upcalls and adding 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.*

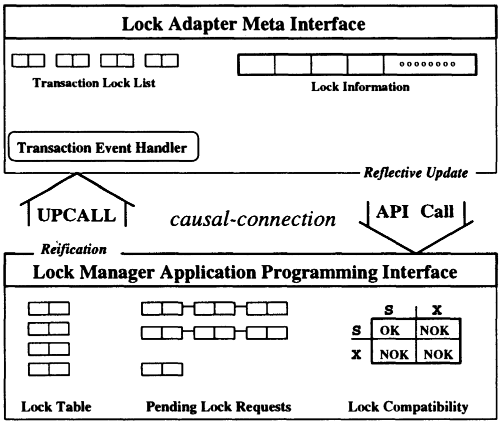


Figure 3.3 Reflective update and reification form causal-connection.

Reflective Computation. 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. When the event is raised execution 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 deny it by simply returning control back to the LOCK MANAGER, effectively implementing semantics-based concurrency control. As this example illustrates, reflective computation not only allows transaction adapters to expose default behaviors of the underlying TP monitor, but also augment legacy functionality with new extended functionality.

Reflective Update. If the reflective computation updates the reified data, then the modifications are *reflected* down to the actual computational state of the underlying TP monitor component in what is called a *reflective update*. 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 3.3. Thus, each transaction adapter not only reifies aspects of the TP monitor functional component, enabling reflective computation, but also provides the means to affect the state and control the component's behavior through reflective update, forming the *causal-connection* between the transaction adapters and legacy TP monitor.

3.3.3 A Separation of Programming Interfaces

Application programmers develop transactional applications using a set of transaction model-specific verbs, or *transaction control operations*. For example, atomic database transactions are initiated by the operation `Begin_Transaction`, and terminated by either a `Commit_Transaction` or `Abort_Transaction` 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, or the operation `Join-Group` introduced in the cooperative group model. Indeed, a transaction model defines not only defines the control operations available to a transaction, but also the semantics of these operations. For example, whereas the `Commit_Transaction` operation of the atomic 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 Management Adapter — reifies state information for transactions executing extended behaviors, and provides meta interface commands to control these extended transactions and adjust the behavior of the underlying TRANSACTION MANAGER functional component. Commands in the *Transaction Management Adapter* meta interface include: `Instantiate`, `Select`, `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, a reflective transaction table, and a transaction dependency graph.

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.

Lock Adapter — reifies information on locks held by transactions and on the state of the lock table, and provides meta interface commands that control the locks held by extended transactions 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.

Figure 3.4 Transaction Adapters in the Reflective Transaction Framework

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 this diversity between different advanced transaction models, we introduce a separation of programming interfaces to the TP monitor. This separation follows the open implementation approach [Kiczales, 1992], pioneered in the meta-object protocol [Kiczales et al., 1991], 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 in Figure 3.5, both the transaction demarcation interface and extended transaction interface are functional, subdivided for clarity only.

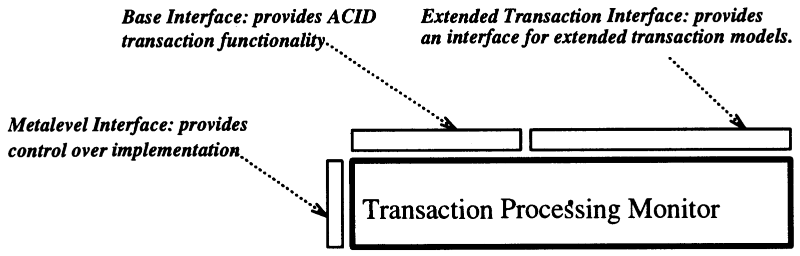
The separation of programming interfaces to the legacy TP monitor provides the means to talk about existing transaction models, and also introduce new extended transaction behaviors and interfaces. Default transaction behaviors remain available through the standard *transaction demarcation interface*. 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 *augments* the default transaction demarcation interface with new extended control operations, so the TP system programmer can perform the meta-programming of the TP monitor in a clean, concise manner that does not deviate significantly from 'normal' programming.

3.4 APPLICATIONS OF THE REFLECTIVE TRANSACTION FRAMEWORK

The Reflective Transaction Framework would not be of great value unless it supported the extended functionality required to rapidly implement a wide range of advanced transaction models and semantics-based concurrency control protocols for advanced applications. Our experience in this regard has been very positive [Barga and Pu, 1996, Barga and Pu, 1995]. In this section, we illustrate the application of the Reflective Transaction Framework to implement advanced transaction models and semantics-based concurrency control protocols. In our discussion, we outline the process of using the framework from the perspective of both TP system programmer and application developer, and briefly describe operational aspects of adapters in supporting the extended transaction functionality.

3.4.1 Implementing Advanced Transaction Models

The split/join transaction model was proposed for open-ended activities such as computer-aided design and manufacturing (CAD/CAM) [Pu et al., 1988]. Open-ended activities are characterized by uncertain duration, uncertain devel-


Base Interface: provides ACID transaction functionality.
Extended Transaction Interface: provides an interface for extended transaction models.
Metalevel Interface: provides control over implementation.
Transaction Processing Monitor

Transaction demarcation interface — presents the default transaction interface offered by the legacy TP monitor. When used alone it provides *default* transaction behavior of atomic 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 is 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`, `Join`, `Spawn`, `Create_Group`, etc.

Meta interface — allows applications to view selected aspects of the underlying TP monitor functionality and to make modifications. The meta interface provides commands for programmers to locally and incrementally adapt the functionality of the TP monitor to the requirements of an extended transaction. Some of the operations in the meta interface include: `delegateOp`, `delegateLock`, `formDependency`, `noConflict`, and `select`.

Figure 3.5 Separation of Interfaces to the Reflective Transaction Framework

opments 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.

Synthesizing the `split` function. When a transaction T_1 splits, by executing the transaction control operation `split(T2)`, 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 to NewTran
  select(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 3.6 `Split` transaction control operation.

Once the extended functionality of the `split` transaction control operation has been defined using the *meta interface*, it can then be added to the *extended transaction interface* where it will be available for applications programmers to use.

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 might use the split and join operations to dynamically restructure a transaction 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)
    select(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 *primary 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 `select` *meta interface* 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, 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 select operations, the programmer simply uses familiar transaction control operations to write the application.

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 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 `select` command in line 3 also causes control to be passed to the *Transaction Management Adapter*, which updates the transaction meta object 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(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 3.7). 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 3.7. For the execution of the split operation, as defined in Figure 3.6, 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 select 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.

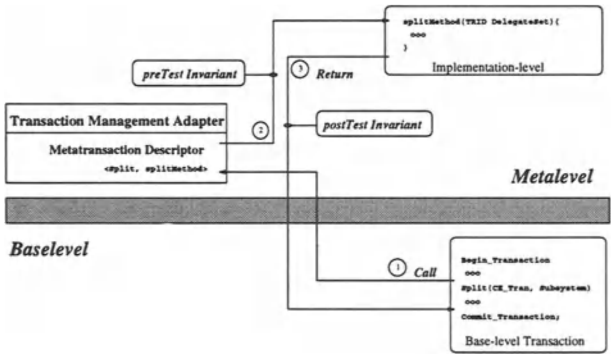


Figure 3.7 Transaction control operation redirection

3.4.2 *Implementing Semantics-Based Concurrency Control*

Concurrency control is based on a simple intuition: if the order in which two operations take place does not affect the results, then the transaction-processing system should allow different transactions to perform these *compatible* operations concurrently. Fundamental to all concurrency control protocols is the notion of *conflict* — incompatibility between operations or transactions. Most commercial transaction processing systems define conflict in terms of `read` and `write` operations [Bernstein et al., 1987] — two operations conflict if both access the same data object and one is a `write` operation. This *syntactic* definition of conflict has been criticized as being too restrictive for advanced applications where conflicts can be defined at a more abstract *semantic* level. The basis of semantics-based concurrency control (SBCC) is the introduction of a *relaxed* notion of conflict, that is typically weaker than traditional `read/write` conflict and thus allows more concurrency [Badrinath and Ramamritham, 1991, Chrysanthis and Ramamritham, 1990, Ramamrithan and Pu, 1995, Ramamritham and Chrysanthis, 1992].

The Reflective Transaction Framework provides an extensible concurrency control facility that enables individual transactions to define semantic notions of conflict, with the only limitation that it be expressible in terms of an operation compatibility table or an *explicit* relaxed conflict relationship between transactions. The compatibility table specifies actions the framework should take given certain conflicting operations, dynamic dependency relationships that are formed as a result of the conflict, and specifies conflicts between transactions that have been explicitly relaxed. An operation compatibility table has the advantage of being simple for application programmers to create, and can be loaded and efficiently tested at run-time. To illustrate the flexibility of this approach we describe how the Reflective Transaction Framework can be used to specify and implement three SBCC protocols. They are operation commutativity, operation recoverability, and transaction cooperation. The framework is not limited to this selection, rather, they were selected because they form the basis for a number of related SBCC protocols and illustrate key operational aspects of the framework.

Specifying Operation Commutativity. The simplest operation compatibility relationship used to determine if two operations can execute concurrently is *commutativity*. If two operations commute, then their effects on the state of a data object or their return values are the same, irrespective of their execution order (for example, two `read` operations commute). When a transaction invokes an operation, it can be executed if it commutes with every other uncommitted operation. Further, if the transaction processing system allows only commuting operations to execute concurrently, then it prevents cascading aborts.

The commutativity of operations on a data object is specified in advance via the operation compatibility table. As a simple example, consider operations on a bank account data object for commercial banking applications. For this data type we define 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 application developer or TP system programmer can construct an operation compatibility table, as illustrated in Table 3.1. Columns in the compatibility table represent operations currently holding a lock, while rows represent operations requesting a lock. Entries marked SOK indicate the requested operation is semantically compatible (commutes) with the concurrently executing operation, while an entry marked CON indicates the requested operation conflicts. There are no dynamic dependencies to be recorded, hence this field is left blank; a semicolon is used as the field delimiter.

Specifying Operation Recoverability. Another semantic notion proposed to relax conflicts among operations, weaker than operation commutativity, is

Table 3.1 Operation commutativity for ACCOUNT data type.

Account:COMM	`Balance`	`Deposit`	`Withdraw`
`Balance`	SOK;;	CON;;	CON;;
`Deposit`	CON;;	SOK;;	CON;;
`Withdraw`	CON;;	SOK;;	CON;;

recoverability [Badrinath and Ramamritham, 1991]. An operation O_i is *recoverable*, relative to another operation O_j, if the value returned by O_i, and hence the observable semantics of O_i, is independent of whether O_i executed immediately before O_j. Thus, if transaction T_i precedes transaction T_j, and T_i aborts then T_j is immune from cascading aborts since the operation effects on T_j remains the same.

Unlike commutativity, recoverability does not require equivalence of states for operations to execute concurrently. Hence, operation commutativity implies operation recoverability, but operation recoverability does not directly imply operation commutativity. Whenever an operation is recoverable but not commutative, relative to another concurrent operation, both operations are allowed to perform concurrently. However, a dynamic commit-dependency relation is set between the transaction that attempts to perform the operation and transactions that have already performed recoverable operations with respect to that transaction. For our example above, T_j can not commit until T_i either commits or aborts. At the time of commit, then, a transaction will have to wait until all the other transactions on which it has a commit-dependency have completed in order to maintain database consistency.

As with commutativity, operation recoverability is specified in advance using a compatibility table designed for recoverability. This is illustrated in Table 3.2 for the ACCOUNT data object, in which the commit dependencies that arise due to recoverability are specified as CD. When the Reflective Transaction Framework is evaluating an operation conflict condition between two transactions and it relaxes the conflict using recoverability semantics, the commit dependency between the two transactions will be recorded in a dependency graph. Commit dependencies that arise from recoverable operations will be tracked through the execution of the transactions and used to sequence transaction completion.

Application Programming Using SBCC Protocols. If an application developer identifies data objects that are *hot spots* hot spot or concurrency bottleneck *concurrency bottlenecks* in a system, they can construct operation compatibility tables for these data objects. Applications using the Reflective Transaction Framework can then select these compatibility tables for semantics-based transaction synchronization. To illustrate, we will continue with the

Table 3.2 Operation recoverability for ACCOUNT data type.

Account:RECV	Balance	Deposit	Withdraw
Balance	SOK;CD	SOK;CD	SOK;CD
Deposit	CON;	SOK;CD	CON;
Withdraw	CON;	SOK;CD	CON;

CAD example introduced previously, in which a team of engineers are working together to design a computer chip. During initial chip design, several component engineers would be inserting new components for the chip, performing lookups on existing components, and 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).

Table 3.3 File Log:comm, operation commutativity for COMPONENTLOG data type.

Log:comm	Insert	Delete	Lookup	Sort	Modify
Insert	SOK;;	SOK;;	SOK;;	CON;;	SOK;;
Delete	SOK;;	SOK;;	SOK;;	CON;;	SOK;;
Lookup	SOK;;	SOK;;	SOK;;	SOK;;	SOK;;
Sort	CON;;	CON;;	CON;;	SOK;;	CON;;
Modify	SOK;;	SOK;;	SOK;;	CON;;	SOK;;

Table 3.4 File Log:recv, operation recoverability for COMPONENTLOG data type.

Log:recv	Insert	Delete	Lookup	Sort	Modify
Insert	SOK;CD;	SOK;CD;	SOK;CD;	CON;;	SOK;CD;
Delete	SOK;CD;	SOK;CD;	SOK;CD;	CON;;	SOK;CD;
Lookup	SOK;CD;	SOK;CD;	SOK;CD;	CON;;	SOK;CD;
Sort	SOK;CD;	CON;;	SOK;CD;	SOK;CD;	CON;;
Modify	SOK;CD;	SOK;CD;	SOK;CD;	CON;;	SOK;CD;

The COMPONENTLOG has five operations defined: `Insert`, `Delete`, `Lookup`, `Sort`, and `Modify`. The operation `Insert` adds a new key (key, item) into the Component_Log. If the key is already in the table it will return failure; else it returns success. `Delete` removes the pair with the given key from the Component_Log. If the key is not present it will return failure; else it returns success. The `Sort` operation sorts the entries in ascending order. `Lookup` returns the value of the item associated with a given key if it exists in the

Component_Log; else it returns failure. `Modify` will replace the current value of the item with the new value for the given key. Tables 3.3 and 3.4 illustrate the commutativity and recoverability properties of the operations performed on data objects of type COMPONENTLOG. For simplicity, it is assumed that transactions will operate concurrently on different parameters (keys) on the objects of type COMPONENTLOG. These operation compatibility tables would be entered into individual files, either using a simple text editor or a graphical utility provided for formatting compatibility tables.

The code fragment below outlines how an application programmer might use these compatibility tables for semantics based concurrency control, and illustrates the use of the framework to permit explicit transaction cooperation.

```
Begin_Transaction CE_Tran                         (1)
begin
    instantiate(CE_Tran)                          (2)
    select(CE_Tran, Conflict, Log:comm)           (3)
    select(CE_Tran, Conflict, Log:recv)           (4)
    Lookup(CID_87, compspec)                      (5)
    ...
    ...{ data manipulation }
    Modify(CID_87, compspec)                      (6)
    ...{ data manipulation }
    Insert(CID_109, nullspec)                     (7)
    ...{ data manipulation }
    NoConflict(QA_Tran,CID_109)                   (8)
    ...{ data manipulation }
    ...
    Modify(CID_109, compspec)                     (9)
end
Commit_Transaction {CE_Tran}                      (10)
```

The `Begin_Transaction` command in line 1 declares the beginning of the component engineer's transaction, and together with the `Commit_Transaction` in line 10 brackets the transaction. The command `instantiate` in Line 2 *registers* the transaction with the Reflective Transaction Framework. The `select` meta interface command in line 3 indicates the transactions intention to use semantic information to relax lock conflicts, and specifies the compatibility table LOG:COMM is to be used (a file pathname could also be supplied). The `select` meta interface command in line 4 specifies an additional compatibility table LOG:RECV is to be used to relax conflicts; the order in which compatibility tables are selected using the `select` command will determine the order which they are applied to relax lock conflicts.

If a syntactic conflict (R/W) is detected during transaction execution, the TP monitor will raise a lock conflict event and the conflict adapter will be invoked for semantic conflict testing. For example, if an uncommitted transaction has performed a Lookup operation (a read-typed operation) on the COMPONENT-LOG data object and transaction CE_Tran requests to perform a Modify operation (a write-typed operation) in line 6, the TP monitor would detect a syntactic conflict. Since the conflict adapter registered a handler for the event, and transaction CE_Tran has selected a commutativity table to relax lock conflicts (Table 3.3), the framework will perform 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 will grant the lock and increment the counter of lock holders, enabling both transactions to own the lock.

In summary, if an application programmer wishes to use semantics-based concurrency control for transaction synchronization, they first create compatibility tables for data objects that have been identified a hot spots or concurrency bottlenecks. To use available compatibility tables, an application will then *register* the transaction with Reflective Transaction Framework and then select from the available semantic compatibility tables. During execution, the Reflective Transaction Framework will allow 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 will record and track the 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 meta interface command `instantiate` in line 2 performs the same initialization of the adapters as the previous advanced transaction model example. The `select` command in line 3 and in line 4 performs two functions. First, it informs the framework of the transactions intension to utilize semantic information to relax lock conflicts, and *Transaction Management Adapter* responds by registering the *Conflict Adapter* as the handler for lock conflict events. Second, it instructs the *Conflict Adapter* to load the specified compatibility tables for the transaction; if the file can not 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 will be handled by the *Conflict Adapter*.

During transaction execution, the `Lock` function of the underlying TP monitor performs usual `Read / Write` conflict testing for all lock requests. If a lock conflict is detected, an event is raised. Information passed to the conflict adapter includes the identifier of the transaction requesting the lock, 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, performed only when a conflict is detected, is executed by the `relaxConflict` function to perform semantic compatibility testing to determine if the two operations are semantically compatible. The function `relaxConflict` relaxes conflicts between transactions by two means: compatibility table(s) defining conflict relationships between operations, and a no_conflict table that records all conflicts explicitly *relaxed* between transactions. Using these two sources of information, `relaxConflict` implements the following rule to determine whether there is a conflict between two transactions:

A conflict detected by the TP monitor can be relaxed if either of the following conditions hold true:

1. the semantics of the data object indicate that the operation for which the lock is being requested is compatible with all uncommitted operations holding a lock in an incompatible mode;
2. the 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;

The relaxed conflict rule effectively states that a 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. The generality of this relaxed conflict rule allows the conflict adapter to selectively present and change the definition of conflict for one or more underlying data objects or transactions. This is illustrated in Figure 3.8.

When a inter-transaction dependency directive, such as a commit dependency CD, is found in an operation compatibility table, the conflict adapter records the dependency in the transaction dependency graph TRAND using the Transaction Adapter command `form-dependency`. Checks are performed to prevent dependency cycles from being formed. During transaction termination the *Transaction Management Adapter* procedures `PreCommit` and `PreAbort` take the necessary actions to ensure that all requisite transactions have completed (the transactions have either committed or aborted), and all pending transactions are notified that PE_Tran has completed.

By utilizing these commands to adapt the definition of conflict offered by the underlying TP system, the conflict adapter is able to implement a variety of semantics-based concurrency control protocols discussed in the literature [Barga et al., 1994]. This semantics based concurrency control is all performed through extensions to the underlying conflict detection and locking performed by the TP monitor, demonstrating that the use of a conventional

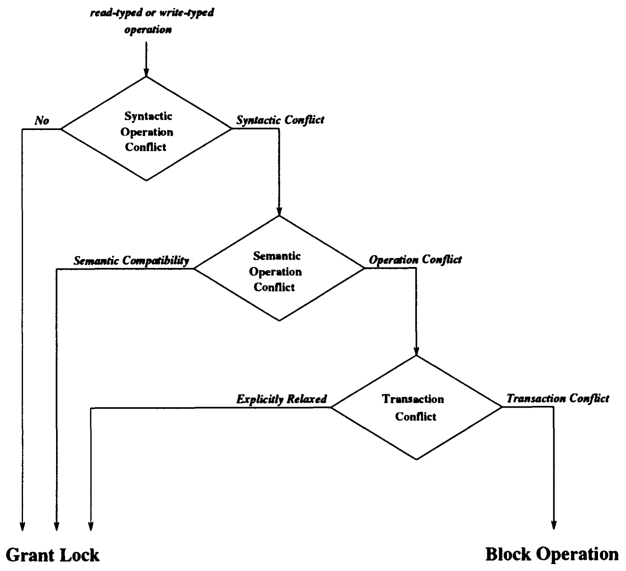


Figure 3.8 Transaction control operation redirection

locking mechanism does not preclude the use of semantics-based concurrency control protocols.

3.5 CONCLUSION

We have introduced the Reflective Transaction Framework as a practical method to implement extended transactions on conventional TP monitors. We described how the framework achieves an open implementation of the TP monitor, so that applications have access to and control over the underlying functionality of the TP in a way that allows the programmer to tailor extended transactions to the needs of a particular application. Access to TP monitor system functionality and extended transaction behaviors is principled in the sense that the meta level interface and extended transaction interface allow access to this functionality without forcing the TP monitor to expose the internal data structures and functions that are actually used. This independence from actual implementation allows intercession guards and runtime checks to be performed. The framework does not expose the entire TP monitor system functionality, but only selected aspects of it. The TP systems programmer only needs to go as far as application developers require. If only certain advanced transaction models or semantics-based concurrency control protocols are required, only those extended transaction behaviors need be provided; other extended transaction behaviores can be incrementally added to the framework over time.

The implementation of the Reflective Transaction Framework is based on transaction adapters, reflective software modules built on top of TP monitor functional components. Transaction adapters use events to reify extended transaction state and selected aspects of the TP monitor into distinct meta-level objects, and use the existing application programming interface to reflect changes to the computational state of the TP monitor. Extensions provided by the transaction adapters build on the available functionality of the TP monitor, to the extent possible, and provide the programmer with a clean meta interface through which they can customize and extend the system functionality. This allows extensions and model improvements to be quickly incorporated, and as a result, the framework can remain up to date with application requirements.

The Reflective Transaction Framework provides a flexible foundation for implementing application-specific extended transactions. We have applied the framework to implement a wide range of advanced transaction models [Barga and Pu, 1995], including split transactions [Pu et al., 1988], cooperative group transactions [Nodine and Zdonik, 1990], and Sagas [Chrysanthis and Ramamritham, 1992], and a number of semantics-based concurrency control protocols [Barga and Pu, 1996], including commutativity [Weihl, 1988a], recoverability [Badrinath and Ramamritham, 1991], cooperative serializability [Ramamritham and Chrysanthis, 1992], and epsilon-serializability [Ramamrithan and Pu, 1995]. We have also used it to incrementally develop new advanced transaction models, building on models previously added to the framework, such as the cooperative-split model which combines cooperative group transactions with split transactions.

It is our hope the Reflective Transaction Framework will provide a clear migration path to incorporate research advances in transaction processing into real, working environments where they can be applied. We have implemented a proof-of-concept prototype of the framework on production transaction processing software, namely the Encina Toolkit [Encina, 1993]. The Encina Toolkit has been used to construct several modern distributed TP monitors, including IBM's CICS/6000, DEC's ACMS/xp, and the Encina TP monitor. As such, our Encina implementation of the Reflective Transaction Framework can be used with any of these commercial TP monitors for experimenting with extended transactions. Our implementation on Encina was clearly facilitated by an available event callback mechanism and open API to the transaction services of the toolkit. A valid question is whether the additional work of exposing the API and adding an event mechanism to other transaction processing systems would be worthwhile. The answer to this is in part economical. There are only a handful of commercially significant TP monitors in circulation, most of which offer only conventional database transactions. This compares to thousands of 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 transaction processing systems software to enable a system, such as the Reflective Transaction Framework, to widen their application reach and make application development easier should yield a large payoff.

Acknowledgments

Support for this research is provided in part by an Intel Graduate Fellowship from the Intel Foundation.

III Long Transactions and Semantics

4 FLEXIBLE COMMIT PROTOCOLS FOR ADVANCED TRANSACTION PROCESSING

Luigi Mancini, Indrajit Ray,
Sushil Jajodia and Elisa Bertino

Abstract: Although numerous extended models have been proposed to overcome the limitations of the standard transaction model, most of these models have been proposed with specific applications in mind and, therefore, they fail to support applications with slightly different requirements. In this paper, we propose the H-transaction framework along with a set of powerful transaction control primitives to support a wide range of transaction dependencies including flexible commit. Our set of transaction control primitives can be broadly classified into two types: basic primitives that are found in almost all conventional transaction processing systems and new primitives that lend expressive power and flexibility. These primitives can be used to separate the coding of the transactions from the application's control aspects needed for preserving cooperation and dependencies among transactions. The reason behind this separation is to simplify the work of the programmer since transactions can be coded without worrying about managing concurrent computations, communications, etc. We show that our primitives have expressive power to support a number of extended transaction models including nested transactions, sagas, workflows and contingent transactions. Moreover, our primitives allow the programmers to define their own primitives — having well-defined interfaces — so that application specific transaction models such as the distributed multilevel secure transactions can also be supported.

4.1 INTRODUCTION

The classical transaction model for managing database systems has been an immense success, both theoretically and commercially. Nonetheless, there has long been recognition (see for example [Gray, 1981, Elmagarmid, 1992, Korth, 1995]) that the standard model is too restrictive for many advanced database applications. For example, in a cooperative environment, if long-duration activities are executed as atomic transactions, they may significantly delay the execution of shorter activities. In the case of multidatabase systems, the autonomy requirements of the component local databases are in direct conflict with the atomicity property of classical transactions. Consequently, in recent years, a number of works have attempted to extend the traditional atomic transaction model to support more flexible transaction processing. Examples of such models are nested transactions [Moss, 1985], Sagas [Garcia-Molina and Salem, 1987], ConTract [Reuter, 1989], ACTA [Chrysanthis and Ramamritham, 1990], Flex [Bukhres et al., 1993], DOMS [Georgakopoulos et al., 1994], and Asset [Biliris et al., 1994].

A crucial limitation of many of these extended transactions models (e.g., [Moss, 1985, Garcia-Molina and Salem, 1987, Reuter, 1989]) is that they have been proposed with specific applications in mind, which seriously limits the flexibility of these models. A specific model may be provided by the system but the user cannot specify which one. Moreover, if an application has needs with slightly different requirements, they lack the necessary expressive power to model these applications. For example, the nested transaction model is most suitable in applications that have a hierarchical structure with a good degree of internal parallelism. The Saga model is useful only when the subtransactions are relatively independent and each subtransaction can be successfully compensated. The ConTract model is also based on rigid compensation policies for transactions.

ACTA, DOMS, and Flex provide formal frameworks to express the properties of extended transactions and dependencies among them. ACTA [Chrysanthis and Ramamritham, 1990] classifies these dependencies into two broad categories based on a transaction's effect on the commit and abort of other transactions and on the data items it accesses. Although ACTA is able to specify a wide variety of transaction models, it fails to capture transaction dependencies which arise due to events other than commit or abort of the transactions. Examples of such events are various error conditions which do not influence the commit or abort of transactions but which nonetheless need to be addressed. The secure dependencies present among subtransactions of a multilevel secure distributed transaction [Jajodia and McCollum, 1993, Jajodia et al., 1994] is another example. The DOMS project's [Georgakopoulos et al., 1994] transaction model provides a specification language similar to ACTA and, therefore, suffers from similar shortcomings as ACTA.

Flex [Bukhres et al., 1993] is a transaction specification model that offers flexibility by providing primitives for specifying dependencies between transactions. The specifiable dependencies can be broadly categorized into two types, those that define the execution order on the subtransactions of a Flex transaction (i.e., commit/abort dependencies) and those that define the dependencies of subtransactions on events not belonging to the transaction. However, Flex does not allow a programmer to specify the communication (i.e., synchronization) between parallelly executing transactions. Also, a Flex transaction specification cannot include any information about how to compensate subtransactions.

Asset [Biliris et al., 1994] is different from other works in that it provides ACTA based language primitives for specifying dependencies between a set of concurrent, cooperating transactions. These primitives allow the programmer to define custom transaction semantics to match the needs of the specific application and are general enough to be incorporated in any database system. However, even with these flexible primitives, Asset, like ACTA, cannot implement transaction dependencies that arise due to events other than commit or abort of transactions or data sharing among them. It does not offer an experienced programmer the flexibility to alter the commit protocol so as to provide a more versatile commit facility, while at the same time retaining simple default interfaces for the naive user. Such a feature seems useful in many situations.

A more recent work, that by Barga and Pu [Barga and Pu, 1995], proposes the reflective transaction framework as a practical and modular method to implement extended transaction models. This work provides the flexbility of language primitives to construct extended transactions. However it does not allow the programmer to specify synchronization between parallely executing transactions; hence the framework's suitability for designing extended transactions that execute in a distributed or multidatabase setting is limited.

In this paper we propose the H-transaction model alongwith a set of language primitives that allow programmers to implement a large number of transaction dependencies including flexible commit. The dependencies that can be implemented in our model include the commit/abort dependencies that can be specified by the ACTA framework and those present in the multilevel secure transaction model. Our work focuses on transaction control at the programming language level, and proposes a linguistic construct that separates the coding of the transaction from the definition of the application's control flow. All control aspects needed for transaction cooperation and dependencies are coded separately. Transactions can be thus coded without worrying about managing concurrent computations, communications, etc. This simplifies the work of the programmer and also increases code reusability.

The programmer can use our primitives directly as part of a programming language to specify various commit and abort dependencies among transaction and to realize relaxed correctness criteria to satisfy their specific application needs. Alternatively, the programmer can use a higher level declarative language to specify the dependencies among transactions. In this case, a precompiler can automatically translate the higher level description of the dependencies into our primitives. Further our primitives allow the programmer to define their own custom primitives having well-defined interfaces. (An example of this is the noSignalServiced primitive shown later in section 4.5.4) This feature adds to the flexibility of our model by allowing application specific transaction models to be supported.

The remainder of this paper is organized as follows. We describe the H-transaction model in section 4.2. Section 4.3 discusses how flexible transaction dependencies can be specified in our model followed by a description of our primitives in section 4.4. In section 4.5, we show that our primitives can support not only various extended transaction models but models that are customized to meet the specific application needs as well. Section 4.6 concludes with a brief discussion of our future work.

4.2 OVERVIEW OF OUR APPROACH

4.2.1 The System Architecture

A *transaction* T_i in our model is defined to be any sequence of operations on data items (both persistent and volatile) delimited by either the Begin_Trans(T_i) ... End_Trans(T_i) pair or the Begin_Trans(T_i) ... Abort_Trans(T_i) pair. A transaction is written in a high level language supporting persistence and the new transaction processing primitives that we introduce.

An *H-transaction* is composed of a set of such transactions and includes a definition of a set of dependencies among these transactions; this set of dependencies includes, but is not limited to, the commit or abort relationships among the component transactions. The programmer is able to specify different relationships among the component transactions by defining a *coordinate block* in the H-transaction that describes these relationships. The coordinate block can be either a program fragment in the high level language or it may be a declarative description of the transaction dependencies. In section 4.3 we show how coordinate blocks can be specified as a program fragment in the high level language.

Basic transaction processing is achieved at every site by the cooperation of Transaction Manager, the Log Manager, the Lock Manager and the Resource Manager. These components together form what is known as the Transaction Processing (TP) subsystem at the particular site and ensures the atomicity, consistency, isolation, and durability (ACID) properties [Gray and Reuter, 1993] of the transactions executing at that site. The TP subsystem implements the basic transaction control operations like commit, abort, savework, rollback, begin-transaction, lock data items etc.

On top of the TP subsystem at every site, we assume that there is a *Transaction Management Adapter* (TMA) module that enhances the functionality of the underlying TP subsystem by implementing an extended interface of the TP system for our new transaction processing primitives. The notion of Transaction Management Adapter is borrowed from [Barga and Pu, 1995] where the authors propose also a lock adapter and a conflict adapter as add-on modules on top of an existing TP system to enhance the TP system's functionality. A discussion on these other adapters is beyond the scope of this work (although we allow these adapters in our architecture), as we focus mostly on transaction termination dependencies.

A transaction T_i executing at some site interacts with the TMA-TP module at that site via a *coordinator module* (CM). This coordinator module acts as a transaction event handler and implements among other things the coordinate block of the H-transaction of which T_i is a part. Before transaction T_i gets executed its CM is started as a set of concurrently executing threads. Execution of a transaction primitive by a transaction T_i is the transaction event that causes the CM corresponding to T_i to react and handle the event.[1] If a thread with the same name as the event is defined within the CM then the thread gets activated otherwise the CM lets the underlying TMA-TP module handle the event as appropriate. The executing threads can in turn invoke other primitives that are part of the TMA-TP module in order to actually handle the event. The coordinator module can be viewed as an extended form of the notion of *metatransaction* of [Barga and Pu, 1995] to include executing codes and a mechanism to specify and handle inter-transaction communication and synchronization.

Specifically a CM can be divided into two distinct parts: A required set of compiler-generated *event interceptors* and an optional set of programmer defined *event handler*. The latter is essentially the programmer defined coordinate block of the H-transaction. The set of event interceptors includes: (1) the mechanism to pass on relevant parameters from the run-time environment to the other system modules and vice versa, and (2) the information as to how a particular event is to be handled, i.e. whether by a programmer defined event handler or by the underlying TMA-TP module. In particular, the set of event interceptors contains mechanism to communicate with other CMs of the same H-transactions and to pass on parameters to these CMs. An event interceptor is awakened by the occurrence of an event and then either invokes one of the threads in the CM or invokes an action exported by the TMA-TP module. Finally, the CM may contain a set of invariants for each component transaction. These invariants constitute the predicates that need to be satisfied before and after a transaction execution.

In a distributed setup the CM at the originating (coordinating) site of an H-transaction is of the form just described. At remote sites where component transactions get executed, lightweight CMs are created. The lightweight CMs contain only the compiler generated event interceptors. Their function is to invoke the relevant threads executing at the CM of the coordinating site or at the TMA-TP module at the local site. They act as the interface between the TMA-TP at the remote site and the CM at the coordinating site. If the programmer does not explicitly specify any coordinate block (i.e. the programmer has not defined any thread to handle transaction events) then such lightweight CMs get loaded at every site and act as forwarding agents for transaction events to the TMA-TP module at each site. The CM at the coordinating site executes the programmer defined code to perform the coordinating operations, as for example the decision to commit an H-transaction, commit some components of the H-transaction while aborting other components or taking some other action. If no component transaction of the H-transaction is executing at the coordinating site, the TMA-TP subsystem at this site is responsible only for the housekeeping functions (e.g., writing log records etc.) for the H-transaction as a whole and for its components, while the TMA-TP subsystems at the remote sites execute the component transactions as well as perform housekeeping (only for the component transaction executing at that site).

An H-transaction submitted by the user to the transaction processing system at a particular site (the coordinating site) is executed as follows:

1. If the programmer has specified a coordinate block with the H-transaction then
   (a) When the H-transaction gets initiated, create a coordinator module at the coordinating site. This CM has two parts - the event interceptor part and the event handler.
   (b) Create a table R in the event interceptor that maps events to the respective handlers located in either the local TMA-TP system or in the programmer defined primitives.
   (c) If the H-transaction consists of component transactions that are to be executed at remote sites then spawn a lightweight CM at each of these sites to contain only the mechanism to communicate with other CMs and a copy of the table R.
2. If the programmer has not specified any coordinate block then
   (a) Create a lightweight CM at the coordinating site to pass on the event that has occurred to the underlying TMA-TP module.
   (b) If it is a distributed setup, spawn similar lightweight CMs at the remote sites too.
3. When a transaction event occurs for some transaction T_i (that is T_i executed some transaction primitive), the event interceptor in the CM associated with T_i is awakened.

(a) If the event has been the execution of some transaction primitive defined by the programmer then the event interceptor first checks the invariants, if any, that constitute the precondition for the primitive. If these invariants are satisfied then the CM invokes the thread with the same name executing within the CM at the coordinating site. Note that prior to or alongwith invoking a thread at the coordinating site's CM, the event interceptor may also invoke actions at the local TMA-TP subsystems.

(b) The thread at the coordinating site's CM gets activated, performs the actions defined by its code and returns control to the to the event interceptor of the CM associated with T_i.

(c) The initiating CM checks for postcondition satisfaction and depending on the outcome of this test, returns the result of the thread execution to the transaction as if for a normal transaction primitive call. Note that before the result is returned, the CM may invoke any function at the local TMA-TP subsystem.

(d) If the event has been the execution of some primitive not defined by the programmer, then the event interceptor allows the local TMA-TP subsystem to handle the event appropriately.

4. The decision to end an executing H-transaction comes from the CM at the coordinating site. When this happens the different CMs at the various site all terminate and control gets returned to the TMA-TP subsystem at the coordinating site.

Figure 4.1 gives a schematic diagram on how transaction events in a remote transaction are handled by the cooperation of the lightweight CM at the remote site, the CM at the coordinating site and the TMA-TP subsystems at both the sites. In the figure the begin_trans event is handled as a local TP system call by the TMA-TP subsystem at the remote site; the end_trans event is intercepted by the lightweight CM at the remote site and forwarded to the CM at the coordinating site. The latter in turn invokes a TP system call at its local TMA-TP subsystem. We have specifically left out the semantics of the two different events here.

Our model allows the programmer to define not only application specific transaction events (an example of which will be given later on in section 4.5.4), but also to redefine with ease the semantics of ordinary transaction events such as transaction completion or transaction begin, commit and abort. The programmer defined behavior get precedence over the default behavior and can thus be imposed on the latter.

In the following we give an example to illustrate the execution model for a programmer defined coordinator.

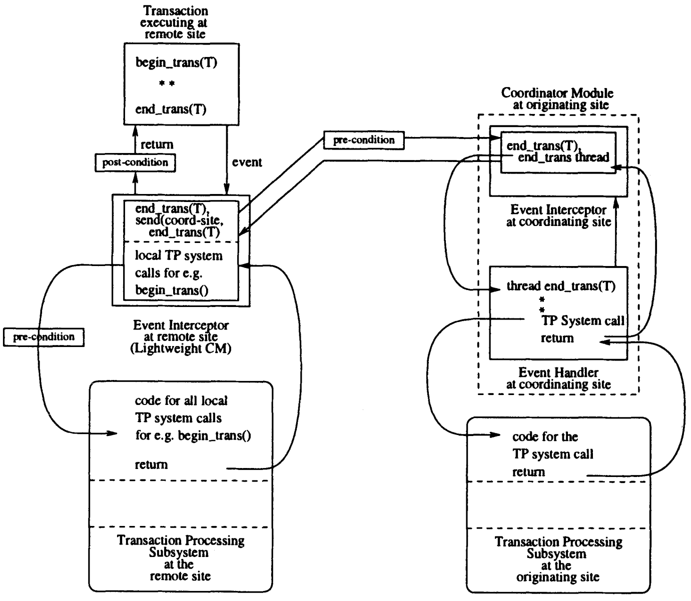

Transaction executing at remote site
begin_trans(T)
end_trans(T)
return
post-condition
event
end_trans(T), send(coord-site, end_trans(T))
local TP system calls for e.g. begin_trans()
pre-condition
Event Interceptor at remote site (Lightweight CM)
code for all local TP system calls for e.g. begin_trans()
return
Transaction Processing Subsystem at the remote site
pre-condition
Coordinator Module at originating site
end_trans(T), end_trans thread
Event Interceptor at coordinating site
thread end_trans(T)
TP System call
return
Event Handler at coordinating site
code for the TP system call
return
Transaction Processing Subsystem at the originating site

Figure 4.1 Event Handling Sequence for Transaction Events

4.2.2 Illustrative Example

Figure 4.2, illustrates the execution of an H-transaction that involves multiple sites. A programmer wants to coordinate the two component transactions T_1 and T_2 such that either T_1 commits or T_2 commits, but not both. Note that T_1 and T_2 may both abort. We assume that for the purpose of commit or abort of transactions the TMA-TP at various sites rely on the commit protocol of the underlying TP system. In particular for the current discussion we assume that the commit protocol used at all sites is Early Prepare (EP) [Stamos and Cristian, 1993]. Figure 4.2 shows how the user's H-transaction, the TMAs and the CMs interface with the TP systems. Directed solid arrows represent the interaction between the different components of the systems.

When the user submits an H-transaction at Site_u, the coordinating code specified in the H-transaction is loaded as the coordinator module CM_u. The TMA_u submits each of the component transactions T_1 and T_2 to the TMAs at Site_1 and Site_2. These remote TMAs in turn loads the respective lightweight CMs (which have a much lesser functionality than CM_u) and then request their underlying TP systems to execute the component transactions.

When T_1 completes, it invokes an end_trans operation that prompts the occurrence of the event. This invocation awakens the event interceptor in CM_1 which asks TMA_1 to prepare to commit T_1. The TP system at Site_1 forces a prepare log record and sends an acknowledgement to CM_1. At this point

CM_1 sends an end_trans message to CM_u. Note that this message can be seen as the yes vote sent by participants when they are ready to commit in the EP commit protocol. CM_u in turn decides to commit T_1 and abort T_2 and accordingly informs its TMA. TMA_u asks the TP subsystem at the coordinating site to force a commit record for T_1, an abort record for T_2 and a commit record for the H-transaction, and then acknowledges to CM_u. After this CM_u sends a commit(T_1) message to CM_1 at $Site_1$ and an abort(T_2) message to CM_2 at $Site_2$. CM_1 will cause TMA_1 to invoke commit(T_1) at its TP system while CM_2 will cause TMA_2 to invoke abort(T_2). The TP system at $Site_1$ writes a commit record for T_1 and forgets about T_1, while the TP system at $Site_2$ writes an abort record for T_2 and forgets about it.

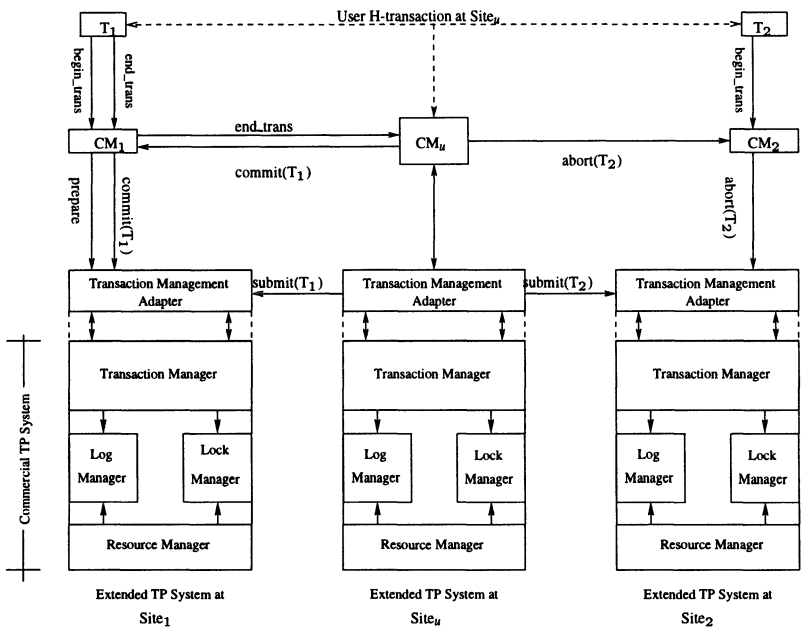


Figure 4.2 Execution model for a programmer defined coordinator

Note that the above scenario represents the execution sequence when T_1 finishes before T_2. If T_2 were to complete first, T_2 would have committed and T_1 would have aborted.

4.3 AN EXAMPLE OF TRANSACTION DEPENDENCIES

We offer the programmer two methods for specifying transaction dependencies. With the first approach the programmer directly uses the high level language and our new transaction processing primitives to specify the dependencies. With the second approach the programmer uses a declarative language in a notation we have proposed to specify the dependencies among transactions. A pre-compiler then translates this higher level description into a corresponding code in the high level language we use to specify transactions. The first approach gives more expressive power to the programmer than the second one and hence we will concentrate mostly on this approach; on the other hand the second approach is easier to use. In the following we show by an example, how the programmer can express these dependencies in our model to define coordinate blocks for a group of transactions using the high level language.

We assume that the programmer wants to design an H-transaction consisting of four component transactions T_1, T_2, T_3 and T_4 which will be executed at different sites. The application requires that at most one of T_1 or T_2 commits with T_1 being preferred to T_2 and either both T_3 and T_4 commit or none of them. In short, one and only one of the following sets of transactions commits: $\{\}$, $\{T_1\}$, $\{T_2\}$, $\{T_3, T_4\}$, $\{T_1, T_3, T_4\}$ or $\{T_2, T_3, T_4\}$. By using our primitives the programmer will be developing the program fragment shown in figure 4.3 for this application. Note that although we use some of our new language primitives before they have been presented in the paper, a detailed understanding of the primitives is not required at this stage.

From the program fragment, we find that the H-transaction consists of two coordinate blocks specified by the two *coordinate ... using* delimiters. Each block contains code that implements the dependencies between those transactions that are defined within the blocks. In the figure, the coordinate block *coordinate ... using ... end* implements the dependency between transactions T_1 and T_2 (viz. only one of T_1 or T_2 can commit with T_1 being preferred) while the block *coordinate ... using default* implements the dependency among T_3 and T_4 (viz. either both commit or none do). Note that the latter commit dependency is the standard commit dependency implemented in the various commit protocols (like Early Prepare). We assume that each transaction processing system implements a default commit protocol. The second coordinate block in the example in figure 4.3 specifies "default" as the coordinator module for transactions T_3 and T_4.

Of interest to this discussion is the coordinate block for transactions T_1 and T_2 specified by the programmer in the form of a program fragment within the sub-block *using ... end.* This program fragment implements the CM for T_1 and T_2 and contains definitions of some of the primitives that the programmer invokes within the transactions.

The transactions T_1 and T_2 are defined sequentially within the H-transaction (and not within a *cobegin ... coend* block which would have implied parallel execution) with T_1 being defined before T_2. The sequential definition of transactions naturally entails a precedence relation between these two transactions. Each transaction must be initiated by the initiate primitive before being able to

```
void example()
{
        coordinate
        initiate(T1,T2) ;
            begin_trans (T1)
                 :
            end_trans (T1);
            begin_trans (T2)
                 :
            end_trans (T2);
        using
                                    thread end_trans (M) {
                                         if M == T1 then
                                              { commit (T1); abort (T2); exit; }
                                         if M == T2 then
                                              { commit (T2); abort (T1); exit; }
                                    }
                                    thread abort_trans (M) {
                                         if M == T2 then
                                         { abort (T1,T2) exit; }
                                    }
        end;
        coordinate
            initiate(T3,T4) ;
        cobegin
            begin_trans (T3)
                 :
            end_trans (T3);
            begin_trans (T4)
                 :
            end_trans (T4);
        coend
        using default
}
```

Figure 4.3 Program in the high level language for an H-transaction consisting of four transactions

start its execution. After a transaction is initiated, it is assigned a transaction identifier in the system and an environment is set up for its execution.

After the H-transaction is submitted to the system, the TMA-TP module at the coordinating site assigns transaction identifiers to the H-transaction as well as its components and then loads the CM for the coordinating site. The TMA-TP module at the coordinating site then submits transaction T_1 to the remote site's TMA-TP. The remote TMA-TP module establishes the remote lightweight coordinator to execute on top of itself and then begins to execute T_1. Note that the CMs as a unit represent the interface to the TMA-TP systems for an H-transaction.

Suppose T_1 executes an end_trans; the CM at T_1's site sends a prepare-to-commit T_1 message to its underlying TMA-TP module and then invokes end_trans at the coordinating site's CM. (In figure 4.3 end_trans has been defined by the programmer in the coordinate block.) The CM at the coordinating site asks the CM for T_1 to invokes commit(T_1) and asks the CM for T_2 to invoke abort(T_2) at the respective underlying TMA-TP. The respective TMA-TP module consequently force a commit log record for T_1 and an abort log record for T_2 and acknowledges the respective CMs. Also the TMA-TP module at the coordinating site forces appropriate log records for T_1 and T_2. Note that the TMA-TP module can force an abort log record for T_2 although T_2 was never submitted to a remote site for execution. This is because when the H-transaction was submitted, the TMA established a local identifier for T_2. This causes T_1 and T_2 to terminate.

T_1 may alternatively execute an abort_trans command during its execution. This abort_trans command may have been invoked explicitly by T_1 or it may have been invoked by the TMA-TP because T_1 could not successfully complete. If the TMA-TP module at the remote site aborts T_1, the CM at the remote site informs the CM at the coordinating site by sending an abort_trans message to the CM at the coordinating site that T_1 has aborted. On the other hand if T_1 invokes abort_trans, the CM at the remote site forwards the invocation of abort_trans by T_1 to the CM of the coordinating site. The CM at the coordinating site executes abort_trans according to the implementation specified by the programmer in the thread abort_trans. The thread returns without executing any explicit abort (or commit) command and the coordinating CM does not send any specific instructions back to the CM at the remote site (it merely returns). As a result T_1 stops its execution but remains alive in the system until an explicit abort comes from the coordinator module to terminate it.[2] The TMA-TP module at the coordinating site now submits T_2 for execution at a remote site. As before a CM will be created at the remote site for T_2. Subsequently invocation of end_trans or abort_trans by T_2 will be trapped by the CM at the remote site and forwarded to the CM at the coordinating site for execution. If T_2 executes end_trans, the corresponding thread will commit T_2 and abort T_1. If, on the other hand, T_2 executes abort_trans, both T_1 and T_2 will be aborted. This will terminate the CMs for T_1 and T_2. Note that for the description of the implementation of the coordinator for T_1 and T_2 in figure 4.3 we have assumed that CM_u in figure 4.2 submits T_1 and T_2 sequentially to each of the respective TMAs. Moreover CM_u does not submit T_2 if T_1 commits.

Once the CM for T_1 and T_2 terminates, the control is transferred to the next step in the program. The TMA-TP module at the coordinating invokes the default coordinator in the system (i.e. the default commit protocol) for T_3 and T_4. T_3 and T_4 proceed concurrently in the system as they are within a *cobegin* ... *coend* block. When T_3 and T_4 commits the execution H-transaction is over.

Note that in any coordinate block, we can refer to only those transaction identifiers that are in the scope of the block. In figure 4.3, transactions T_1 and

T_2 are scoped in the coordinate block defined by the programmer but not T_3 and T_4. Consequently we can define end_trans and abort_trans only for T_1 and T_2 within this block.

4.4 PRIMITIVES FOR FLEXIBLE COMMIT

We now describe our transaction processing primitives. These primitives are broadly classified into two types: basic primitives and new primitives. The basic primitives are so named because their semantic counterparts are found in almost all conventional transaction processing systems. The new primitives are the ones we define and that lend expressive power and flexibility to our model. These primitives are essentially control primitives which modify the state of the transaction. There are six possible states. A transaction which has been submitted to the system, but has not yet started its execution is in the *initial* state. While executing its code, the transaction is in the *running* state. After it has executed all its code (either successfully or unsuccessfully), the transaction moves to the *completed* state. From the completed state the transaction terminates by moving either to the *committed* state or to the *aborted* state. From any of these five states a transaction can enter a sixth state - the *error* state. In this case the transaction can either execute an error handler (if provided by the system or by the programmer as part of the transaction) or return to its previous state and then continue execution from the next instruction in the transaction's code (possibly generating an error message in the process).

4.4.1 Basic Primitives

initiate($T_1, \ldots, T_n$) This primitive initiates the transactions $T_1, \ldots T_n$. It returns new transaction identifiers in the variables $T_1, \ldots T_n$ and sets up the environment necessary for the execution of the transactions. The transactions are started by calling the begin_trans() primitive. The scope of the variables T_i used in an initiate primitive is the program block containing this initiate primitive. The initiate(T_i) primitive must precede all use of the variable T_i within an H-transaction.

begin_trans(T_i) This primitive starts the execution of the transaction whose transaction identifier is T_i. This primitive can be redefined by the programmer.

sid = savework() The savework() primitive is used to establish a savepoint in the transaction execution. The invocation of this primitive causes the system to save the current state of processing. Each transaction manager writes a savepoint record on the local transaction log, while the current values of any local variables are saved on the volatile memory. The savework call returns a handle which is assigned to the identifier sid (called a *savepoint identifier*). This identifier can be used subsequently to refer to that savepoint, and in particular to the state of the system when this savepoint was established. The scope of the binding between the savepoint and the identifier sid is the block in which the "sid = savework()" is executed. Control can jump from inside of a block to a savepoint within an encompassing block, but not the other way round.

rollback(sid) This primitive takes as a parameter the identifier of a previously established savepoint and reestablishes (or returns to) the savepoint. More precisely, when the rollback(sid) function is invoked, it restores the state of the system to the state that existed when the savepoint denoted by the savepoint identifier sid, was established; the execution of the transaction then continues from the statements that follow the savepoint sid. The successful termination of the rollback primitive is indicated by the restoration of the savepoint denoted by sid. This primitive can only be invoked within a transaction code.

restart(T_i) This primitive is a part of the coordinator module and cannot be invoked by a transaction. When called, this primitive starts the execution of the transaction whose identifier is T_i. If the transaction T_i was previously executed (partially or fully), then all changes effected by T_i are discarded before the transaction execution is restarted.

commit($T_1, \ldots, T_n$) This primitive is implemented in the TMA-TP module and is part of the commit protocol. It cannot be invoked directly by a component transaction. Rather, it has to be invoked by the coordinator module. This primitive commits the operations of the transactions which are its parameters, by first writing the log records and then communicating the commit decision to the transaction managers of these transactions. In other words, this command forms the final phase of any commit protocol between the coordinator and the transactions $T_1, \ldots, T_n$.

abort($T_1, \ldots, T_n$) This primitive aborts the transactions specified as parameters. If the primitive abort(T_i) is invoked before T_i has started its execution, then T_i never starts its execution and is discarded from the system. Like the commit primitive, abort is a part of the TMA-TP module and can be invoked only by the coordinator module.

cobegin ... coend These two primitives act as bracketing constructs for specifying concurrently executing transactions. Control flow does not proceed beyond the cobegin ... coend block until all of the transactions created by the block complete. Cobegin ... coend can be nested.

4.4.2 New Primitives

end_trans(T_i)< *support_code* > The end_trans is a system defined primitive which can be redefined by the programmer as a coordinator module thread.

Its execution signifies the successful completion of the transaction T_i, specified by a previous matching begin_trans(T_i), and indicates a willingness to commit the work of T_i. The programmer's definition of end_trans gets precedence over the default definition for the primitive.

If the programmer has not redefined the end_trans primitive, the default execution takes place. In this case the CM for T_i notes the completion of T_i; it asks the relevant TMA-TP module to force a prepare log record and sends vote messages relevant to the default commit protocol to the coordinator for the H-transaction. The control flow does not proceed beyond the end_trans call, until the transaction manager for T_i receives either a commit or an abort decision. If the primitive is invoked without any parameter, then it commits the transaction within which it has been invoked.

As mentioned earlier, it is possible to overload this primitive to have a more flexible programmer-defined commit protocol. From transaction T_i's point of view the execution of the programmer-defined end_trans is the same as the default execution. That is the completion of the transaction T_i is recorded by a prepare log record and control is passed to the thread of the same name being executed at the coordinator. If the thread for end_trans does not contain an explicit invocation of the commit or abort primitives, the control proceeds beyond the transaction T_i after the thread completes execution and returns. However, the transaction T_i remains unterminated until an explicit invocation of commit or abort is eventually performed by the coordinator module for T_i.

Note that this primitive has two parts: end_trans(T_i) and *support_code*. The second part is an optional piece of program code which can be included by the programmer. This program code is not executed when the end_trans primitive is invoked. Rather, the coordinator module can direct the TMA-TP module for T_i to execute this program code by invoking the call_support primitive (explained next).

call_support($T_j, \ldots, T_m$) This primitive can be invoked only as part of the programmer defined coordinator. With this primitive the coordinator module can direct the transaction managers of the transactions $T_j, \ldots, T_m$ to execute the support_codes specified as part of the corresponding end_trans primitives in these transactions. The program fragment for support_code of each T_k runs within the scope of T_k. If a support_code is invoked while the corresponding transaction is running, then the execution of the support_code is deferred until the transaction completes. The call_support returns to the invoking thread, when all the executing support_codes finish. This primitive along with the programmer specified support_code are useful in cases where the coordinator module wants to perform some task beyond merely committing/aborting after the transaction has executed an end_trans primitive.

abort_trans(T_i) The abort_trans is a system defined primitive which can also be redefined by the programmer as a coordinator module thread. In both cases, it signifies the unsuccessful completion of the transaction T_i, specified by a previous begin_trans(T_i), and indicates a decision to abort the work. However, the abort_trans does not actually abort the transaction. Rather the actual abort is performed by the abort primitive which is invoked by default and will always abort T_i. In case the abort_trans primitive is redefined by the programmer, the new definition gets preference over the default definition. If the thread for abort_trans does not contain an explicit invocation of the abort primitive, status of the transaction T_i remains unterminated until an explicit invocation of abort is eventually performed by the coordinator for T_i. The termination of the abort_trans primitive itself is similar to the end_trans primitive as explained above.

Table 4.1 Partial Syntax for the Coordinate Block

Keyword		*Syntax*
coordinate-block	::=	**coordinate** transaction **using** protocol
transaction	::=	**begin_trans**(trans-id) trans-command **end_trans**(trans-id) | **initiate**(trans-id) | **cobegin** transaction **coend** | transaction ; transaction
trans-command	::=	**abort_trans**(trans-id) | host-language-command
protocol	::=	**default** | protocolcode
protocolcode	::=	protocol-command | protocolcode ; protocol-command
protocol-command	::=	**thread begin_trans** | **thread end_trans**(parameter) thread-command | **thread abort_trans**(parameter) thread-command | **thread** identifier(formal-par-sequence) thread-command
thread-command	::=	**commit**(trans-ids) | **abort**(trans-ids) | **restart**(trans-ids) | **call_support**(trans-ids)| **exit** | thread-command;thread-command | host-language-command

coordinate < *transaction* > **using** < *protocol* > This primitive defines the *coordinate block* whose partial syntax is described in table 4.1. (Note that only the enhancements required in a standard programming language to support such a syntax is shown in the table. Anything not been defined is assumed to follow the syntax of the host language.) The coordinate block consists of two components: the *transaction* component and the *protocol* component. The protocol component defines the coordinator module for the set of transactions specified in the transaction component. The protocol component can be the keyword default, in which case one of the traditional commit protocols like two-phase commit or early prepare is used. Or it can be a programmer specified dependency among the transactions in the high level declarative language or it can be a programmer defined code in which case it contains the code for each of the primitives that the programmer wants to define or redefine, including begin_trans(t), end_trans(t) and abort_trans(t). The scope of the redefined primitives is limited to the corresponding transaction component.

Within the coordinate block the programmer can define persistent variables which can live across the boundaries of transactions involved in the coordinate block. These persistent variables may be useful for flow control.

The control flow does not proceed beyond the coordinate block until either the transaction component completes or the coordinator module is terminated in a manner explained below.

thread *identifier* For efficiency and ease of implementation as daemons, the protocol component is programmed as a set of concurrently executing threads. The thread primitive allows the programmer to define a coordinator module thread which is activated by a transaction event. The identifier specifies the event which activates the thread. When a coordinate block is encountered, the coordinator module is created. It waits for any of the events named in its threads. When such an event occurs the corresponding thread is activated. If the thread encounters an **exit** command, the coordinator module terminates thereby causing the entire coordinate block to end. This is true even if there are transactions in the transaction component which are either yet to be executed or are currently executing concurrently. These transactions have to be taken care of by a subsequent coordinate block otherwise may lead to the problem of *orphan transactions.*[3]

On the other hand if an exit command is not encountered, the thread does not cause the coordinator module to terminate. Instead when the thread completes, it returns control to the transaction component. If an exit command is never encountered, the coordinator module terminates when all transactions have completed their execution and the coordinate block has terminated.

exit Invocation of this command causes the termination of the coordinator module.

Table 4.2 summarizes the transaction primitives in our model. The first column of table 4.2 lists the different primitives. The primitives have been grouped into three categories - (i) primitives that allow the structuring of the H-transaction (ii) primitives that can be invoked from within the transaction component of a coordinate block and (iii) primitives that can be invoked from only within the protocol component of a coordinate block. The second column specifies where the programmer can use each primitive from viz., inside or outside a protocol component. The third column gives the system component that provides the interface to a particular primitive. When a primitive is invoked, this system component executes the primitive first. It in turn may invoke other system components in order to carry on the execution of the primitive.

Table 4.2 Summary of Transaction Primitives

Primitive Name	*Invocation Relative to Protocol Component*	*Interface Exported By*	*Nested Definition*	*Redef-inition*
coordinate ... using	outside	language support	no	no
cobegin ... coend	outside	language support	yes	no
initiate	outside/inside	TMA-TP	-	no
sid = savework	outside	TMA-TP	-	no
rollback	outside	TMA-TP	-	no
begin_trans	outside	CM or TMA-TP	yes	yes
end_trans	outside	CM or TMA-TP	yes	yes
abort_trans	outside	CM or TMA-TP	-	yes
thread	inside	CM	-	no
commit	inside	TMA-TP	-	no
abort	inside	TMA-TP	-	no
restart	inside	TMA-TP	-	no
call_support	inside	TMA-TP	-	no
exit	inside	TMA-TP	-	no

The fourth column specifies which primitives provide a bracketing construct to specify nesting from the syntactic point of view. Finally the fifth column indicates whether a primitive can be redefined by the programmer in the coordinate block. Note that we allow only begin_trans, end_trans and abort_trans to be redefined in the current model of H-transactions.

4.4.3 Discussion

In the course of a transaction execution, a sid = savework() primitive may be executed more than once. In such cases it is preferable to assign each time a new handle which is generated by the system since otherwise the transaction loses the ability to refer to the exact savepoint among those that were established previously.[4] As an example, suppose the programmer wants to undo the effects of a loop based on certain conditions established during the execution of the loop. If a savepoint is established within the loop and the same savepoint identifier is employed, then the programmer can undo only the latest iteration of the loop. This is because the programmer loses reference to the other savepoint handles. (Note, however, that if a programmer establishes a savepoint just before a loop, then the effects of all iterations of the loop can be fully undone. This is possible because the scoping rules of the savepoint identifier is the block in which the sid = savework() primitive is executed.)

Note that although the restart(T_i) command may seem semantically equivalent to a rollback to the beginning of the transaction, there is one important difference between the two. The restart primitive can be executed only by the coordinator; a transaction cannot restart itself. The rollback primitive, on the other hand, is invoked by the transaction itself. The coordinator does not have any idea about savepoints established by a transaction and hence is not allowed to execute a rollback primitive.

Finally, note that a coordinator for a transaction T_i can multiply invoke commit or abort primitives for T_i. Usually this occurs if T_i is to be conditionally aborted or committed. In such cases, the first execution at runtime of either primitive takes effect while the others, if executed subsequently, performs only null operations and generates warning messages. Further, the initiate command can be invoked from both outside or inside a coordinate block. If it is invoked from oustide a coordinate block then the scope of the identifier T_i specified in the invocation of initiate is the entire program code for the H-transaction; else the scope is limited only to the particular coordinate block from which initiate is invoked. In the former case we can have a number of coordinate blocks for a single transaction T_i defined within the scope of the identifier T_i; however, at most two coordinators can actually be involved for terminating T_i. The scoping rules ensure that every time an end_trans or an abort_trans is invoked, it gets bound to only one thread, viz. to the thread which is defined at point closest to the invocation. Hence, the closest coordinator will execute end_trans (or abort_trans) without committing or aborting T_i and a second will perform the actual commit or abort operation.

4.5 REALIZING VARIOUS TRANSACTION DEPENDENCIES

We now show how different transaction dependencies that are present in various extended transaction models can be specified using our primitives. We would like to emphasize here that we are interested in only the termination dependencies among transactions. We do not attempt to capture dynamic dependencies that are not known a priori. Such dynamic dependencies arise mostly due to data sharing among the transactions.

4.5.1 ACTA Framework

The ACTA framework defines two major types of termination dependencies among pairs of transactions. (i.e. dependencies that arise between transactions due to commit or abort of one of them and not due to data sharing among them). These dependencies are the commit dependency and the abort dependency defined as follows:

Commit Dependency If transaction T_i develops a commit dependency on transaction T_j then T_i cannot commit until T_j either commits or aborts. Note that this does not imply that if T_j aborts, then T_i should abort as well.

Abort Dependency If transaction T_i develops an abort dependency on transaction T_j then if T_j aborts T_i should also abort. Note that this does not imply that T_i should commit if T_j commits, nor that T_j should abort if T_i aborts.

Note that an abort dependency implies a commit dependency. If T_1 develops an abort dependency on T_2, then T_1 must wait for the commit decision of T_2; hence T_1 cannot commit before T_2, i.e. there is a commit dependency between T_1 and T_2. In figures 4.4 and 4.5 we show how our primitives can be used to express the commit and abort dependencies of ACTA.

In figure 4.4, suppose that T_1 wants to commit. It executes an end_trans primitive which causes the end_trans thread at the coordinator to be executed. Since T_2 has not yet executed the end_trans (or abort_trans) primitive, the variable doneT2 is false. Consequently the end_trans thread sets the variable completedT1 to true and then returns. As no commit or abort decision has been taken for T_2 by the coordinator module, T_1 cannot terminate at this time by committing. On the other hand if T_1 had decided to abort, it would have executed the abort_trans primitive, which in turn, would have caused the abort_trans thread to be executed at the coordinator. This would abort T_1 irrespective of whether T_2 commits or aborts.

When T_2 decides to commit or abort, the variable doneT2 will be set to true by one of the threads end_trans or abort_trans. If T_2 executes an abort_trans primitive, the corresponding thread aborts T_2. The abort_trans thread then finds that the variable completedT1 is set to true (which indicates that T_1 is waiting to commit) and hence commits T_1. If, on the other hand, T_2 executes the end_trans primitive (indicating that it wants to commit), the end_trans thread commits T_2 first and then, noticing that completedT1 is set to true, commits T_1. At this point the program terminates.

From the above discussion it is clear that the program in figure 4.4 implements the ACTA commit dependency between T_1 and T_2. Figure 4.5 implements an abort dependency between T_1 and T_2. The reason is similar to the one above with the only difference being that if T_2 executes an abort_trans primitive, the corresponding abort_trans thread in the coordinator aborts both T_2 and

```
void commit_dependency ()
{       coordinate ;
        initiate (T1,T2) ;
        cobegin
           begin_trans (T1)
           ...
           end_trans (T1) ;
           begin_trans (T2)
           ...
           end_trans (T2) ;
        coend ;
        using {
                    completedT1 := false ; doneT2 := false ;
                        thread end_trans (M) {
                            if doneT2 then {commit(T1); exit ;}
                            else if M == T2 then {
                                    commit(T2); doneT2 = true ;
                                    if completedT1 then { commit(T1); exit ;}}
                                else completedT1 = true ; }
                        thread abort_trans (M) {
                            if M == T2 then {abort(T2) ; doneT2=true;
                                if completedT1 then { commit(T1); exit ;}}
                            else abort(T1) ;
                        }
           }
        end
}
```

Figure 4.4 ACTA commit dependency

T_1, even if T_1 has previously decided to commit. Moreover, if T_1 is yet to reach a decision when T_2 has decided to abort, T_1 gets aborted.

4.5.2 Sagas

Saga [Garcia-Molina and Salem, 1987] is a transaction model that provides system support for the execution of a long-lived transactions. In sagas, a long-lived transaction is executed as a number of shorter subtransactions without sacrificing the atomicity of the larger transaction, although other transactions may see the effects of a partial saga execution.

A saga consists of a set of flat transactions T_1, T_2, ..., T_n that execute sequentially within the context of the saga, but can interleave arbitrarily with component transactions of other sagas. For each T_i ($1 \leq i < n$) there is a compensating transaction CT_i which, if executed, semantically undoes the effects of T_i. A compensating transaction CT_i is executed iff the transaction T_i has committed and the saga of which T_i is a part, has aborted. A saga commits if all T_i's successfully commit and aborts if any T_i aborts. If a saga aborts, it compensates for the effects of all committed components T_j's by executing their corresponding compensating CT_j's. The compensating transactions are executed in the reverse order of the commits of the corresponding T_i's. Note that there is no compensating transaction for the last component transaction T_n.

```
void abort_dependency ()
{       coordinate ;
        initiate (T1,T2) ;
        cobegin
           begin_trans (T1)
           ...
           end_trans (T1) ;
           begin_trans (T2)
           ...
           end_trans (T2) ;
        coend ;
        using {
                    completedT1 := false ; doneT2 := false ;
                       thread end_trans (M) {
                          if doneT2 then {commit(T1); exit ;}
                          else if M == T2 then {commit(T2); doneT2 = true ;
                                   if completedT1 then { commit(T1); exit ;}}
                              else completedT1 = true ;
                       }
                       thread abort_trans (M) {
                          if M == T2 then { abort(T2,T1); exit; }
                          else abort(T1) ;
                       }
           }
        end
}
```

Figure 4.5 ACTA abort dependency

This is because if T_n commits then the entire saga commits. The final outcome of a saga is either the sequence:

1. $T_1, T_2, \ldots, T_{n-1}, T_n$ if all T_i's commit, or
2. $T_1, T_2, \ldots, \underbrace{T_i}_{abort}, CT_{i-1}, \ldots, CT_2, CT_1$ if any T_i aborts.

In figure 4.6 we show how the semantics of a saga can be achieved with our primitives. The saga program consists of one coordinate block which controls the execution flow of the transactions $T_1, \ldots, T_n$ and the corresponding compensating transactions $CT_{n-1}, \ldots, CT_1$. In the transaction component of the coordinate block the T_i's and the CT_j's (if so required) are executed sequentially. If T_n successfully completes, then the coordinator aborts $CT_{n-1}, \ldots, CT_1$ (as no compensation is required) and the saga terminates successfully. On the other hand, if any T_i aborts, the thread abort_trans(T_i) in the coordinator is executed, which aborts the transaction $T_i, \ldots, T_n$ as well as the compensating transactions $CT_{n-1}, \ldots, CT_i$. In this way the transactions remaining to be executed, viz., $CT_{i-1}, \ldots, CT_1$ become exactly those required to compensate the effects of the already committed transactions $T_1, \ldots, T_{i-1}$. If a CT_k aborts, the thread abort_trans(CT_k) in the coordinator gets executed, which in turn restarts the compensating transaction CT_k. In this way the effects of all the committed transactions are compensated for and the saga aborts.

```
void saga ()
{ initiate (T1,T2,...,Tn,CT1,CT2,...,CTn-1);
    coordinate
        begin_trans (T1)
            ...
        end_trans;
        begin_trans (T2)
            ...
        end_trans;
        :
        begin_trans (Tn)
            ...
        end_trans;
        begin_trans (CTn-1)
            ...
        end_trans;
        :
        begin_trans (CT2)
            ...
        end_trans;
        begin_trans (CT1)
            ...
        end_trans ;
    using
                    thread end_trans (M) {
                        commit (M) ;
                        if M == Tn then {abort(CT1,...,CTn-1); exit };
                    }
                    thread abort_trans (M) {
                        case (M) do
                        T1: { abort(T1,...,Tn, CT1,CT2,...,CTn-1 ); exit} ;
                        T2: abort(T2,...,Tn,CT2,...,CTn-1) ;
                        T3: abort (T3,...,Tn,CT3,...,CTn-1) ;
                        :
                        Tn: abort(Tn) ;
                        CT1: restart(CT1) ;
                        CT2: restart(CT2) ;
                        :
                        CTn-1: restart(CTn-1) ;
                    }
    end ;
}
```

Figure 4.6 Implementation of a saga

4.5.3 Workflows and Long Lived Activities

ACID properties of transactions have the limitation that hide any internal structure to be perceived and referred to from outside of the transaction. Consequently if there is an activity that consists of multiple steps of processing with an explicit flow of control among these steps, it is difficult to model it as a transaction.

Workflows have been suggested as a way of implementing long-lived activities which have some kind of an internal structure, in terms of shorter transaction like components [Dayal et al., 1990]. Workflows allow dependencies among transactions to be expressed and also allows correctness requirements among the component transactions that are less stringent than serializability and isolation.

We show by an example how a workflow can be expressed by our primitives. The example workflow involves planning for a trip by John Doe. He plans to leave on the 3rd of June by either Delta, or United, or American in that order, stay at the hotel Ambassador from the 3rd until the 6th of June. and rent a car from either National or Avis with no preference. If any of the reservations (i.e., flight, hotel or car) cannot be made, John Doe would like to cancel his trip.

In the example in figure 4.7 the different components flightReservation, hotelReservation, carReservation, cancelFlightReservation and cancelHotelReservation perform the actual reservation or cancellation operations. The single coordinate block for the workflow contains the transactions T_1, ..., T_6 and the compensating transactions CT_1,CT_2. CT_1 compensates for any committed flight reservation made by T_1, T_2, or T_3 in case either the hotel reservation or the car reservation cannot be made. CT_2 compensates for a committed hotel reservation if the car reservation is unsuccessful.

Every time a transaction completes, it invokes the end_trans thread at the coordinator which then enforces the control flow of the activity. The successful completion of the workflow is indicated by the commit of either T_5 or T_6. In this case the coordinator ensures that CT_1 or CT_2 are aborted.

If any transaction decides to abort, it invokes the abort_trans thread at the coordinator. The execution of the abort_trans thread for a transaction T_i aborts all transactions T_j that follow T_i in the workflow and compensates for the committed T_k's preceding T_i in the workflow. In case any compensating transaction CT_i gets aborted, it has to be reexecuted until it successfully completes.

4.5.3.1 Semiatomicity. A formalization of the workflow model is provided in [Zhang et al., 1994b]. In this paper a workflow is synonymous to a flexible transaction. The structure of a flexible transaction T is viewed as a set of the so called *representative partial order* of subtransactions. The subtransactions within a representative partial order are related by the *precedence* relation. Each representative partial order gives an alternative for the execution of the flexible transaction. There is also a *preference* relation which defines the preferred order of the alternatives. Each subtransaction is categorized as either *retriable*, *compensatable*, or *pivot.*

The execution of a flexible transaction T preserves the property of *semiatomicity* if one of the following conditions is satisfied:

```
void workflow ()
{
        initiate(T1,T2,T3,T4,T5,T6,CT1,CT2)
        coordinate
        airline* air;          % persistent variable
            begin_trans(T1)
                flightReservation(Delta, 6/3/96) ;
                air = Delta ;
            end_trans(T1) ;
            begin_trans(T2)
                flightReservation(United, 6/3/96) ;
                air = United ;
            end_trans(T2) ;
            begin_trans(T3)
                flightReservation(American, 6/3/96) ;
                air = American ;
            end_trans(T3) ;
            begin_trans(T4)
                hotelReservation(Ambassador, 6/3/96, 6/6/96) ;
            end_trans(T4) ;
            cobegin
                begin_trans(T5)
                    carReservation(National, 6/3/96, 6/6/96) ;
                end_trans(T5)
                begin_trans(T6)
                    carReservation(Avis, 6/3/96, 6/6/96) ;
                end_trans(T6)
            coend;
            begin_trans(CT1)
                cancelFlightReservation(air, 6/3/96) ;
            end_trans(CT1);
            begin_trans(CT2)
                cancelHotelReservation(Ambassador, 6/3/96, 6/6/96) ;
            end_trans(CT2);
        using
                    thread end_trans (M) do {
                        case M of {
                            T1 : { commit (T1); abort (T2,T3);} ;
                            T2 : { commit (T2); abort (T1,T3);} ;
                            T3 : { commit (T3); abort (T1,T2);} ;
                            T5 : { commit (T5); abort (T6,CT1,CT2); exit;} ;
                            T6 : { commit (T6); abort (T5,CT1,CT2); exit;} ;
                            default: commit (M);    %  commit T4 or CT1 or CT2
                        };
                    };
                    thread abort_trans (M) do {
                        noSet = union (noSet,M) ;
                        if subseteq({T1,T2,T3},noSet) then
                            { abort (T1,T2,T3,T4,T5,T6,CT1,CT2); exit }
                        if subseteq(T4,noSet) then
                            abort (T4,T5,T6,CT2);
                        if subseteq({T5,T6},noSet) then
                            abort (T5,T6);
                        if M==CT1 or M==CT2 then
                            restart(M);
                    };
        end
}
```

Figure 4.7 Workflows: reservations

1. All its subtransactions in one representative partial order commit and all attempted subtransactions not in the committed representative partial order are either aborted or have their effects undone.

2. No partial effects of its subtransactions remain permanent in local database.

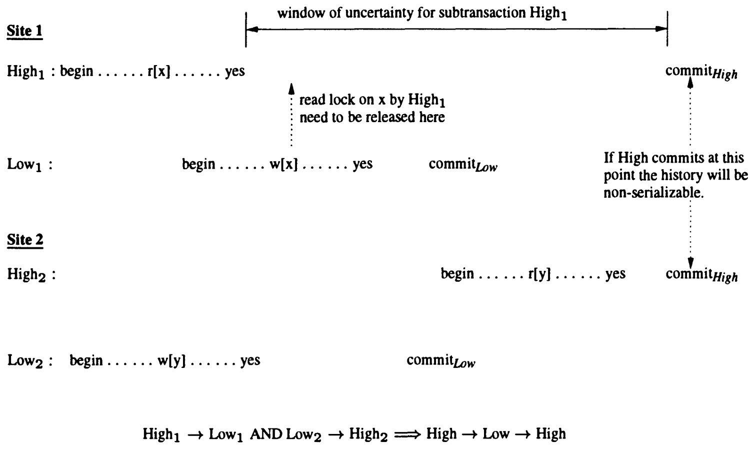
In [Zhang et al., 1994b] the authors provide a commit protocol which dynamically commits subtransactions as soon as possible. An alternative representative partial order is executed if an attempted subtransaction aborts. In this case subtransactions which have already been committed in the failed representative partial order are compensated for.

Given an instance of a flexible transaction we can implement a coordinator for this flexible transaction in a manner similar to implementing a workflow, shown previously. In fact a compiler can be made to generate the codes for such a coordinator, given an appropriate description of the flexible transaction with the different precedence and preference relations.

4.5.4 Secure Distributed Transactions

A major problem of all lock-based concurrency protocols in multilevel secure (MLS) database systems is that in order to avoid a covert channel any read lock acquired by a higher security level transaction on a lower security level data object must be released whenever a lower level transaction attempts to acquire a write lock on the same data object. Unfortunately, this requirement has grave implications for the corresponding commit protocol, specially the early prepare commit protocol (EP) [Mohan et al., 1986, Stamos and Cristian, 1993]. What it implies is that read locks may get released within a subtransaction's *window of uncertainty* (period after a participant has voted yes to commit a subtransaction, but before it receives the commit or abort decision from the coordinator), possibly resulting in nonserializable executions [Jajodia and McCollum, 1993, Jajodia et al., 1994].

Consider the history in figure 4.8 showing two distributed transactions *Low* and *High* such that transaction *Low* is at a lower security level than transaction *High*. Each distributed transaction consists of two subtransactions Low_1, Low_2 and $High_1$, $High_2$ with Low_1 and $High_1$ executing at Site 1 and Low_2 and $High_2$ executing at Site 2 respectively. Among the data objects accessed by *Low* and *High* are x and y with the security level of x being the same as that of y and equal to the security level of transaction *Low*. Data object x is at Site 1 while y is at Site 2. The order of execution of each subtransaction is shown in figure 4.8. The event yes in the figure signifies that the subtransaction has completed execution and has sent an yes vote to the coordinator. Note that when $w[x]$ is invoked by Low_1 the operation cannot be delayed waiting for $High_1$ to release the read lock on x. This is in order to avoid a covert channel between the the security levels of transactions *High* and *Low*. Consequently, although $High_1$ is in its window of uncertainty when Low_1 requests write lock on x, the read lock on x by $High_1$ has to be released. Basic EP protocol does not take into account that read locks may be released during a subtransaction's window of uncertainty. In this case EP will commit both distributed transactions *High* and *Low* thereby leading to the non-serializable history shown in figure 4.8.



window of uncertainty for subtransaction High1
Site 1
High1 : begin ...... r[x] ...... yes
commitHigh
read lock on x by High1 need to be released here
Low1 : begin ...... w[x] ...... yes
commitLow
If High commits at this point the history will be non-serializable.
Site 2
High2 : begin ...... r[y] ...... yes
commitHigh
Low2 : begin ...... w[y] ...... yes
commitLow
High1 → Low1 AND Low2 → High2 ⟹ High → Low → High

Figure 4.8 Example history illustrating problem with EP in MLS systems

To overcome this problem, a secure EP commit protocol (SEP) has been proposed in [Atluri et al., 1994]. It implements the following *secure commit* dependency in addition to the conventional commit/abort dependencies for distributed systems:

> Given any two participants T_i and T_j of a multilevel secure distributed transaction T, there is a secure commit dependency between T_i and T_j, denoted by $T_i \xrightarrow{s} T_j$, defined as follows:
>
> If either T_i or T_j releases any of its low read locks within its window of uncertainty, before all participants complete, then both T_i and T_j are aborted.

In other words, to prevent nonserializable executions, SEP aborts a distributed transaction if any of its lower reading subtransactions is compelled to release a lower level read lock within the subtransaction's window of uncertainty, before the other subtransactions complete. As usual, SEP guarantees that either all participants abort or all of them commit.

SEP for MLS systems can be implemented within our framework by incorporating the *GetSignal* primitive introduced in [Bertino et al., 1997]. The basic idea is that the lock manager at a site notifies the transaction manager by sending the latter a *signal* (similar to raising an exception), that a higher level subtransaction at that site has released one of its lower level write locks because a lower level subtransaction at the same site has requested a write lock on the same data item [5]. Each signal received by the transaction manager from the lock manager identifies a lower level data object x that has been read by the higher level subtransaction T_i and indicates a new value for x. Before the higher level subtransaction can commit, the subtransaction has to handle these exceptions generated by the lock manager and the GetSignal primitive is used by the programmer to specify how signals from lower level subtransactions are to be handled by the higher level subtransaction.

The GetSignal primitive has the syntax: GetSignal[[sl$_1$ $\rightarrow$ *handler*$_1$], ..., [sl$_n$ $\rightarrow$ *handler*$_n$]]. It has two exit points: A standard one which is the next instruction after the GetSignal and an exceptional continuation which is represented by the expression

$$[sl_1 \rightarrow handler_1], \ldots, [sl_n \rightarrow handler_n]$$

On receiving a signal from the lock manager, the transaction manager locates the savepoint sl_i that immediately precedes the read of the data object identified by the signal and associates the savepoint identifier sl_i with this signal. For example, if the signal indicates a new value for the data object x, then the signal label sl_i established by the first step sl_i = SaveWork() preceeding the operation $r_i[x]$ in the subtransaction body, is chosen. Each of the sl_i's in the expression for the GetSignal primitive, represents one such savepoint identifier that has been associated to a signal; *handler*$_i$ represents a programmer specified piece of code to be executed in order to handle the associated signal. We say the savepoint sl_i *covers* the data object in question.

If multiple low read locks of T_i had to be released, the transaction manager receives multiple signals, one for each broken lock. It buffers all such signals. Later on when T_i invokes a GetSignal call, the transaction manager considers all the signals it has buffered for T_i, and selects one signal to be serviced as follows: It selects that signal whose associated savepoint identifier covers all the low reads with released read locks.

The default invocation for the getSignal primitive is: GetSignal[$\rightarrow$ *handler*]. In this case, for any signal that needs to be serviced, the same code of *handler* is executed.

We now show how it is possible to implement a commit protocol that enforces the secure dependency among subtransactions of a multilevel secure distributed transaction T. The SEP protocol is achieved by adding a suitable GetSignal call as the support_code (refer to the discusion on the semantics of the primitive end_trans(T_i)< *support_code* > in section 4.4.2) for each subtransaction T_i and making the coordinator module invoke a call_support for each of the lower reading subtransactions to invoke in its turn, the GetSignal calls.

Figure 4.9 shows an H-transaction, T, that implements the SEP protocol for committing three concurrent subtransactions T_1, T_2 and T_3. These three subtransactions are related to each other by the secure dependency $T_i \xrightarrow{s} T_j$ ($1 \leq i, j \leq 3, i \neq j$). In this example transactions T_1 and T_3 read data at lower security levels, but not T_2. We assume that a secure two-phase locking protocol is used to provide local concurrency control at each site.

As each subtransaction T_1, T_2, and T_3 completes, it invokes the end_trans thread in the coordinator module and enters its corresponding window of uncertainty. When the last of the subtransactions has invoked the end_trans thread, the coordinator module executes the call_support primitive for transactions T_1 and T_3. Note that the call_support is not invoked for T_2 as this subtransaction does not read down. The call_support primitive in turn causes the support_codes defined in T_1 and T_3 to be executed.

Each support_code is of the form < GetSignal[$\rightarrow$ abort_trans(T_i)]; noSignalServiced >. The GetSignal call has the format of the default invocation. Thus if there is any signal to be serviced the exceptional continuation of GetSignal denoted by abort_trans(T_i) gets executed. On the otherhand if there is no signal the statement following the GetSignal is executed - in this case the thread noSignalServiced defined in the coordinator module.

If any of T_1 or T_3 invokes abort_trans (indicating it had to release a lower level read lock within its window of uncertainty), the coordinator module thread abort_trans aborts all the three subtransactions T_1, T_2 and T_3 and then exits. On the other hand if both T_1 and T_3 invokes noSignalServiced it implies that none of them had a signal to service, i.e. none of the subtransactions had to release a lower level read lock within its window of uncertainty. At this point it is assured that the H-transaction T comprising of the three subtransactions is two-phased and hence the noSignalServiced thread commits the three subtransactions and exits.

Note that the GetSignal and call_support primitives can be used in tandem by the programmer to implement more complex secure commit protocols like the ones shown in [Ray et al., 1996]. All that the programmer has to do is write a suitable support_code and coordinator module thread corresponding to the desired behavior of each subtransaction. The support code should define how signals are to be serviced and what needs to be done in the absence of any signal and may invoke programmer defined coordinator module threads.

4.5.5 Contingent Transactions

A contingent transaction [Elmagarmid, 1992] is a set of two or more component transactions $T_1, T_2, \ldots, T_n$ with the property that at most one of the transactions, say T_i, commits. A contingent transaction $T = \{T_1, T_2, \ldots, T_n\}$ is executed as follows: T_1 gets executed first. If it commits then the transaction T

```
void secure_distributed_commit ()
{
        initiate (T1,T2,T3) ;
        coordinate
        cobegin
           begin_trans (T1)
                r[x] ;
                sl2 = SaveWork() ;
                w[z] ;
                r[y] ;                    /* this is a low read */
                sl3 = SaveWork() ;
                r[q] ;                    /* another low read */
           end_trans (T1) {GetSignal[→ abort_trans(T1) ];
                               noSignalServiced ; } ;
           begin_trans (T2)
                r[o] ;
                w[p] ;
           end_trans (T2) ;
           begin_trans (T3)
                r[s] ;                    /* this is a low read */
                sl2 = SaveWork() ;
                r[y] ;                    /* another low read */
                w[q] ;
           end_trans (T3) {GetSignal[→ abort_trans(T3) ];
                               noSignalServiced ;} ;
        coend ;
        using
                       thread end_trans (M) {
                           completedSet := union(completedSet,M);
                           if completedSet = {T1,T2,T3}
                               then call_support(T1,T3);
                       }
                       thread noSignalServiced (M) {
                           commitSet := union(commitSet,M) ;
                           if commitSet = {T1,T3}
                               then { commit (T1,T2,T3); exit ;}
                       }
                       thread abort_trans (M) {
                           abort (T1,T2,T3); exit;
                       }
        end
}
```

Figure 4.9 A secure distributed commit protocol

commits and ends. If T_1 aborts, T_2 gets executed and if it commits, T commits and ends, and so on.

The program fragment in figure 4.10 shows how a contingent transaction can be implemented within our framework. In the example the contingent transaction consists of three component transactions T_1, T_2 and T_3. Note the sequential definition of the three transactions in the body of the H-transaction (they are not within any cobegin ... coend block) ensures that first T_1 gets executed and invokes end_trans() or abort_trans(). Then depending on whether the H-transaction terminates or not T_2 and/or T_3 gets executed.

```
void contingent()
{
        initiate(T1,T2,T3) ;
        coordinate
           begin_trans (T1)
                 .
                 .
                 .
           end_trans (T1);
           begin_trans (T2)
                 .
                 .
                 .
           end_trans (T2);
           begin_trans (T3)
                 .
                 .
                 .
           end_trans (T3);
        using
                              thread end_trans (M) {
                                   if M == T1 then
                                        { commit (T1); abort (T2,T3); exit; }
                                   if M == T2 then
                                        { commit (T2); abort (T1,T3); exit; }
                                   if M == T3 then
                                        { commit (T3); abort (T1,T2); exit; }
                              }
                              thread abort_trans (M) {
                                   abortSet = union(M,abortSet) ;
                                   if subseteq({T1,T2,T3},abortSet) then
                                   { abort (T1,T2,T3) exit; }
                              }
        end
}
```

Figure 4.10 Example of a contingent transaction

4.5.6 Nested Transactions

A nested transaction is a transaction that is executed from inside the dynamic scope of another transaction. Nested transactions can further create nested transactions and the nesting can proceed to arbitrary depths. The transaction at the root of this tree of transactions is called the root transaction and the transactions at the interior nodes (called parents) or leaves of this tree are jointly called subtransactions. Subtransactions execute atomically with respect to their siblings.

Each of the parent transactions is suspended until all its nested transactions terminates (i.e., commits or aborts). However, the semantics of commit for the nested transactions are different from that for the root transaction. When a nested transaction (parent or leaf) commits, the changes that it made to the database are made accessible to its parent, but are not made permanent. The changes are made permanent only when the root transaction commits. Abort semantics for both root and subtransactions are similar to the abort semantics for the classical transaction. Furthermore, a subtransaction can access any data item that is currently accessed by one of its ancestors without forming a conflict.

We illustrate the implementation of the termination dependency of a nested transaction in our model by a simple example. The example involves a transaction nested to two levels, which makes travel arrangements for John Doe. If at any stage a reservation cannot be made, the trip is cancelled. At any stage thus, if the trip is to be cancelled, any previous reservation has to be cancelled. Note that unlike in the workflow model where previous reservations are cancelled by explicitly executing compensating transactions, in the nested transaction we do not require any compensating transaction. This is because of the fact that the effects of subtransactions are made permanent only at the commit of the root transaction. We assume that the code for the subtransactions are already there for the example in figure 4.11. Also note that the actual implementation of nested transactions requires proper implementation of the data-sharing dependencies among the subtransaction. We assume that such mechanism are already in place.

```
void nested-transactions ()
{
      initiate(T1,T2,T3,T4);
      coordinate
      begin_trans(T1)
         :
         begin_trans(T2)
             flightReservation(United, 6/3/96)
         end_trans(T2) ;
         :
         begin_trans(T3)
             hotelReservation(Ambassador, 6/3/96, 6/6/96)
         end_trans(T3) ;
         :
         begin_trans(T4)
             carReservation(Avis, 6/3/96, 6/6/96)
         end_trans(T4) ;
         :
      end_trans(T1) ;
      using
                    thread end_trans (M) do {
                        if M == T1 then commit (T1,T2,T3,T4); exit ;
                    };
                    thread abort_trans (M) do {
                            abort (T1,T2,T3,T4); exit ;
                    };
      end
}
```

Figure 4.11 Nested transactions

4.6 CONCLUSIONS AND FUTURE WORK

This paper presents a flexible commit facility that allows the programmer to achieve various transaction dependencies of different extended transaction models. The transaction dependencies are implemented by a set of coordinator modules that interact with the system's default commit/abort mechanism. The programmer is provided with a small set of transaction primitives by which he can develop application specific coordinator modules. Moreover, the programmer can redefine some of these primitives for additional flexibility by providing the code for the implementation of the new definitions. The compiler of a database programming language can also use these primitives to support higher level constructs for transactions. In this case, the compiler can automatically generate the appropriate codes needed for coordination of a set of transactions from a high level description of their dependencies.

Not only can the programmer re-define some of the existing primitives, he can also define newer primitives with well-defined interfaces to satisfy his particular requirements. In this case these new primitives are defined as new threads of a coordinator and are invoked from a transaction. An example of such a new application specific primitive has been the noSignalServiced primitive shown in section 4.5.4, where it was used to support the secure dependencies among transactions. Allowing custom primitives with well-defined interfaces seems useful for supporting some other extended transaction models not discussed in this work, like the split-join transaction model. For example in the case of split-join transactions, the programmer can define two new threads *split and join* in the protocol component. The split thread starts a new transaction and delegates a set of data to the new transaction. The join thread is the complement of the split thread; it joins to transactions.

Our commit facility seems to be a practical way to implement extended transaction models on top of existing TP systems following the same approach as that of [Barga and Pu, 1995]. In this work, the authors extend Transarc's Encina TP system [Encina, 1993, Gray and Reuter, 1993] by developing transaction management adapters on top of Encina. We choose to use a similar approach. Our transaction management adapters offer the same functionality as the transaction management adapter of [Barga and Pu, 1995] while our coordinator module can viewed as an extended version of the notion of metatransactions of [Barga and Pu, 1995] built on top of transaction management adapters. A coordinator module in an H-transaction lists the set of primitives that are invoked by a component transaction alongwith an indication as to what type of primitive each is (for example if it is a system primitive or it is one of the new primitives that we have defined). It also contains the codes for these primitives. In this manner we can support extended transaction models on conventional TP system once the transaction adapter layer has been implemented.

One advantage of our scheme over [Barga and Pu, 1995] is our ability to support application specific dependencies that do not fit into any general model. The secure dependency is one such example. We plan to implement the proposed primitives within the framework of an ongoing project on MLS transaction processing system. When these current set of primitives are combined with the flexible secure two-phase locking proposed in [Bertino et al., 1997], we should have a complete flexible MLS transaction processing system that supports both classes of dependencies, transaction as well as data dependencies between MLS transactions.

Notes

1. Note that a process can be made to react to an event in many different ways: The event can generate an interrupt to the process; the event can send a message to a port at which the process listens or the event can invoke a RPC at the process. We choose not to specify the exact mechanism so as to keep the model as much implementation independent as possible.

2. In most commit protocols, if any subtransaction aborts, the coordinator always sends an abort decision to all participants. However, in our protocol the coordinator may not send an abort decision. Instead the coordinator can ask the transaction to restart its execution. This can be useful in many situations. For example suppose the subtransaction was aborted because of a site crash. Then when the site comes up, the subtransaction can be restarted.

3. A transaction T_i is an orphan if it is never explicitly terminated by any coordinator module within the H-transaction. When a transaction T_i is orphan the locks acquired by T_i are not released and the updates made by T_i are not made permanent. This may cause a number of problems like deadlock or unsatisfiable dependencies. A complete discussion is outside the scope of this paper.

4. We assume here that the programmer does not save the contents of an sid before reusing it.

5. Such a facility of the lock manager notifying the transaction manager about early lock release by transactions is available in some secure transaction processing system like Informix Online/Secure [Informix, 1993]

Acknowledgments

This work was partially supported by an ARPA grant, administered by the Office of Naval Research under grant number N0014–92–J–4038, by National Science Foundation under grants IRI–9303416 and INT- 9412507, and by National Security Agency under grant MDA904–94–C–6118. The work of E. Bertino and L. Mancini was carried out while visiting George Mason University during summer 1995.

5 CONTRACTS REVISITED

Andreas Reuter, Kerstin Schneider
and Friedemann Schwenkreis

Abstract: To meet the correctness requirements of mission-critical processes workflow systems have to commit guarantees regarding their behavior in case of failures and concurrency. The ConTract model is a conceptual framework for the reliable execution of long-lived computations in a distributed environment including workflows. This paper focuses on the aspect of maintaining consistency in ConTracts and containing consistency violations. It will give an overview of how consistent execution is formally treated in the ConTract model. We present a correctness criterion, which introduces a formal basis to verify execution histories and to build up correctness ensuring mechanisms. It is a unified criterion for recoverability and permeability, named as invariant-based serializability, which is based on a conflict-relationship between invariants in the ConTract model. A formal definition of compensation is given and extensions of the compensation mechanism are introduced. These extensions are a first step to leverage the concept of compensation such that it can be used as a general-purpose mechanism as in real applications. In particular, the support of semi-transactional steps and the performance can be enhanced and advanced semantics of workflows can be supported.

5.1 INTRODUCTION

The ConTract model is a conceptual framework for the reliable execution of long-lived computations in a distributed environment. Properties like this are particularly important for applications which during the past ten years or so have come to be known as "workflow". This does not say ConTracts embody a workflow system; they do, however, provide a complete run-time system, including an execution model, a failure model, etc. for advanced workflow applications. One might say that ConTracts are to workflow what the Java virtual

S. Jajodia et al. (eds.), *Advanced Transaction Models and Architectures*
© Springer Science+Business Media New York 1997

machine is to Java-based applications. Since it was first presented in 1988, the ConTract model has evolved in multiple ways: First, there has been a sequence of prototype implementations, none of which encompasses the full set of concepts. Second, a number of ideas from the ConTract model have been incorporated into commercial products, either indirectly, via the literature, or via osmosis, by members of the ConTract team joining the respective development teams. And finally, the original research group (at the type level, that is; the people have changed) has continued to work on some of the more fundamental issues of long-running computations, such as formal consistency constraints and their implications on the execution model, on recovery, and so on.

This paper will focus on the aspect mentioned last, i.e. the problem of maintaining consistency in ConTracts and containing consistency violations. Existing workflow systems basically ignore those problems. Their vendors put it more mildly by saying the system provides the application with all the interfaces required to take care of consistency by itself. So, whereas many of the technical aspects related to distributed execution, naming, security etc. have been solved and made their ways into standards, consistency maintenance is still a hard problem, where research has to - and can - make a contribution. The paper will give an overview of how consistent execution is formally treated in the ConTract model.

5.1.1 The Motivation For ConTracts

First and foremost, ConTracts were designed to achieve reliable execution for long-running computations. In a sense, ConTracts were to provide a level of system support to such computations that is comparable to what transactions do for short interactive applications. Don't get this wrong: The level at which a distributed system supports both models is what is comparable; the actual concepts and techniques are quite different.

The first question to come up is: What is a long-lived computation, as opposed to a short (interactive) transaction? It certainly does not help to set a fixed elapsed time limit, such as: Whatever completes faster than within 10 seconds is a short computation, everything else is a long-lived computation. Either there are counter-examples on both sides of the limit, or the limit is set so high (or so low) that it becomes irrelevant. So we better define long-lived computations by their properties, especially those properties that lack support in current (operating) systems. Here is a list of some important traits that can be found in long-running computations such as workflow, but which do not hold for transactional applications:

- A long computation is one which cannot or should not be rolled back. The transactional style of aborting a failed computation implies the notion of retry: When a transaction has been aborted, just try again after system restart, or after having checked the input data, or whatever. Now, if restarting the computation is too expensive, or if it causes the application to miss a critical deadline, or if rollback is not feasible in the first place, we have a computation that must be continued rather than rolled back, even if something goes wrong: a long-lived computation.

- A long-lived computation must be kept alive across system shutdowns, reorganizations and other regular interruptions of normal system operation. In particular, a deactivation of all participating clients must not cause the computation to terminate.
- A long-lived computation involves many clients, mostly in the sense that it moves through the distributed system, activating one client interaction after the other. It must be possible, though, for two or more clients to be attached to the same long-lived computation simultaneously.
- A long-lived computation may not be specified completely at the moment it starts. Depending on its progress and some intermediate results its further plan will be developed as it progresses. In many cases, the decision about what to do next depends on the computation's own execution history.

Of course, some so-called long-lived computations do really go on for a very long time: If you consider everything related to the construction of, say, a power plant as processing one big order, the related computation that maps the order processing onto the distributed system will be active for a couple of years.

Once you accept the goal of providing system services that will make such long-running computations persistent in that they will automatically be recovered and continued as long as the application has not declared completion, the question is, which particular mechanisms are required, and how they interact with existing system services.

5.1.2 A Brief Survey of the Model

On first approximation, a long-running computation is just the execution of a program - a long one, for that matter. So if we had a persistent programming language, i.e. one which allows the program to be restarted after a crash right where it was interrupted, wouldn't that solve the problem?

It would indeed solve a portion of the problem, but leave out some important aspects. Of course, a persistent run-time environment is mandatory for achieving reliable execution of long-lived computations. But there is another side to this observation: Which functions are needed in a programming language that is suited for writing long-running programs? Will any one do, such as a persistent C?

We claim the answer is "no", and we hope some of the more subtle reasons can be appreciated by the end of the paper. A simple quantitative argument is the following: Classical programming languages have been designed for implementing software modules, which get invoked, perform their function within, say, 10ms and then return without leaving any context around. A workflow, on the other hand, can last for years, i.e. we look at elapsed times of 10^8 seconds. So the temporal horizon of programming a workflow is 10 to 11 orders of magnitude larger compared to implementing some module - it is quite obvious that the programming constructs adequate for the short range will not be completely sufficient for activities that are 100 million times longer.

In the following, we will briefly outline the additional mechanisms that have been introduced in the ConTract model to support "programming in the long".

First and most obviously, a long-running activity has explicit control flow, with all the constructs such as sequence, case, and loop. In addition, in long computations one typically finds many asynchronous (parallel) execution paths, so this must be part of the model. Fig. 5.1 shows a simple graphical representation of this.

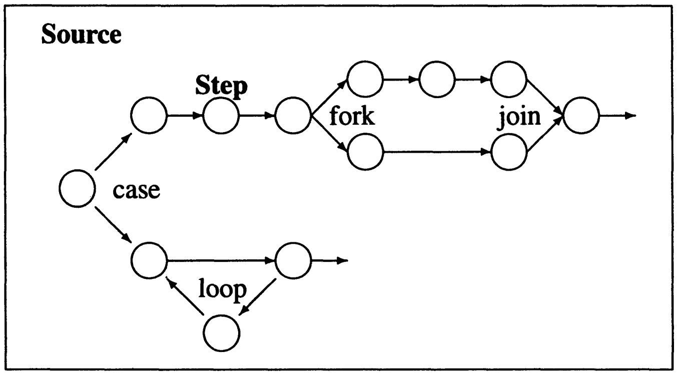


Figure 5.1 A ConTract script describes the control flow of a long-lived computation using all the basic constructs of a parallel programming language.

The ConTract model assumes that the nodes of the control flow graph (called "steps") are not single statements of some programming language (or base blocks); they rather represent programs, methods, applications, etc. which can be invoked through a call interface. Each such program comes with its own run-time system, executes in single-user mode and eventually returns control to the run-time system of the long-running computation[1]. So there is a clear division of responsibilities - a contract, if you will: The application is responsible for what happens inside a step, the system is responsible for keeping the control flow between steps alive, according to the specification. The control flow description is often referred to as "script" in workflow systems.

In order to support programming of truly long-lived computations, one needs more than persistent control flow, though. Fig. 5.2 shows the specifications that can be associated with a step in the ConTract model.

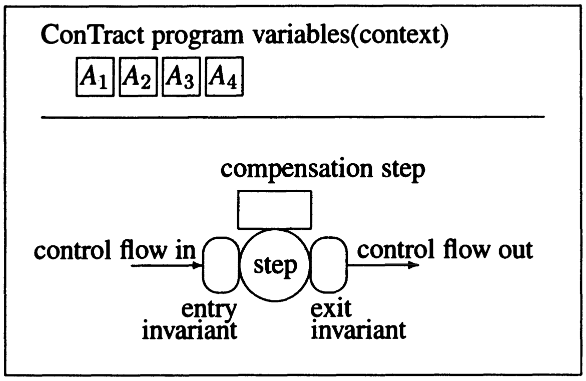


Figure 5.2 Control features associated with each ConTract step

First, each step must be complemented by a compensation step, which performs the (semantically) inverse function of the step. This is required, because in a long-running computation one cannot keep the updates locked until the end, as is the case in transactional systems.

Second, each entry point into a step is protected by a so-called entry invariant. This is a predicate expression, typically based on shared data in a database, that must evaluate to "true" in order to actually invoke the step procedure. So even if the control flow has arrived at a certain step, its invocation will not happen unless the entry invariant holds[2]. While compensation takes care of the fact that updates cannot be locked for a long time, entry invariants cope with the fact that data read by a long-running computation cannot be protected either.

Finally, there is a construct called exit invariant. It basically binds result values of a step to the variables in a predicate expression, thereby establishing the fact that a certain condition was fulfilled at that point in time. Steps that will be executed in the future can then refer back to such an exit invariant as part of their own entry invariant, checking whether something important has changed since "that step back there" executed.

Fig. 5.2 also shows an example of a local programming variable of a ConTract, i.e. a long-lived computation. Such variables are visible to all steps belonging to the same computation, but they are not visible to either the outside (other computations) or the inside (programs executing as a step). Since these variables reflect the execution history of a computation, which must be made accessible in an easy way, the ConTract model suggests a versioning scheme for all variables. So each assignment operation leaves the most recent value unchanged and creates a new version of the same variable. Since this is very different from normal program variables, they are called "context variables" in the ConTract model.

5.2 TRANSACTIONS IN A WORKFLOW ENVIRONMENT

This paper started out by saying that transactions are not adequate for modeling long-lived computations. On the other hand, they have the great virtue of providing a model for execution, failure, recovery, and synchronization in one simple formalism - an atomic state transition. There is no point in trying to relax or modify the transactional properties, hoping that the result will be a comparably simple model for long-lived executions. Each of the problems referred to by the ACID-properties has to be addressed individually for such environments, and the resulting architecture will not be as uniform and elegant as a transaction - but then, the problem to be solved is substantially more complicated.

So modifying the transaction model will not do, and ConTracts are not an extended transaction model. They will use transactions, though, in a variety of ways, which will be explained in this section.

5.2.1 Use of ACID-Transactions

ACID transactions are used in a ConTract environment at two levels of abstraction.

First, transactions appear at the control flow level. It is possible to let multiple steps execute as part of one (distributed) ACID transaction. This specification (which is not shown in Fig. 5.2) is part of the overall definition of steps and control flow. The default, enforced by the system in case the application does not explicitly specify transaction control, is the execution of each step as an ACID transaction. Of course, this has an effect only if the resource managers used by the step program do support transactions.

The second usage of transactions happens "under the covers" of a ConTract system, and it is totally unrelated to whether or not transactions are used at the control flow level.

Fig. 5.3 illustrates the basic idea. For simplicity, assume a linear control flow from step B to step C, which in general will be running on different nodes of the network. Once B has completed, the fact that it has completed must be reliably recorded - otherwise a system crash in that time window might cause B to be activated again. In addition, control must be transferred to step C, and the fact that all this has happened must be recorded at yet another node (called CM for ConTract manager). This makes sure that somebody will be there to initiate recovery in case the node executing C should crash before completion. Since all three actions must happen together or none must take effect, the ad-

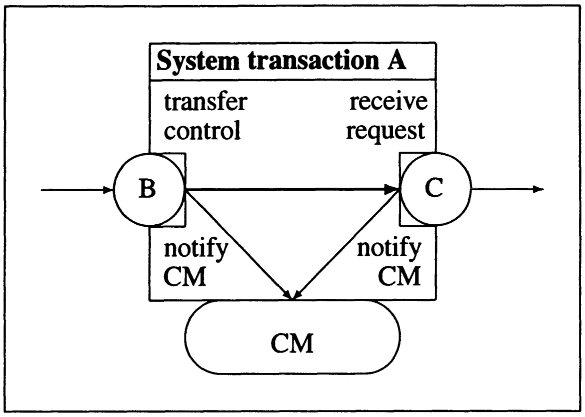


Figure 5.3 Distributed ACID transactions are the base mechanism for implementing reliable transfer of control.

equate implementation mechanism is a distributed ACID transaction involving the nodes of steps B and C, and the node running CM.

5.2.2 Semi-Transactional Activities

As mentioned in the previous section, there are and will be components which are not aware of something like a distributed transaction or a two phase commit protocol. However, many real world applications need these components to fulfill their tasks. Hence, the question comes up how ConTracts can cope with such *non-conforming* components and what the benefits will be.

The major benefit of using ConTracts even in case of non-conforming steps is the guarantee that state changes of the process (script) are made atomic. That means, that the effects of such steps regarding the process are protected by a transaction. In contrast, their effects to the "outside world" may be unprotected. That's the reason why we will call them *semi-transactional* in the following.

It is an obvious observation that without a transactional protection and without the control of the transaction by the system there will be intervals in time when the system is unable to determine the state of a step automatically. Hence, the system needs "help" from outside which results usually in a message to the administrator. However, a reliable system should minimize the "window in time" when human intervention is needed and provide as much information as possible to support humans while solving the problem.

To achieve the two objectives in ConTracts, several extensions to the original model are necessary. We will briefly introduce some of these extensions:

- The system must be aware of semi-transactional steps, i.e. the definition of the script has to contain a classification of steps such that the run-time system is enabled to determine necessary actions in case of problems.
- Recovery strategies are much more complex (but also flexible). The recovery of a semi-transactional step may require the execution of several other steps. This will be taken into account e.g. by the *partial compensation* in section 5.4.4.
- The notion of dependencies has to be extended. Up to now, control flow can be defined by using abort and commit dependencies.
- The usage of semi-transactional steps do impact the semantics of other parts of a script, e.g. transactions and compensation. Proper constraints have to be introduced to avoid indeterministic behavior.

The more we extend the features of ConTracts, the more information must be provided by the script programmer to use these features. Or in other words, the more you can use the transactional features of ConTracts, the less information is needed on the script-level.

5.3 RECONSIDERING CORRECTNESS

The original model of ConTracts [Waechter and Reuter, 1992] introduced an implicit notion of correctness by describing the properties of a ConTract in an informal fashion. In particular, the definition of the invariant based concurrency control mechanism was very brief which lead to confusion. Since workflow systems are more and more demanding transactional features, *execution models* like ConTracts need to come up with a very precise definition of their semantics.

5.3.1 Transactional Properties and ConTracts

The major benefit of classic database transactions was their simplicity, represented by the *ACID* properties [Gray and Reuter, 1993]. Unfortunately, these properties have major drawbacks in case of long-running executions like workflows [Gray, 1981]. Anyway, programmers of applications have to be supported by a proper abstraction like transactions to avoid the programming overhead for failure handling, recovery and multi-user anomalies. The ConTract model was introduced to provide such an abstraction with the following properties.

5.3.1.1 Recoverability. To avoid the shortcomings of the atomicity property, a two-layered recovery approach has been introduced:

1. Recovery at the step level
   Steps which are protected by a transaction are recovered by recovering

the surrounding transaction, i.e. active transactions are rolled back. For non-transactional steps a message to the administrator is generated. The administrator has to recover the step (either forward or backward) and then has to inform the ConTract processing system about the result of the recovery.

2. Recovery at the script level.
   A ConTract [3] is forward recoverable, i.e. after a failure the state of the script is recovered and then recovery is initiated for every step (and every transaction) which was active when the failure occured. After this first phase of recovery, the ConTract will continue its execution (forward recovery).

Recovery is handled by the run-time system of ConTracts except for non-transactional steps. Hence, a programmer of a ConTract does not need to provide any code for failure recovery.

Forward recovery is performed after any type of failure, in order to keep the ConTract going. It must be possible, though, for the application to terminate an active ConTract and ask the system to revoke what has been done so far; this type of recovery is called *compensation* [Gray, 1981].

Thus, a ConTract guarantees its compensability. The details of compensation in ConTracts will be described in section 5.4.1.

5.3.1.2 Permeability. Work performed by a ConTract is isolated in a transactional sense only while a transaction is executed. If the transaction finishes, all changes will become visible to the outside world if the application does not define any further restrictions by using the so-called *invariant concept.*

5.3.1.3 Consistency. The consistency property of transactions is based on the properties of atomicity and isolation. If a transaction runs isolated and atomic, and it is started on a consistent state, it produces a consistent state after it has finished its execution. During the execution a transaction may produce inconsistent states which are not visible to the "outside world". A basic assumption of this approach is that transactions have to check themselves if they violate any consistency constraints defined on the data (e.g. during the commit phase). If they encounter the violation of a constraint, they have to roll-back.

This notion of consistency has been extended. A ConTract may define intermediate states (during the execution) as consistent. So, intermediate results will become visible to other executions.

5.3.1.4 Durability. The notion of durability has also been extended in the ConTract model. The execution itself is durable, i.e. the state of the process and all variables (context) are durable. Furthermore, intermediate results which become visible during the execution do have the durability property.

5.3.2 Recovery and Serializability

The correctness criteria in the area of transaction processing systems [Bernstein et al., 1987] are derived from the properties of transactions:

1. *Serializability (SR)*
   Due to the isolation property, an execution history must be equivalent to a history which contains only the serial (non-interleaved) execution of transactions.

2. *Strictness (ST)*
   Due to the atomicity property of transactions, it has to be guaranteed that either the complete results of a transaction become visible to other transactions (commit case) or all effects are undone (rollback case). Since *cascading aborts* must be avoided, simple *recoverability (RC)* is not sufficient.

Since both criteria have to be guaranteed during the execution of a transaction, Alonso et al. [Alonso et al., 1994] came up with a unified criterion: the so-called *prefix reducibility (PRED)*.

In the following, we will define the correctness criterion used in ConTracts. Similar to the approach in [Alonso et al., 1994] we will develop a unified criterion for both, recoverability and permeability. The difference between transactional correctness and our approach hinges on the special notion of what a conflict is.

5.3.3 The Conflict Relationship

The core element of almost every correctness criterion is the definition of a *conflict relationship* between the basic operations of executions. As described in [Ramamritham and Chrysanthis, 1996] two classes of conflicts can be distinguished:

1. Conflicts between operations of the same execution must be handled by the structural dependencies of the operations (control flow) which are defined at programming-time.

2. Conflicts between operations of different executions are due to a conflict relation. The conflict relation can be used by a scheduler to generate only correct schedules.

A basic assumption of all the criteria is that an execution will be correct if it is the only execution in the system; this corresponds to the C of "ACID". Hence, execution histories which are equivalent to a serial execution history will be correct. In essence, this means that almost every correctness criterion is based on some sort of serializability.

Since ConTracts are not isolated the way transactions are, conflicts are due to explicitly defined constraints - the *invariants*. To get this straight, some helpful definitions are introduced to give a better understanding of what invariants are and how they are used.

Definition 5.1 (Path) *$\langle a,b \rangle$ says there exists a direct path from step a to step b. This means step b must be executed directly after step a. If there is a path $\langle a,b \rangle$ and a path $\langle b,c \rangle$ then we say that there is a path $\langle a,c \rangle^+$ (transitivity). And if either $\langle a,b \rangle$ or $\langle a,b \rangle^+$, we will use $\langle a,b \rangle^*$*

The concatenation of paths $\oplus$ is defined as $\langle a,b \rangle^* \oplus \langle b,c \rangle^* = \langle a,c \rangle^*$.

We will use $\langle b,c \rangle^* \in \langle a,d \rangle^*$ to denote the fact that $\langle a,d \rangle^*$ can be written as: $(\langle a,b \rangle^* \oplus \langle b,c \rangle^*) \oplus \langle c,d \rangle^*$.

Paths define the structural dependencies of steps, i.e. the flow of control (see also [Schwenkreis and Reuter, 1996]). By convention, it is allowed to use the special notation $\langle start, a \rangle^*$ to denote the path from the start of a ConTract to the step a.

Definition 5.2 (Step execution) *The successful execution of a step f transforms a state of data objects s (see [Bernstein et al., 1987]) to another state s'. We will use $f(s)$ to denote the state s', i.e. the state produced by step f.*

Definition 5.3 (Exit invariant) *An exit invariant i_x^s of a step s is a conjunction of predicates p_i*

$$i_x^s = p_1 \wedge p_2 \wedge \ldots \wedge p_n$$

We will use $p_k \in i_x^s$ to denote the fact that p_k is one of the predicates of i_x^s.

If a step is executed it checks whether its exit invariant holds and requests the system to ensure that it will not be violated (called *establishing* an invariant). If the exit invariant is not fulfilled at the end of a step, the step will be rolled back.

Definition 5.4 (Predicate reference) *A predicate reference $r(i_x^s, p_k)$ is a predicate with the following property:*

$$p_k \in i_x^s \wedge r(i_x^s, p_k) \Leftrightarrow p_k$$

Predicate references can be used in a ConTract definition to "point" to a predicate established by a previous step (see section 5.1.2).

Definition 5.5 (Entry invariant) *An entry invariant i_n^s of a step s is a conjunction of predicate references r_j:*

$$\begin{gathered} i_n^s = r_1 \wedge r_2 \wedge \ldots \wedge r_3 \\ \wedge \\ \forall r_j : \; r_j = r(i_x^a, p_k) \wedge \\ \exists \langle a,s \rangle^* \text{ such that } \langle s,a \rangle^* \notin (\langle start,a \rangle^* \oplus \langle a,s \rangle^*) \end{gathered}$$

We will use $r \in i_n^s$ to denote the fact that r is one of the predicate references of i_n^s.

An *entry invariant* can be used to define a condition which is needed by a step as a prerequisite for a successful execution.

Entry invariants can only be defined by using predicate references which refer to predicates of exit invariants of previous steps, i.e. a constraint needed by a step s_2 must be established by a previous step s_1.

In the following we will use $i(s)$ to denote the result of the evaluation of the predicate i at a current state s.

Given the definitions of invariants the special conflict relation of ConTracts can be introduced:

Definition 5.6 (Potential conflict) *A step a is in a potential conflict with a step b denoted by $conf(a,b)$ if there exists a state s with:*

$$i_x^a(s) \Rightarrow \neg i_x^a(b(s)) \lor i_n^a(s) \Rightarrow \neg i_n^a(b(s))$$

A step a is in conflict with another step if its invariants may be violated by the other step. Note that this notion of conflict is not symmetric.

5.3.4 Execution Histories and Correctness

Loops which are defined in the ConTract instance will be *un-rolled* at run-time. Therefore, the system generates so-called *step instances* from steps to be able to distinguish multiple executions of the same step.

Definition 5.7 (Step instance) *A step instance $\tilde{s}^i$ is a run-time version of a step s. $\tilde{s}^i$ has the semantics and effects of s and has the same invariants. The index i denotes the i-th instantiation of step s. We will use $\tilde{s}^{current}$ to denote the most recent instantiation of a step during the execution of a script (or zero if it is the first time).*

The following rules are used to execute a script and preserve the ordering defined in the script:

Definition 5.8 (Script-conform execution) *An execution of a ConTract is script-conform if the following rules are used to interpret a script:*

1. *At the start of a ConTract the system generates instances for all steps s_i which do not have a predecessor step $(\forall s_i : \neg \exists \langle a, s_i \rangle)$.*

2. *If a step instance $\tilde{s}$ has finished its execution, the system looks for all successor steps a_i which can be executed $(\forall a_i : \exists \langle s, a_i \rangle)$. A new instance $(\tilde{a}_i^{current+1})$ is created and executed for each of these steps.*

With the execution algorithm of ConTract the history of a ConTract processing system can be defined.

Definition 5.9 (Processing history) *The history H of a ConTract processing system is a set $\tilde{S}$ of step instances $\tilde{s}$ and a partial order $\prec$ defined over the set of step instances $H = (\tilde{S}, \prec)$. The set of step instances may also contain special step instances EOC which indicate the end of a ConTract. $EOC(\tilde{s})$ will be used to denote the End-of-ConTract step of the ConTract which has executed $\tilde{s}$. The ordering relation $(\tilde{s}_i \prec \tilde{s}_k)$ says that $\tilde{s}_i$ was executed before $\tilde{s}_k$.*
We will use H_C to denote the reduced history of a single ConTract C.

Similar to the approach in [Alonso et al., 1994] the history can be expanded to include the compensational semantics of ConTracts.

Definition 5.10 (Expanded processing history) *Let $H = (\tilde{S}, \prec)$ be a history. Its expansion $\hat{H}$ is a tuple $(\hat{S}, \hat{\prec})$ where:*

1. *$\hat{S}$ is a set of step instances which is derived from $\tilde{S}$ in the following way:*
   (a) *For each ConTract $C_i \in H$, if $\tilde{s}_i \in \tilde{S}$ then $\tilde{s}_i \in \hat{S}$.*
   (b) *For all $\tilde{s}_i \in \tilde{S} \wedge EOC(\tilde{s}_i) \notin \tilde{S}$, a compensating step instance $\tilde{s}_i^{-1}$ must appear in $\hat{S}$.*
2. *The partial order, $\hat{\prec}$, is determined as follows:*
   (a) *For every two step instances, $\tilde{s}_i$ and $\tilde{s}_k$, if $\tilde{s}_i \prec \tilde{s}_k$ then $\tilde{s}_i \hat{\prec} \tilde{s}_k$.*
   (b) *All non-compensating step instances of a ConTract must precede the compensating step instances of this ConTract.*
   (c) *For every two compensating step instances, $\tilde{s}_i^{-1}$ and $\tilde{s}_j^{-1}$, if $\tilde{s}_j \prec \tilde{s}_i$ then $\tilde{s}_i^{-1} \hat{\prec} \tilde{s}_j^{-1}$*

The expanded history contains all step instances of the original history. Additionally, for all running ConTracts, the history is expanded by all compensating step instances of all non-compensating step instances. The order of the compensating step instances is the reverse order of their original steps.

Now that we have introduced the notion of histories, the conflict relation of definition 5.6 can be refined

Definition 5.11 (Specific conflicts) *A step instance $\tilde{a}$ is in a conflict with another step instance $\tilde{b}$ of a different ConTract due to a predicate p_k of an exit invariant, denoted by $conf_x(\tilde{a}, \tilde{b}, p_k)$ if:*

$$\exists p_k \in i_x^{\tilde{a}} \text{ with } \neg\, p_k(\tilde{b}(s)) \;\wedge\; \tilde{a} \,\hat{\prec}\, \tilde{b} \;\wedge$$
$$\exists \tilde{c} \in \hat{S} \text{ with } \exists r_j \in i_n^{\tilde{c}} \wedge r_j = r(i_x^{\tilde{a}}, p_k) \text{ where } \neg r_j(\tilde{b}(s)) \;\wedge\; \neg\, \tilde{c} \,\hat{\prec}\, \tilde{b}$$

A step instance $\tilde{a}$ is in conflict with another step instance $\tilde{b}$ of a different ConTract due to a predicate p_k referenced by an entry invariant, denoted by $conf_n(\tilde{a}, \tilde{b}, p)$ if:

$$\exists r_j = \in i_n^{\tilde{a}} : \text{ with } \neg\, r_j(\tilde{b}(s)) \;\wedge\; \tilde{b} \,\hat{\prec}\, \tilde{a} \;\wedge$$
$$\exists \tilde{c} \in \hat{S} \;\wedge\; r_j = r(i_x^{\tilde{c}}, p_k) \;\wedge\; \tilde{c} \,\hat{\prec}\, \tilde{b}$$

Since the invariant mechanism is based on a paradigm similar to a producer / consumer relationship, a real conflict may only arise if an invariant invalidating step is executed in between a step which established a part of the (exit) invariant and a step which needs the established constraint (a part of the entry invariant).

With this more specific definition of conflicts of steps (or step instances) the binary conflict relation of ConTracts becomes obvious:

Definition 5.12 (Invariant-based ordering) *A ConTract C_A is in conflict with another ConTract C_B due to a predicate p, denoted by $C_p(C_A, C_B)$ if there are two step instances $\tilde{s}^A, \tilde{s}^B$ of these ConTracts in $\hat{S}$ where:*

$$conf_x(\tilde{s}^A, \tilde{s}^B, p) \vee conf_n(\tilde{s}^B, \tilde{s}^A, p)$$

An expanded history $\hat{S}$ implicates a partial order of ConTracts based on the conflict relation of definition 5.11. As in every serializability based criterion the last step is to define the correctness criterion of a history.

Definition 5.13 (Invariant-based serializability) *A history $\tilde{S}$ is correct if its expanded history $\hat{S}$ fulfills the following constraints:*

1. *The history of every single ConTract was generated by a script conform execution (see definition 5.8).*

2. $\forall C_k \in \hat{S}, \forall p : \neg\, C_p^*(C_k, C_k)$

There are some implications of this correctness criterion which should be mentioned.

- It can be shown that the correctness criterion is prefix closed, i.e. if a history is correct, it implicates that every prefix of the history is correct. Hence, it can be directly used for a scheduler; even though, it will never be implemented using the classical scheduling approach.

- Basically, the criterion differentiates between two classes of invariants - invariants for compensating steps and invariants for non-compensating steps.

- Invariant-based serializability does not force serializability for ConTracts as a whole. Only parts protected by an exit-/entry-invariant "bracket" do have the serializability property.

- One of the interesting features of the invariant concept, the selection of policies (*cooperation of ConTracts*), is currently not taken into account and will be covered by future extensions. Since the criterion is mainly intended to ensure the compensability of ConTracts, these extensions will only result in minor changes.

5.4 COMPENSATION IN DETAIL

Compensating activities as introduced in section 5.3.1.1 are a very common approach to realize undo behavior for long-running executions [Elmagarmid, 1992, Garcia-Molina and Salem, 1987]. Although the mechanism is used in almost every *advanced transaction model*, it is introduced in a very informal way.

5.4.1 A Basic Definition of Compensation

The idea of compensation in the area of transactions came up when it was realized that atomicity/rollback is not applicable in case of long-running executions [Gray, 1981]. The first attempt to formalize compensation was presented in [Korth et al., 1990b] which tried to unify rollback and compensation. The resulting notion of compensation was very restrictive in terms of what compensating (trans-)actions have to guarantee: Compensating activities as defined in [Korth et al., 1990b] have to generate a state of the accessed data objects which is identical to the state at the point in time the original activity started, i.e. objects in the database(s) must have the same value.

Observations of the real world have shown that compensating actions usually do not reestablish a previous state (of data). In particular, they do not reestablish the state at the start of the original activity. Hence, compensation is a very flexible means and almost similar to the forward running case. However, a simple property of compensation motivates the need to distinguish compensating activities from usual ones:

Compensation must not (finally) fail

To be more specific, if a compensation is needed, there is no way to execute an alternative like another compensation. Hence, if a run-time system cannot execute a compensating activity, the only thing it can do is to inform the administrator.

Definition 5.14 (Acceptance function) *There exists a function g_f (acceptance function) for every step f which maps a state s to a boolean value:*

$$g_f : s \to TRUE, FALSE$$

The acceptance function checks whether the state s satisfies the constraints of f in order to be executed successfully.

With the introduced definitions of paths and of acceptance functions the semantics of compensation can be defined.

Definition 5.15 (Compensation) *f_c is a compensation step of step f if:*

$$\begin{aligned} g_f(s) &= TRUE \Rightarrow g_f(f_c \circ f(s)) = TRUE \wedge \\ &\langle f, f_c \rangle^* \wedge g_{f_c}(f(s)) = TRUE \end{aligned}$$

In the general case $comp(a,b)$ is used to indicate that step a is a compensation of step b.

The acceptance function of f_c returns true if applied to the state after the execution of $f(s)$ (denoted by $f_c \circ f(s)$ in the above definition). The compensating function f_c generates a state which fulfills the requirements of the acceptance function of f.

Problem:

The execution of the compensating step f_c after the execution of the original step f can be intervened by other steps a^i. Hence, precautions must be taken to guarantee that the acceptance function of f_c is not violated such that f_c cannot be executed in the future.

A criterion which does not make any assumptions about the usage of steps inside of a ConTract is very restrictive. It does not allow the violation of the acceptance function of the compensating steps during the execution of a ConTract. This is taken into account by definition 5.16.

Definition 5.16 (Indirect compensability) *A step f executed by a ConTract is indirectly compensable (denoted $C_I(f,a)$) with regard to another step of the same ConTract $a \in CSteps(f)$ if:*

$$comp(a,f)$$
$$\vee$$
$$\langle f,a\rangle \wedge \langle a_c,f_c\rangle \wedge (g_{f_c}(s) = TRUE \Rightarrow g_{f_c}(a_c \circ a(s)) = TRUE)$$

$$C_I(f,a^i) \wedge C_I(a^i,a^j) \Rightarrow C_I^+(f,a^j) \text{ (transitivity)}$$
$$C_I(f,a^i) \vee C_I^+(f,a^i) \Rightarrow C_I^*(f,a^i)$$

The violation of the acceptance function of a compensating step f_c can be allowed, if the step a, which causes the violation, belongs to the same ConTract. It must be executed after the original step f, and it must be guaranteed that its compensation a_c is executed before f_c to reestablish a state which satisfies the acceptance function of f_c. One implication of this property is that $g_{f_c}(a_c(s)) = TRUE$. The criterion holds also for a compensation step, if its original step satisfies the criterion.

Definition 5.17 (Indirect compensation chain) *The ordered set of all a^k with $C_I^*(f,a^k)$ is called indirect compensation chain of f ($I_c(f)$):*

$$I_c(f) = \{a^k \mid C_I^*(f,a^k)\} \text{ and } a^i < a^j \text{ if } C_I^*(a^i,a^j)$$

The reduced chain $I_c^r(f)$ containing only non-compensating steps can be directly derived:

$$I_c^r(f) = I_c(f) \setminus \{a^k \mid comp(a^k,a^j), j<k, a^j \in I_c(f)\}$$

The indirect compensation chain consists of all steps for which the indirect compensation relation holds. The reduced chain omits all compensation steps contained in $I_c(f)$

Definition 5.18 (Absolute compensability) *A step f executed by a ConTract is absolutely compensable with regard to an arbitrary step a (denoted $C_A(f,a)$) if:*

$$\neg C_I(f,a) \wedge$$
$$g_{f_c}(s) = TRUE \Rightarrow g_{f_c}(a(s)) = TRUE \wedge$$
$$\forall b^k \in I_c^r(f) : \; C_A(b^k,a)$$

$$C_A(f,a^i) \wedge C_A(a^i,a^j) \Rightarrow C_A^+(f,a^j) \; (transitivity)$$

A step f is absolute compensable with regard to another step a, if it is ensured that the acceptance function of the compensation function f_c is not violated by the execution of a. Additionally, step a must not violate the acceptance functions of the compensation steps belonging to the steps in the reduced indirection chain in order to preserve the possibility to reestablish a proper state for f_c.

Based on our notion of compensability (definition 5.15) we can prove that the introduced criteria are sufficient to guarantee the compensability of ConTracts.

Given an arbitrary point in time after the execution of a step f of a ConTract, we will find a state s produced by the execution of an ordered set of steps denoted by $(a^n \circ \ldots \circ a^1 \circ f(s))$. In the following we will use $A = \{a^1, \ldots, a^n\}$ to denote the set of steps executed after f.

Theorem 1 (Execution dependent compensability) *If all steps which have been executed successfully after f either preserve absolute compensability or indirect compensability, it is guaranteed that the compensating step f_c is executable when it has to be executed.*

Proof:

We will prove the theorem above by an induction over the set A.

1. Basic assumption:

   If A is empty the current state is f(s), then $g_{f_c}(f(s)) = TRUE$ (def. 5.15), i.e. the compensating step of f can be executed if directly applied to f(s).

2. Conclusion:

   If A has n elements $n \in N_0$ and $g_{f_c}(a^n \circ \ldots \circ a^1 \circ f(s)) = TRUE$ we have to prove that $g_{f_c}(a^{n+1} \circ a^n \circ \ldots \circ a^1 \circ f(s))$ is true, when f_c has to be executed. If $g_{f_c}(a^n \circ \ldots \circ a^1 \circ f(s)) = FALSE$ it must be ensured that g_{f_c} will become $TRUE$ when f_c has to be executed.

(a) a^{n+1} fulfills $C_A(f,a^{n+1})$ and $g_{f_c}(a^n \circ \ldots \circ a^1 \circ f(s)) = TRUE$:
Since $g_{f_c}(a^n \circ \ldots \circ a^1 \circ f(s)) = TRUE$ and $C_A(f,a^{n+1})$, it is ensured that $g_{f_c}(a^{n+1} \circ a^n \circ \ldots \circ a^1 \circ f(s)) = TRUE$ (see definition 5.18). Hence f_c can be executed successfully. □

(b) a^{n+1} fulfills $C_A(f,a^{n+1})$ and $g_{f_c}(a^n \circ \ldots \circ a^1 \circ f(s)) = FALSE$:
There must be steps a^k in A which satisfy $C_I^+(f,a^k)$. The *executability* of f_c is guaranteed by guaranteeing the executability of the steps belonging to $I_c^r(f)$. Since a^{n+1} preserves the executability (def. 5.18) of all the compensation steps in $I_c^r(f)$, the acceptance g_{f_c} will become *TRUE* when f_c has to be executed. □

(c) a^{n+1} fulfills $C_I^*(f,a^{n+1}) \wedge g_{f_c}(a^n \circ \ldots \circ a^1 \circ f(s)) = FALSE$:
In this case, there must be a step b with $C_I(b,a^{n+1}) \wedge b \in I_c^r(f)$ for which $g_{b_c} = TRUE$ (see def. 5.16). Since $C_I(b,a^{n+1})$ guarantees that g_{b_c} will become *TRUE* after the execution of a_c^{n+1}, and a_c^{n+1} must be executed before b_c the executability of f_c is guaranteed when it can be executed (sometimes after the execution of b_c). □

(d) a^{n+1} fulfills $C_I^*(f,a^{n+1}) \wedge g_{f_c}(a^n \circ \ldots \circ a^1 \circ f(s)) = TRUE$:
This is the case where a step probably violates the acceptance function f_c. Two sub-cases can be distinguished:

- There exists a step $b \in A, b \neq f$ with $C_I(b,a^{n+1}) \wedge C_I^*(f,b)$.
This case can be treated similar to the previously dicussed case. □

- $C_I(f,a^{n+1})$:
Since the precedence relation of definition 5.16 guarantees that a_c^{n+1} will be executed before f_c and reestablishs a state where $g_{f_c} = TRUE$, the executability of f_c is guaranteed when needed.□

5.4.2 Script-based Compensation

Compensating a step can be a complex task with several branches in the control flow. Moreover, compensating activities can contain real actions and require interactions. In some cases several machines and people are involved in the execution of the compensation. This requires to allow (sub-)scripts as compensations rather than simple steps only.

Script-based compensation has the following additional advantages compared to a simple step-based compensation.

- Script-based compensation allows the use of forward recovery in case of failures during the execution of the compensations.

- In case of the final failure of compensation the usage of a script-based compensation leaves more parts in an consistent state and requires less manual intervention.

- The parts of a compensating script which were finished successfully, become durable. Hence, results of the compensation become available as early as possible.

Introducing script-based compensation leads to a slightly extended programming model. The implementation of a script-based compensation can contain - as the name suggests - script level code. Still, the activity compensated by a script is a single step and basically this step code has to be understood. In a workflow environment, on the other side, there are many constraints to be met by a compensation, for example constraints derived from informational, behavioral, administrational, technical or organizational aspects [Curtis et al., 1992]. One example is the selection of different counter-actions, that are needed for the cancelation of a flight reservation, depending on the point in time and state of the execution. This behavioral aspect is best expressed on script level.

Script-based compensations are expressed as a set of steps with a defined control flow resembling a block.

Definition 5.19 (Block) *Let A be a set of steps and $\langle\rangle^*$ the binary path relation from definition 5.1. A tuple $B = (A, \langle\rangle^*)$ is a Block if:*

$$\begin{gathered}
(\exists a \in A : ((\forall c \in A \backslash \{a\} : \langle a,c\rangle^*) \wedge (\forall c \in A : \neg\, \langle c,a\rangle))) \\
(\textit{We will use } s_B \textit{ to denote the step of the block } B \textit{ with this property}) \wedge \\
(\exists b \in A : ((\forall c \in A \backslash \{b\} : \langle c,b\rangle^*) \wedge (\forall c \in A : \neg\, \langle b,c\rangle))) \\
(\textit{We will use } e_B \textit{ to denote the step of the block } B \textit{ with this property}) \wedge \\
(\forall a \in A \backslash \{s_B, e_B\} : (\neg\, \exists b \notin A : \langle a,b\rangle \vee \langle b,a\rangle)) \wedge \\
((\forall a \notin A : (\neg\, \langle a,e_B\rangle \ \vee\ \neg\, \langle s_B,a\rangle)) \ \vee\ (s_B = e_B))
\end{gathered}$$

We are referring to all steps of a block B with the notation BSteps.

As mentioned above, the control flow in a ConTract is defined by paths. For the moment, we make no further assumptions about the events, transitions, context or final states that are defined in a process except the existence of paths. In the following we take as a basis the flow of control in the case of no failures.

The structure of a block has to fulfill some requirements. Exactly one step starts the block and a path exists from this step to any other step in the block. There is exactly one step at the end of the block, and a path exists from any other step in the block to this step. Only the first step and the last step are allowed to have direct predecessors or direct successors outside of the block, respectively. If the block contains only a single step, the first step is the last step.

Definition 5.20 (Acceptance function of a block) *The acceptance function of the start step of a block B is also the acceptance function of B. We will use g_B to denote the acceptance function of a block B.*

Definition 5.21 (Script-based compensation) *A block $B = (A, \langle\rangle^*)$ is a script-based compensation of step f if:*

$$\begin{gathered}\langle f, s_B\rangle^* \wedge \\ g_B(f(s)) = \mathit{TRUE} \wedge \\ \forall a,b \in A \text{ with } \langle a,b\rangle : g_a(s_i) = \mathit{TRUE} \Rightarrow g_b(a(s_i)) = \mathit{TRUE} \wedge \\ g_f(s) \;=\; g_f(B \circ f(s))\end{gathered}$$

We will use $comp(B,a)$ to indicate that block B is a compensation of step a.

Assuming the compensation definition contains only a single step it is equivalent to the standard step-based compensation. It has to be mentioned that a compensation definition is not allowed to contain compensating steps for the included activities.

A script-based compensation does not change the correctness criterion of the execution of a ConTract (def. 5.13). It has to be ensured that the compensation of every successfully finished step is executable. The state after the execution of a step in the ConTract has to fulfill the requirements of the acceptance function of the start step of its compensation. The state after the execution of a step within the compensating block has to fulfill the acceptance function of its direct successors. It may be useful to allow only certain structures for the compensating block (e.g., only sequences of steps).

5.4.3 Comprehensive Compensation

So far the compensating activities relate only to single steps. But there are situations, especially in workflows, where it is suitable to compensate a sequence or a group of steps with a single compensating activity. We call this comprehensive compensation. For example, if there are some activities, which together create and work on a complex document, it is most efficient to compensate them all together by destroying the whole document. It is possible for a comprehensive compensating activity to invalidate the associated compensating activities of previous steps.

Furthermore observations of the real world have shown that the point in time a compensation is initiated is very important. And not only the actual time for compensation is important, but although the state of the execution of the ConTract. Franking a letter can be compensated separately as long as the letter is not dispatched. After that, the charges for the stamps are lost. This means compensating activities associated to previous activities will not be needed anymore. Depending on the actual state of the execution a dynamic selection of the valid compensating activities is required.

These examples motivate the compensation of groups of steps as a whole and the dynamic determination of compensating activities. Of course, it is not practical to allow compensating activities for any arbitrary group of steps. Only blocks can be compensated by the corresponding compensating blocks.

Definition 5.22 (Comprehensive compensation) *A block $B_c = (A_{B_c}, \langle\rangle^*)$ is a comprehensive compensation of block $B = (A_B, \langle\rangle^*)$ if:*

$$\begin{gathered}\langle e_B, s_{B_c}\rangle^* \wedge \\ g_{B_c}(B(s)) = TRUE \wedge \\ \forall a, b \in A_{B_c} \text{ with } \langle a, b\rangle : g_a(s_i) = TRUE \Rightarrow g_b(a(s_i)) = TRUE \wedge \\ g_B(s) \ = \ g_B(B_c \circ B(s))\end{gathered}$$

We will use $comp(A, B)$ to indicate that block A is a compensation of block B.

Definition 5.23 (Compensable block) *A block in a ConTract which has an associated compensating block is a compensable block. We will use CBlocks to denote the set of compensable blocks of ConTract C.*

The correctness criterion (def. 5.13) ensures that each set of successfully completed steps can be compensated. We have to adapt the correctness criteria for compensation [Waechter and Reuter, 1992] to the enhanced definition of compensation. So far it was only necessary for each step in the script to have exactly one valid compensating step. Now the requirement of a deterministic and unambiguous compensation for each partial execution of the ConTract is more difficult to fulfill. For every set of successfully completed steps we need an unambiguous disjunctive partitioning into compensable blocks. If that is guaranteed, the correctness criteria for the execution of a ConTract (def. 5.13) can still be fulfilled. It is sufficient to ensure the existence of the compensations for the definition of a script. The actual compensation depends on the set and order of successfully executed step instances and must be determinable at runtime.

Definition 5.24 (Compensable ConTract) *A ConTract C is compensable if:*

$$\begin{gathered}\forall B_1 \in CBlocks : (\forall B_2 \in CBlocks \backslash \{B_1\} : e_B \neq e_{B_2}) \wedge \\ \forall a \in CSteps : (\exists B \in CBlocks : a = e_B)\end{gathered}$$

Every single step of the ConTract is the end step for exactly one single compensable block. This can be tested for the definition of a script.

Definition 5.25 (Block instance) *A block instance $\tilde{B}^i$ is a run-time version of Block B. $\tilde{B}^i$ denotes the i-th instantiation of block B. $\tilde{B}^i$ contains the instantiation of every step in block B generated for the i-th successful execution of block B. Not necessarily an instance for every step in B shows up in $\tilde{B}^i$, because some steps might not have been executed. Instances for the start step and the* *end step of B are always included in $\tilde{B}^i$. $\tilde{C}Blocks$ denotes all block instances of ConTract C in the history.*

It has to be mentioned that a single step instance can belong to more then one block instance.

Definition 5.26 (Partitioning) *P is a partitioning of the set of step instances of a ConTract C into block instances if:*

$$\begin{gathered} P \subseteq \tilde{C}Blocks \wedge \\ \forall B_1, B_2 \in P \text{ with } B_1 \neq B_2 : (B_1 \cap B_2 = \{\}) \wedge \\ \forall \tilde{s} \in H_C : (\exists \tilde{B} \in P : \tilde{s} \in \tilde{B}) \end{gathered}$$

A unique partitioning into compensable block instances is always possible for a compensable ConTract. This partitioning can be efficiently computed using the order of block instances in the history. Reversing this ordering the compensation can be derived directly from the partitioning.

Definition 5.27 (Partial order of block instances) *Let $H = (\tilde{S}, \prec)$ be a history of a ConTract processing system. Let $\tilde{B}$ be a set of block instances. A binary relation $\overset{H}{\prec}$ is defined on $\tilde{B}$ with ($\forall\ \tilde{B}_1, \tilde{B}_2 \in \tilde{B} : (\tilde{B}_1 \overset{H}{\prec} \tilde{B}_2 :\Leftrightarrow \tilde{e}_{\tilde{B}_1} \prec \tilde{e}_{\tilde{B}_2}$)).*

We will briefly describe how to determine the unique partitioning. The step instances are sorted by their completion time in the history, which in turn is determined by the flow of control. We take as the basis the reduced history of the ConTract. A compensable block instance of the ConTract becomes valid with the successful completion of its last step.

The following actions will be repeated until H_C is empty and all valid compensable block instances are determined.

1. We select the last completed step instance from the reduced history; that is, a step without successor in the history. If there are several such steps, we take one of them randomly. This step instance determines an instance of a valid compensable block.

2. All step instances associated to this block instance are removed.

It can be easily shown, that the computed partitioning is the only possible partitioning with H_C and $\tilde{C}Blocks$.

After the valid compensable block instances are determined, their associated compensation can be executed according to the inverse order of block instances.

The definitions of indirect compensability (def. 5.16), of absolute compensability (def. 5.18), and of the indirect compensation chain (def. 5.17) are adapted accordingly. The theorem of execution dependent compensation (theorem 1) can be applied to the modified definitions.

5.4.4 Partial Compensation

The analysis of processes in workflow environments has shown that sometimes it is necessary or suitable to go back to an earlier state of the execution and to proceed with the process in a different way than before. For example, it is a good idea to periodically confer with the customer during the planning phase of a power plant. If the customer disagrees with the actual plans, all effects which where introduced since the last agreement have to be compensated.

Supporting partial compensation of a ConTract will change the execution model. Partial compensation can lead to a lot of complications. It is more difficult to ensure the correctness of the execution of a ConTract. Therefore partial compensation has to be applied with great care.

If we decide to reject only a part of a ConTract, we have to pay attention to the relationships between the blocks to be compensated.

Definition 5.28 (Compensation dependency) *A compensation dependency between a compensable block A and a compensable block B exists if:*

$$\textit{A and B finished successfully} \Rightarrow$$
$$\textit{it is allowed to compensate only both or none of them.}$$

The compensation dependency relation is transitive. We will use $A \bowtie B$ to denote that a compensation dependency between block A and block B exists.

On account of the application, dependencies between the compensable blocks may exist, such that the compensation of one block leads to the necessity of compensating the other. Hence, certain groups of compensable blocks always have to be compensated as a whole. These dependencies are modeled by the programmer. Additionally, all steps which are grouped into the same transaction are compensation dependent of each other. Moreover, only a part of a ConTract with a certain structure is allowed to be compensated separately to preserve correctness. We call a set of steps with this property a *compensable section.* Compensable sections are the subsets of the total set of steps of the ConTract, which we allow to be compensated separately.

Definition 5.29 (Partial ordering of blocks) *Let B be a set of blocks. A precedence relation $\prec$ is defined over B with $\forall A_1, A_2 \in B : (A_1 \prec A_2 :\Leftrightarrow (\langle e_{A_1}, e_{A_2}\rangle^*)$.*

Definition 5.30 (Compensable section) *A set of steps CS is a compensable section of ConTract C if:*

$$\begin{gathered}
CS \subset CSteps \wedge \\
\forall a \in CS : ((\exists B \in CBlocks : a = e_B) \Rightarrow BSteps \subseteq CS) \wedge \\
\forall a \in CS : ((\exists B \in CBlocks : a = e_B) \Rightarrow \\
(\forall B_iSteps \subseteq C : (B \bowtie B_i \Rightarrow B_iSteps \subseteq A))) \wedge \\
\forall B \in CBlocks : ((\exists B_iSteps \subseteq CS : B_i \prec B) \Rightarrow BSteps \subseteq CS)
\end{gathered}$$

The structure of a compensable section has to fulfill the following requirements. Either all steps of a compensable block are included in the compensable section or none at all. Inclusion of a compensable block leads to the inclusion of all compensation dependent blocks. A compensable section has to contain all successors of its included compensable blocks.

Due to limited space, we omit the definition for the compensable section for block instances. However, this definition can be derived straight forward from definition 5.30. To determine a compensable section at runtime the actual disjunctive partition of a history H_C is taken as the basis. Either all steps of a valid compensable block instance are included or none at all. Inclusion of a valid compensable block instance leads to the inclusion of all compensation dependent block instances as well as all of its successors in the current partition.

It is obvious that partial compensation does not compromise the correctness notion of ConTracts. Hence, a formal proof is not presented in this article.

5.5 SUMMARY

The ConTract model is not intended to be just another "extended transaction model". Instead, it has been developed (and maintained) to define a reliable basis for long-running executions like workflows. In the last five years we have spent a lot of effort to permanently evaluate our approach based on observations of the "real world". In result, many features have been made more concrete and others have been extended. For instance, the support of semi-transactional steps is one of the recent extensions while the internal usage of transactions has been revised several times already.

The presented correctness criterion introduces a formal basis to verify execution histories and to build up correctness ensuring mechanisms. We are strongly convinced that in mission-critical processes correctness will become more and more important. Hence, workflow systems have to commit guarantees regarding their behavior in case of failures and concurrency.

There is also a need to elaborate on compensation. The presented extensions are a first step to leverage the concept of compensation such that it can be used as a general-purpose mechanism as in real applications. In particular, the support of semi-transactional steps can be enhanced by a flexible compensation and advanced semantics of applications can be supported (e.g. durable parts).

In parallel to the evolution of the model itself the prototype implementation (APRICOTS) is continued to illustrate two things: We do build real systems to prove our concepts, and we still have a long way to go to prove the consistency-related concepts.

Notes

1. For ease of reference, this run-time environment will be referred to as the "ConTract manager".

2. Of course, something must happen in such a situation to keep the computation going. For details of this mechanism called "conflict resolution" see [Waechter and Reuter, 1992].

3. In the following we will use the word ConTract to denote an executable *instance* of a ConTract definition (a script or *ConTract template*).

6 SEMANTIC-BASED DECOMPOSITION OF TRANSACTIONS

Paul Ammann, Sushil Jajodia
and Indrakshi Ray

Abstract: Sometimes transactions must be decomposed into steps. The need for decomposition arises in a variety of different domains. For example, long duration transactions may be decomposed to improve performance, global transactions in multidatabases may be decomposed to preserve local database autonomy, and multilevel secure transactions may be decomposed to avoid leaking sensitive information. To achieve these various objectives, a decomposition sacrifices those desirable properties, namely atomicity, consistency, and isolation, that form the foundation of syntactically based correctness approaches such as conflict serializability. We remedy this loss by defining a semantic view of correctness organized around a new set of desirable properties that are specifically designed for reasoning about decompositions. The exact details of the semantic correctness properties depend on the domain being addressed; in this chapter, we focus on the long duration transaction domain. Using our method an application developer can show that a given decomposition indeed refines the original transactions in a satisfactory way. The semantic correctness properties are formulated in terms of semantic histories. For efficiency reasons, allowable interleavings of steps are described with syntactically specified successor sets. We discuss a two-phase locking based mechanism for realizing successor sets in a typical database system.

6.1 INTRODUCTION

Decomposing transactions into steps is a common method of achieving diverse goals in a variety of domains. To illustrate this point, we describe the decomposition of transactions in three domains, namely long-duration transactions, multidatabases, and multilevel secure databases. In each case, the decomposition undermines one or more of the foundation properties of syntactically-based correctness approaches. The properties are atomicity, consistency, and isolation, and they are used in correctness approaches such as conflict serializability. In this chapter we remedy the loss of these properties by developing a semantic approach to correctness, which we illustrate in the long duration transaction domain. We identify a set of replacement properties that are specifically designed to reason about decompositions and show how these properties can be established for a given application.

In database applications where some transactions are of long duration, performance requirements may dictate that execution histories be accepted even though operations of transactions interleave in ways that are not correct with respect to serializability criteria. For example, locks may be released early, or transactions may be split explicitly into steps. Consider the simple example of making a hotel reservation. A reserve transaction might consist of ensuring that there are still rooms vacant, selecting a vacant room that matches the customer's preferences, and recording billing information. Since the reserve transaction might last a relatively long time – for example, when the customer makes reservations by phone – it may be desirable to execute the three steps of the reserve transaction separately, thereby allowing other transactions access to key database objects. Some steps may be undesirable at sensitive points in a given execution history. For example, a report transaction may be undesirable if interleaved between certain steps in one or more reserve transactions. As will be subsequently illustrated in this chapter, our semantic approach can determine if a decomposition into steps is correct with respect to the original collection of transactions.

A multidatabase is an integrated collection of heterogeneous databases [Bukhres and Elmagarmid, 1996]. The constituent or local databases require both design autonomy to accommodate their diverse legacy nature and execution autonomy to ensure that local transactions are not unduly blocked by global transactions. Control of global transactions, which are decomposed into steps that execute at the local database, must be distributed to avoid bottlenecks and tolerate failure in the global database. Integrity constraints must be maintained, not only on each local database, but also on the global database. These goals cannot be achieved simultaneously with syntactic correctness criteria such as serializability, but a semantic based approach can determine if a given application does indeed have the desired properties.

In multilevel secure databases there is a need for multilevel transactions – transactions that both read and write at a range of security levels. A major outstanding problem with standard methods of handling multilevel transactions is the treatment of atomicity. Specifically, for a multilevel transaction decomposed into single-level sections there is no assurance that either all or none of the sections will be present in a given execution history. The chief difficulty is that a high section of a transaction may be unable to complete due to violations of the integrity constraints, and a rollback of sections at lower or incomparable levels can be exploited to implement a covert channel. For details of how a semantic approach to correctness can overcome this problem, see [Ammann et al., 1996].

The traditional transaction model relies on the properties of *atomicity*, *consistency*, and *isolation* [Bernstein et al., 1987]. Atomicity ensures that either all actions of a transaction complete successfully or all of the transaction's effects are absent. Consistency ensures that a transaction when executed by itself, without interference from other transactions, maps the database from one consistent state to another. Isolation ensures that no transaction ever views the partial effects of some other transaction even when transactions execute concurrently. Decomposing transactions into steps generally forces one to relinquish these three properties to some degree.

Decomposition of a transaction into steps sacrifices atomicity since the atomicity of the single logical action is lost. Interleaving steps of transactions exposes each to the partial effects of the others. Hence, if a transaction aborts after committing some of its steps, it may not be possible to remove all of its effects. This difficulty arises because transactions that read from the aborted transaction may have committed. In addition, the aborted transaction may have generated outputs. Thus traditional undo [Bernstein et al., 1987] is not possible; the solution is to semantically undo the actions of the aborted transactions. We achieve semantic undo with compensating steps [Garcia-Molina, 1983, Garcia-Molina and Salem, 1987].

Decomposition not only sacrifices atomicity, but also impacts consistency and isolation. Execution of a step may leave the database in an inconsistent state, which other transactions or steps may access, so it is necessary to reason about the interleavings of the steps of different transactions. Although the step-by-step decomposition of a single transaction into steps may be understood easily in isolation, reasoning about the interleaving of these steps with other transactions, possibly also decomposed into steps, is more difficult.

To remedy the loss of atomicity, consistency, and isolation, we develop properties suitable for reasoning about decompositions. These properties are enumerated in section 6.5. The properties are formulated in terms of *semantic* histories, which not only list the sequence of steps forming the history, but also convey information regarding the state of the database before and after execution of each step in the history. All possible semantic histories must satisfy the given properties for a particular decomposition to be considered acceptable, both when considered by itself, and also with respect to the original set of transactions.

We adopt the Object Z specification language [Duke and Duke, 1990] for expressing model-based specifications. Object Z is an extension of the Z specification language [Spivey, 1992] to include object oriented features. Object Z is based on set theory, first order predicate logic, and a schema calculus to organize large specifications. Knowledge of Object Z is helpful, but not required, for reading this chapter, since we narrate the formal specifications in English. Table 6.1 briefly explains the Object Z notation used in our examples. Other specification and analysis conventions specific to Object Z are explained as the need arises.

The rest of the chapter is organized as follows. Section 6.2 briefly describes the work related to semantic based transaction processing. Section 6.3 specifies an example application in Object Z. Section 6.4 describes our model. Section 6.5 describes the necessary and desirable properties of a correct decomposition. Section 6.6 gives examples of decompositions. Section 6.7 describes the notion of successor sets which is necessary for efficiently implementing our model. In section 6.8 we develop our correctness criterion for concurrent execution of transactions and present a concurrency control mechanism. Section 6.9 concludes the chapter.

6.2 RELATED WORK

The work on semantic based concurrency control can be classified into two major categories. In the first category [Herlihy, 1987, Herlihy and Weihl, 1991, Weihl, 1984, Weihl, 1988b] the authors exploit the semantics of operations to increase concurrency. Instead of using low level database operations like read or write to access the database objects, the authors propose the use of high level operations for this purpose. Commutativity of these operations, and not the read/write operations, is used to determine conflicts between transactions, resulting in more concurrency. In these works, the authors use serializability as the correctness criterion.

Our work falls in the second category [Agrawal et al., 1993, Farrag and Özsu, 1989, Garcia-Molina, 1983, Garcia-Molina and Salem, 1987] which is based on exploiting semantics of transactions to increase concurrency. In these works, the researchers decomposed transactions into steps and developed semantic based correctness criteria. Researchers have variously introduced the notions of transaction steps, countersteps, allowed vs. prohibited interleavings of steps, and implementations in locking environments. The focus is typically on implementing a decomposition supplied by the database application developer, with relatively little attention to what constitutes a desirable decomposition and how the developer should obtain such a decomposition. We find the decomposition process itself to be worthy of attention, so we give the developer a model in which to decompose transactions, and we define properties to assure the developer as to the soundness of a given decomposition. Only then

Table 6.1 Relevant Object Z Notation

$\mathbb{N}$	Set of Natural Numbers
$\mathbb{P}A$	Powerset of Set A
$\#A$	Cardinality of Set A
$\setminus$	Set Difference (Also schema 'hiding')
$A \mathbin{\S} B$	Forward Composition of A with B
$x \mapsto y$	Ordered Pair (x, y)
$A \rightarrowtriangle B$	Partial Function from A to B
$A \rightarrowtail\!\!\!\!\!\!\;\mapsto B$	Partial Injective Function from A to B
$B \lhd A$	Relation A with Set B Removed from Domain
$A \rhd B$	Relation A with Range Restricted to Set B
$\mathrm{dom} A$	Domain of Relation A
$\mathrm{ran} A$	Range of Relation A
$A \oplus B$	Function A Overridden with Function B
$x?$	Variable $x?$ is an Input
$x!$	Variable $x!$ is an Output
x	State Variable x before an Operation
x'	State Variable x' after an Operation
Δx	Before and After State of Variable x
$\Box$	Temporal Operator Always
$\Diamond$	Temporal Operator Eventually
$\bigcirc$	Temporal Operator Next
op	Operation

do we consider the problem of implementing our decomposition in a two-phase locking environment.

6.3 THE HOTEL DATABASE

We present our ideas with a running example of a hotel database. We use the *class* definition of Object Z to specify the hotel database. Syntactically, a class definition is a named box, in which the constituents of the class are defined and related. The constituents of a class include type and constant definitions, state schema, initialization schema, operation schemas, and history invariants. A schema is a two-dimensional notation used in Object Z to specify the state as well as operations on the state. The state schema is nameless and consists of two parts: the top part contains declarations of the state variables, and the bottom part specifies the constraints on these variables. The initialization schema, named *INIT*, defines the possible initial states. An operation schema, which is named after the operation, also consists of two parts: the bottom part specifies the operation using preconditions and postconditions and the top part contains declarations of the variables used in the bottom part. A history invariant is a predicate defined over a sequence of operations which further constrains the behavior of the objects.

An Object Z specification appears in figure 6.1. The specification assumes two basic types, *Guest* and *Room*, which enumerate all possible guests and all possible hotel rooms, respectively. The enumerated type *Status* lists the two values, *Available* and *Taken*, which indicates the status of each room.

$[Guest, Room]$

$Status ::= Available \mid Taken$

Hotel

$total : \mathbb{N}$

$res : \mathbb{N}$

$ST : Room \mathrel{+\!\!\!\!\rightarrow} Status$

$RM : Guest \mathrel{\rightarrowtail\!\!\!\!\!+} Room$

$\#RM = res$

$\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM$

Reserve

$\Delta(res, ST, RM);\ g? : Guest;\ r! : Room$

$res < total;\ g? \notin \mathrm{dom}\, RM$

$ST(r!) = Available$

$res' = res + 1$

$ST' = ST \oplus \{r! \mapsto Taken\}$

$RM' = RM \cup \{g? \mapsto r!\}$

Cancel

$\Delta(res, RM, ST)$

$g? : Guest$

$g? \in \mathrm{dom}\, RM$

$res' = res - 1$

$RM' = \{g?\} \mathbin{\lhd\!\!\!\!-} RM$

$ST' = ST \oplus \{RM(g?) \mapsto Available\}$

Report

$currentST! : Room \mathrel{+\!\!\!\!\rightarrow} Status$

$currentRM! : Guest \mathrel{\rightarrowtail\!\!\!\!\!+} Room$

$currentST! = ST$

$currentRM! = RM$

Figure 6.1 Initial Specification of the Hotel Database

The class *Hotel* models the hotel database. The database objects may be constants or variables. The hotel database has a constant *total* which is the number of rooms in the hotel. The hotel database has three variables, namely, *res*, *RM* and *ST* which are declared in the state schema. The natural number *res* counts current reservations, and the partial injection *RM* relates guests to rooms. Our particular example does not allow guests to register multiple times, which is reflected in the fact that *RM* is an injective function. The example could be modified easily with different constraints. The partial function *ST* records the status of each room. Additional integrity constraints on the objects in hotel database appear in the bottom part of the state schema. There are two such constraints:

1. $\#RM = res$. The number of guests who have been assigned rooms (the size of the *RM* function) equals the total number of reservations (*res*).

2. $\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM$. The set of rooms that are taken ($\mathrm{dom}(ST \rhd \{Taken\})$) is exactly the set of rooms reserved by guests ($\mathrm{ran}\, RM$).

The three operation schemas *Reserve*, *Cancel* and *Report* describe the three transactions in the hotel database. *Reserve* reserves a room for guest $g?$ and produces as output a room assignment $r!$. *Reserve* has a precondition that there must be fewer than *total* reserved rooms and $g?$ must be a new guest. Since the domain of *RM* is the set of guests with reservations, the latter is captured by checking that $g? \notin \mathrm{dom}\, RM$. *Reserve* has a postcondition that some room $r!$ with status *Available* is chosen, the number of reservations is incremented, the status of $r!$ is changed to *Taken*, and the ordered pair $g? \mapsto r!$ is added to the function *RM*. *Cancel* cancels a reservation for guest $g?$. *Cancel* has a precondition that $g?$ is a guest and a postcondition that *res* is decremented, $g?$ is removed from the domain of the function *RM*, and the status of the room for $g?$ is changed to *Available*. *Report* has no precondition, and merely produces the state components *ST* and *RM* as outputs.

Since the role of initialization is peripheral to our analysis, we omit initialization schemas here. Instead, we assume that the database has been initialized to a consistent state. As no history invariants are needed to restrict the execution of operations, we do not specify any history invariant.

6.4 THE MODEL

In our model, a *database* is specified as a collection of objects, along with some *invariants* or *integrity constraints* on these objects. At any given time, the *state* is determined by the values of the objects in the database. A change in the value of a database object changes the state. The invariants are predicates defined over the objects. A database state is said to be *consistent* if the values of the objects satisfy the given invariants.

A *transaction* is an operation that transforms one database state to another. Associated with each transaction is a set of *preconditions* and a set of *postconditions*. A precondition limits the database states to which a transaction can be applied. A postcondition constrains the possible database states after a transaction completes. For example, a *Reserve* transaction has a precondition that the hotel have at least one room available and a postcondition that there be some room available before the reservation that is taken after the reservation. Postconditions also constrain outputs. For example, the room $r!$ output by *Reserve* must be available initially. Together, preconditions and postconditions must ensure that if a transaction executes on a consistent state, the resulting state is also consistent.

Instead of executing a transaction as an atomic unit, we wish to break a transaction into steps and execute each of these steps as an atomic unit. A *decomposition* of a transaction is a sequence of *steps*. In place of the transaction, the steps execute atomically in order. A transaction that has been decomposed into two or more steps is referred to as a *multistep* transaction.

One possible approach to decomposition is to treat the steps as transactions. In particular, one could insist that the integrity constraints hold after each step, which is the decision taken in the Saga model [Garcia-Molina and Salem, 1987]. As the naive decomposition below demonstrates, such a requirement is too strong for some applications, and so we develop a more flexible approach.

6.4.1 A Naive Decomposition of the Reserve Transaction

Suppose we break up the *Reserve* transaction into the following three atomic steps.

Step 1: Increment the number of reserved rooms.

Step 2: Pick a room with status *Available* and change it to *Taken*.

Step 3: Assign the room selected in Step 2 to the guest.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NaiveR1}} \\
\hline
\Delta(\mathit{res}) \\
\hline
\mathit{res} < \mathit{total} \\
\mathit{res}' = \mathit{res} + 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NaiveR2}} \\
\hline
\Delta(\mathit{ST}) \\
r! : \mathit{Room} \\
\hline
\mathit{ST}(r!) = \mathit{Available} \\
\mathit{ST}' = \mathit{ST} \oplus \\
\quad \{r! \mapsto \mathit{Taken}\} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NaiveR3}} \\
\hline
\Delta(\mathit{RM}) \\
r! : \mathit{Room} \\
g? : \mathit{Guest} \\
\hline
g? \notin \operatorname{dom} \mathit{RM} \\
\mathit{RM}' = \mathit{RM} \cup \{g? \mapsto r!\} \\
\hline
\end{array}
$$

Figure 6.2 A Naive Decomposition of *Reserve*

A naive specification of these steps is given in figure 6.2. From a formal perspective, the naive decomposition has a fatal flaw, in that none of the proposed steps, considered by itself, maintains the invariants in *Hotel*. For example, *NaiveR1* does not maintain the invariant $\#\mathit{RM} = \mathit{res}$ since *NaiveR1* increments the value of *res*, but does not alter *RM*. Formally, the computed preconditions of all three steps simplify to false, indicating that none of the steps can be safely executed in an implementation. Executing any of the proposed steps violates the invariants, and other transactions are exposed to the inconsistent state. For example, *Report* produces an inconsistent output if executed in a state in which the second invariant does not hold.

6.4.2 Generalizing the Original Invariants

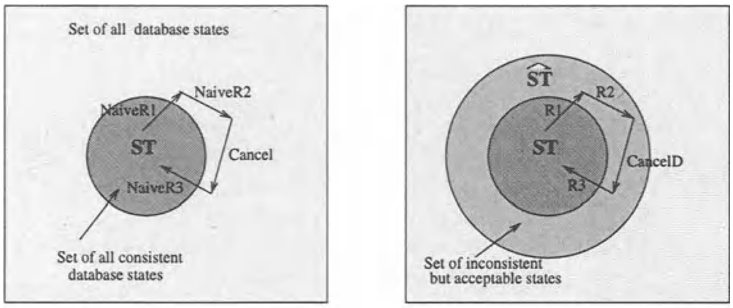
The example demonstrates that some decompositions are unacceptable. Specifically, a decomposition may yield steps that leave the database in a state in which the invariants do not hold. The arrow labeled *NaiveR1* in figure 6.3(a) illustrates this possibility. Once the invariants are violated, the formal basis for assessing the correctness of subsequent behavior collapses.

Insisting on decompositions where each step maintains database consistency does solve this problem. However, the informal description of the steps into which *Reserve* is broken is perfectly satisfactory, and it is excessive to insist that the invariants of *Hotel* hold at all intermediate steps. Later in figure 6.4 we show the correct formal specification of the three steps of the *Reserve* transaction which we denote by $R1$, $R2$ and $R3$; *CancelD* and *ReportD* denote the single steps of *Cancel* and *Reserve* transaction respectively. But before showing the specifications we present a formal model that can accommodate the notion that some – but not all – violations of the invariants are acceptable.

Figure 6.3(b) illustrates a model that allows inconsistent states – as defined by the invariants – that are nonetheless acceptable. The temporary inconsistency introduced by $R1$ is allowed, and steps of some other transactions, for example *CancelD*, can tolerate the inconsistency introduced by $R1$, and so are allowed to proceed. The chosen approach is to generalize the original set of invariants and decompose transactions such that each step satisfies the new set of invariants. The model in figure 6.3(b) has many advantages, including greater concurrency among steps. We formalize the model as follows.

Let I denote the original invariants, and let **ST** denote the set consisting of all consistent states; that is, $\mathbf{ST} = \{ST : ST \text{ satisfies } I\}$. In the standard model, a transaction T_i always accesses a consistent $ST \in \mathbf{ST}$. If ST_i denotes the state after the execution of T_i, then ST_i is also in **ST**. When T_i is broken up into steps $S_{i1}, \ldots, S_{in}$, each step S_{ij} executes atomically. If ST_{ij} represents the state resulting from the partial execution of T_i through step S_{ij}, it is possible that ST_{ij} no longer satisfies the invariants I and so lies outside **ST**. Figure 6.3(a) illustrates this possibility for the naive decomposition of the hotel example.

In our approach, we define a new set of invariants $\hat{I}$ by relaxing the original invariants I. We decompose each transaction such that execution of any step results in a state that satisfies $\hat{I}$. Let $\widehat{\mathbf{ST}} = \{\mathrm{ST} : \mathrm{ST} \text{ satisfies } \hat{I}\}$. The relationship between **ST** and $\widehat{\mathbf{ST}}$ is shown in figure 6.3(b). The inner circle denotes **ST** and the outer circle denotes $\widehat{\mathbf{ST}}$ (signifying that $\mathbf{ST} \subset \widehat{\mathbf{ST}}$). The ring denotes the set of all states that satisfy $\hat{I}$ but not I. The important part about figure 6.3(b) is that the set of inconsistent but acceptable states is formally identified and distinguished from the states that are unacceptable. The advantage is that formal analysis can be used to investigate activities in $\widehat{\mathbf{ST}}$.


Set of all database states
NaiveR1
NaiveR2
ST
Cancel
NaiveR3
Set of all consistent database states
$\widehat{\mathrm{ST}}$
R1
R2
ST
CancelD
R3
Set of inconsistent but acceptable states

(a) Standard classification of database states

(b) Database states as classified in our model

Figure 6.3 Classification of the Database States

To reason about decomposing transactions into steps and to avoid the problems of a naive decomposition, we use *auxiliary variables* to generalize the invariants. Auxiliary variables are a standard method of reasoning about concurrent executions [Owicki and Gries, 1976] and, in particular, have been applied to the problem of semantic-based concurrency control [Garcia-Molina, 1983, Appendix C]. Our work focuses more on the decomposition than does [Garcia-Molina, 1983], and so we emphasize the rôle of auxiliary variables more strongly. We stress that the auxiliary variables are introduced for purposes of analysis; the goal is to eliminate such variables from an implementation.

6.4.3 Compensating Steps

When transactions are decomposed into steps, it may not complete successfully if a precondition of a step is not satisfied, or if the user aborts the transaction, or if the system crashes. Incomplete transactions pose special problems in semantic oriented models because steps may commit before it is determined whether the transaction can complete. For example, suppose a *Reserve* transaction aborts after its first step $R1$ commits. Some mechanism must undo the effects of $R1$. Nullifying the effects of $R1$ using the backwards recovery method of traditional *undo* [Bernstein et al., 1987], where the state that existed before $R1$ is physically restored, is not possible because steps of other transactions may have read the updates of $R1$. Instead we adopt the forward recovery solution of a compensating step [Garcia-Molina, 1983, Garcia-Molina and Salem, 1987]; such a step semantically undoes the effects of the committed step $R1$ but does not disturb transactions that may have read from $R1$.

Like other steps of a transaction, compensating steps execute atomically. However, the role of a compensating step differs from that of other steps. A compensating step is not considered part of the normal processing of a transaction; it is initiated only to semantically undo a transaction.

For a transaction T_i that has been decomposed into n steps $S_{i1}, \ldots, S_{in}$ we specify $n-1$ compensating steps denoted by $C_{i2}, \ldots, C_{in}$. The compensating step C_{ij} semantically undoes the cumulative effect of steps $S_{i1}, \ldots, S_{i(j-1)}$. This is in contrast to the approach used in [Garcia-Molina, 1983, Garcia-Molina and Salem, 1987] where a compensating step C_{ij} is used to semantically undo the operations performed by a single step S_{ij}. The difference between the approaches is not significant; our choice simplifies the presentation.

6.4.4 Semantic Histories

We are interested in the relationship between the original specification and the specification with the generalized invariants. In particular, we would like to know if and when the database returns to a consistent state.

Before we proceed further, we make a distinction between a type of a step and an instance of a step. The three steps $R1$, $R2$ and $R3$ of the *Reserve* transaction are examples of different types of steps in the hotel example.

Histories, defined subsequently, reflect actual transactions, and must reference instances of steps and compensating steps. A history may contain many instances of a step of a given type. In cases where it not necessary to distinguish the role of steps from that of compensating steps, we use the term 'step' generically and denote an instance of either a step or a compensating step of transaction T_i as T_{ij}. Where the roles differ, we use S_{ij} to denote an instance of a step and C_{ij} to denote an instance of a compensating step. The type of an instance of a step T_{ij} is denoted by $ty(T_{ij})$.

Definition 6.1 [Stepwise Serial History] *A stepwise serial history H over a set of transactions $\mathbf{T} = \{T_1, \ldots, T_m\}$ is a sequence of steps and compensating steps such that*

1. *a step T_{ij} either appears exactly once in H or does not appear at all,*
2. *for any two steps S_{ij} and T_{ik}, S_{ij} precedes T_{ik} in H if S_{ij} precedes T_{ik} in T_i,*
3. *if $T_{ik} \in H$, then $S_{ij} \in H$ for $j = 1, \ldots, (k-1)$,*
4. *if a compensating step $C_{ij} \in H$, then $S_{ij} \notin H$ and $T_{ik} \notin H$ for $k > j$.*

Condition (1) ensures that every step of a transaction occurs at most once. Condition (2) ensures that the order of the steps in a transaction is preserved.

Condition (3) ensures that for every step, preceding steps in the corresponding transaction are present. Condition (4) ensures that a compensating step terminates a transaction.

Unlike typical definitions of histories, our notion of a history does not reference the operations on data elements, such as read and write. Such operations are introduced and integrated into the definition of histories as we further refine our specifications.

Example 6.1 *$< S_{11}, S_{11} >$ is not a stepwise serial history since it violates condition 1. $< S_{13}, S_{12} >$ is not a stepwise serial history since it violates conditions 2 and 3. $< S_{11}, C_{12}, S_{12} >$ is not a stepwise serial history since it violates condition 4. $< S_{11}, S_{21}, S_{12}, S_{13} >$ and $< S_{11}, S_{21}, C_{12}, S_{22} >$ are stepwise serial histories.*

Definition 6.2 [Complete Execution] *Consider a transaction T_i decomposed into steps $S_{i1}, \ldots, S_{in}$ with compensating steps C_{i2}, ..., C_{in}. The execution of T_i in a history H is a complete execution if either (i) all n steps of T_i appear in H or (ii) some steps of T_i, namely, S_{i1}, ..., S_{ij} appear in H followed by the corresponding compensating step $C_{i(j+1)}$, where $j < n$.*

The sequences $S_{i1}, \ldots, S_{in}$ and $S_{i1}, \ldots, S_{ij}, C_{i(j+1)}$ are examples of *execution sequences* [Garcia-Molina, 1983] of transaction T_i. The sequence $S_{i1}, \ldots, S_{in}$ is a *successful* execution sequence of T_i, and the sequence $S_{i1}, \ldots, S_{ij}, C_{i(j+1)}$ is an *unsuccessful* execution sequence of T_i.

Example 6.2 *For the hotel database, an execution of a Reserve transaction T_i is complete in H if either (i) all three steps S_{i1}, S_{i2}, and S_{i3} of T_i appear in H, or (ii) S_{i1} and C_{i2} appear in H, or (iii) S_{i1}, S_{i2} and C_{i3} appear in H. Case(i) is an example of successful complete execution. Cases(ii) and (iii) are examples of unsuccessful complete executions.*

To introduce state information, we define *semantic history.*

Definition 6.3 [Semantic History] *A semantic history H is a stepwise serial history bound to*

1. *an initial state, and*

2. *the states resulting from the execution of each step in H.*

Informally, we use the term partial semantic history for cases in which the execution of at least one transaction actually is incomplete, but from a formal perspective, partial semantic histories are just semantic histories. Complete semantic histories are a special case of a semantic histories:

Definition 6.4 [Complete Semantic History] *A semantic history H over a set of transactions* **T** *is a complete semantic history if the execution of each T_i in* **T** *is complete.*

Next we define what it means for a semantic history to be correct.

Definition 6.5 [Correct Semantic History] *A semantic history H is a correct semantic history if*

1. *the initial state is in* **ST**,
2. *the states before and after each step in H are in* $\widehat{\mathbf{ST}}$, *and*
3. *the precondition for each step is satisfied in the corresponding state.*

Definition 6.6 [Correct Complete Semantic History] *A complete semantic history H is a correct complete semantic history if*

1. *H is a correct semantic history, and*
2. *the final state is in* **ST**.

6.5 PROPERTIES OF VALID DECOMPOSITION

To ensure the correct behavior of an application in which transactions have been decomposed into steps, we propose a set of necessary and desirable properties.

6.5.1 Composition Property

When transactions have been decomposed into steps, we can state a property relating steps in a decomposition to the original transaction. We call this requirement the *composition property*.

Composition Property Let $S_{i1}, \ldots, S_{in}$ be the steps of transaction T_i and ST be a state that satisfies the original integrity constraints I. Then executing the sequence of steps $S_{i1}, \ldots, S_{in}$ in isolation on ST is equivalent to executing T_i on ST, except for constraints on auxiliary variables.

The composition property does not address what happens if the precondition of some step is not satisfied and thus the execution cannot complete. From an implementation perspective, the composition property is similar to requiring that the sequential execution of the steps be view equivalent to that of the original transaction.

6.5.2 Sensitive Transaction Isolation Property

In our model, we allow transactions to access database states that do not satisfy the original invariants (that is, states in $\widehat{\mathbf{ST}}$ – **ST**). But we may wish to keep some transactions from viewing any inconsistency with respect to the original invariants. For example, some transactions may output data to users; these transactions are called *sensitive* transactions [Garcia-Molina, 1983]. We require sensitive transactions to appear to have generated outputs from a consistent state. This leads us to the next property.

Sensitive Transaction Isolation Property All output data produced by a sensitive transaction T_i should have the appearance that it is based on a consistent state in **ST**, even though the decomposition of T_i may access database states in $\widehat{\mathbf{ST}} - \mathbf{ST}$.

In our model, we ensure the sensitive transaction isolation property by construction. There are two aspects to such a construction. First, for each sensitive transaction, we compute the subset of the original integrity constraints, I, relevant to the calculation of any outputs. Second, as pointed out by Rastogi, Korth, and Silberschatz [Rastogi et al., 1995], if outputs are generated by multiple steps, interleavings between these steps must be controlled to ensure that outputs from later steps are consistent with outputs from earlier steps.

6.5.3 Consistent Execution Property

Similar to the consistency property for traditional databases, we place the following requirement on semantic histories:

Consistent Execution Property If we execute a correct complete semantic history H on an initial state (i.e., the state prior to the execution of any step in H) that satisfies the original invariants I, then the final state (i.e., the state after the execution of the last step in H) also satisfies the original invariants I.

6.5.4 Semantic Atomicity Property

When transactions have been broken up into steps, it may not be always possible to complete a transaction. This happens if the precondition of some later step is not satisfied and the effects of the partially executed transactions cannot be undone by executing compensating steps. The semantic atomicity property ensures that such a situation is avoided; if a transaction has been partially executed, then it can complete.

Semantic Atomicity Property Every correct semantic history H_p defined over a set of transactions **T** is a prefix of some correct complete semantic history H over **T**.

Like all the other properties stated so far, semantic atomicity is a necessary property. The definition of semantic atomicity property is very general. Some applications may require a stronger property, the successful execution property, stated below.

6.5.5 Successful Execution Property

The interleaving of steps of different transactions may result in a state from which it is not possible to successfully complete some transaction. The successful execution property ensures that such a situation is avoided; if a transaction has been partially executed, then it can complete without resorting to compensation.

Successful Execution Property Every correct semantic history H_p defined over a set of transactions $\mathbf{T}$ is a prefix of some correct complete semantic history H over $\mathbf{T}$ such that for each $T_i \in \mathbf{T}$ that is incomplete in H_p, H contains a successful execution sequence of T_i.

Unlike the other properties we have stated so far, successful execution is an optional property. Successful execution property requires that all the preconditions of a transaction should appear in the first step. This in turn would require a large number of updates to be performed in the first step. (Precondition checks are often associated with updates; in such cases we require to perform the check and update atomically, that is, in the same step.) Thus insisting on successful execution property may force too many operations in the first step of the transaction - which is undesirable from the performance point of view. Hence we do not insist that applications have the successful execution property.

6.6 EXAMPLES OF DECOMPOSITION

6.6.1 A Valid Decomposition

We now provide a valid decomposition of the hotel database which satisfies all the necessary properties described in the previous section. The class *HotelD* (short for *Hotel Decomposition*) in figure 6.4 specifies this valid decomposition.

We generalize the invariants by adding the auxiliary variables *underway* and *acquired*. *underway* is a natural number which denotes the reservations that have been partially processed. The auxiliary variable *acquired* denotes the set of rooms that have been taken but which have not yet been assigned to guests. The declarations of these auxiliary variables appear in the top part of the state schema. The two generalized invariants appear at the bottom of the state schema. They are:

1. $\#RM + underway = res$

2. $\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM \cup acquired$

Since the steps and compensating steps now comprise the operations of the decomposed hotel database, we specify operation schemas corresponding to each type of step and compensating step. Only the *Reserve* transaction is decomposed into three steps, namely $R1$, $R2$ and $R3$. $R1$ has a precondition that

that there must be fewer than *total* rooms. The postcondition of $R1$ increments *res* and *underway*. The precondition of $R2$ is that the room to be assigned to the guest $r!$ is *Available*. The postcondition of $R2$ changes the status of $r!$ to *Taken* and inserts $r!$ in the set *acquired*. The precondition of $R3$ is that $g?$ must be a new guest. The postcondition of $R3$ inserts the ordered pair $g? \mapsto r!$ to the function RM, removes $r!$ from the set *acquired* and decrements *underway*. The compensating step *CompR*2 semantically undoes the effect of $R1$. The preconditions of *CompR*2 are that the variables *res* and *underway* must be positive. The postcondition of *CompR*2 decrements *res* and *underway*. The compensating step *CompR*3 semantically undoes the cumulative effects of $R1$ and $R2$. The preconditions of *CompR*3 are that *res* and *underway* must be positive and $r!$ must be in the set *acquired*. The postconditions of *CompR*3 decrement *res* and *underway* and remove $r!$ from the set *acquired* and change the status of $r!$ to *Available*. *CancelD* and *ReportD* represent the single steps of the *Cancel* and *Report* transaction respectively; the specifications of these steps are identical to the corresponding transactions.

6.6.1.1 Composition Property. To implement a *Reserve*, its three steps must execute in order. The composition property for the hotel example, formally stated in Object Z, is as follows.

$Hotel \wedge ((R1 \mathbin{⨾} R2 \mathbin{⨾} R3) \setminus (underway, acquired, underway', acquired')) \Leftrightarrow Reserve$

The left hand side gives the composition of the steps where the initial state is constrained to satisfy the original invariants and the auxiliary variables are hidden or suppressed. The right hand side is the original transaction *Reserve*. In Object Z a propositional relation between schemas – equivalence in this case – translates into the same relation between the predicates defining the schemas. The three steps satisfy the composition property; we omit the details of the proof in this chapter.

6.6.1.2 Sensitive Transaction Isolation Property. *Report* is a sensitive transaction, and we establish the sensitive transaction isolation property by construction. A formal treatment is given in [Ammann et al., 1997]. Informally, *Report* transaction outputs values of ST and RM. ST and RM appear in the following original invariant:

$\operatorname{dom}(ST \rhd \{Taken\}) = \operatorname{ran} RM$

which can be derived from the generalized invariant

$\operatorname{dom}(ST \rhd \{Taken\}) = \operatorname{ran} RM \cup acquired$

if the auxiliary variable *acquired* satisfies $acquired = \varnothing$. Hence, to ensure that *ReportD* does not output inconsistent data we specify the following restriction as a history invariant.

$\Box((acquired \neq \varnothing) \Rightarrow (\bigcirc \mathbf{op} \neq ReportD))$.

The above notation means that it is always true when the auxiliary variable *acquired* is not the empty set, the next operation must not be the step *ReportD*.

Although *Reserve* is a sensitive transaction, it turns out that no additional preconditions are needed to ensure that the output $r!$ reflects a consistent state.

6.6.1.3 Consistent Execution Property. Consider any correct complete history H generated from the decomposition specified in figure 6.4. To prove the consistent execution property we must show that if H is executed in a consistent state, the final state is also consistent.

When the database is in a consistent state, the auxiliary variables satisfy the following condition: $underway = 0 \wedge acquired = \emptyset$.

Let $r1$, $r2$, $r3$, $compr2$, $compr3$ be the number of steps of type $R1$, $R2$, $R3$, $CompR2$, $CompR3$ respectively in H. The auxiliary variable *underway* is incremented by steps of type $R1$ and decremented by steps of type $R3$, $CompR2$ and $CompR3$. Since the initial state of H is consistent, the value of *underway* in the final state of H is given by the following expression

$$underway = r1 - (r3 + compr2 + compr3) \quad \cdots \quad (6.1)$$

Similarly we have,

$$|\, acquired \,| = r2 - (r3 + compr3) \quad \cdots \quad (6.2)$$

Since H is complete, each step of type $R1$ has a corresponding step of type $R2$ or $CompR2$. Similarly, each step of type $R2$ has a corresponding step of type $R3$ or $CompR3$. Thus we have

$$r1 = r2 + compr2 \quad \cdots \quad (6.3)$$

$$r2 = r3 + compr3 \quad \cdots \quad (6.4)$$

From (6.1 –6.4) we can derive that in the final state of H, $underway = 0 \wedge acquired = \emptyset$ which means that the final state is consistent.

6.6.1.4 Semantic Atomicity Property. Let H_p be any correct partial semantic history. H_p has one or more incomplete *Reserve* transactions. Consider an incomplete *Reserve* transaction. If this transaction has committed only step $R1$, then it can complete by executing $CompR2$. This is possible because steps of no other transaction executing after $R1$ can violate the preconditions of $CompR2$. Similarly it can be shown that if the *Reserve* transaction has committed steps $R1$ and $R2$, it is possible to execute $CompR3$ and complete the *Reserve* transaction. In this way all the incomplete *Reserve* transactions can be completed and the partial history H_p extended to a correct complete semantic history H_c. H_p is therefore a prefix of H_c.

HotelD

$res, underway : \mathbb{N}$

$ST : Room \mathrel{⇸} Status$

$RM : Guest \mathrel{⤔} Room$

$acquired : \mathbb{P}\, Room$

$\#RM + underway = res$

$\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM \cup acquired$

R1

$\Delta(res, underway)$

$res < total$

$res' = res + 1$

$underway' = underway + 1$

R2

$\Delta(ST, acquired);\ r! : Room$

$ST(r!) = Available$

$ST' = ST \oplus \{r! \mapsto Taken\}$

$acquired' = acquired \cup \{r!\}$

R3

$\Delta(RM, acquired, underway)$

$g? : Guest;\ r! : Room$

$g? \notin \mathrm{dom}\, RM$

$RM' = RM \cup \{g? \mapsto r!\}$

$acquired' = acquired \setminus \{r!\}$

$underway' = underway - 1$

CancelD

$\Delta(ST, RM, res)$

$g? : Guest$

$g? \in \mathrm{dom}\, RM$

$res' = res - 1$

$ST' = ST \oplus \{RM(g?) \mapsto Available\}$

$RM' = \{g?\} \mathbin{⩤} RM$

ReportD

$currentST! : Room \mathrel{⇸} Status$

$currentRM! : Guest \mathrel{⤔} Room$

$currentST! = ST$

$currentRM! = RM$

CompR2

$\Delta(res, underway)$

$res > 0;\ underway > 0$

$res' = res - 1$

$underway' = underway - 1$

CompR3

$\Delta(res, underway, acquired, ST)$

$r! : Room$

$res > 0$

$underway > 0$

$r! \in acquired$

$res' = res - 1$

$underway' = underway - 1$

$acquired' = acquired \setminus \{r!\}$

$ST' = ST \oplus \{r! \mapsto Available\}$

$\Box(acquired \neq \varnothing) \Rightarrow \bigcirc(\mathbf{op} \neq ReportD)$

Figure 6.4 A Valid Decomposition for the Hotel Database

6.6.2 An Invalid Decomposition

In this section, we give an example of an invalid decomposition. Unlike the naive decomposition, the decomposition given below generates correct semantic histories. The decomposition is invalid because it does not satisfy one of the necessary properties, namely, the semantic atomicity property.

To illustrate the possibility, we modify the *HotelD* specification. The modified specification, known as *DeadlockHotel*, is shown in fig. 6.5.

In the example specification, the cancel transaction is decomposed into steps C1 and C2. We introduce the auxiliary variable *underwayC* which keeps count of the cancel transactions that have completed step $C1$ but not step $C2$. The invariant $\#RM = res - underway$ in the *HotelD* is changed to $\#RM = res - underway + underwayC$ in *DeadlockHotel*.

Also, we introduce a new structure *clist* which keeps track of the guests whose cancelations are in progress. The guest whose reservation is being canceled is added to the *clist* in step $C1$ and is removed from the *clist* in step $C2$. To ensure that a guest whose cancelation is in progress is not canceled by some other transaction, we include precondition $g? \notin clist$ in step $C1$. $C1$ has another precondition $res > 0$ which ensures that $C1$ executes when there is at least one reservation. The postcondition of $C1$ decrements *res*, increments *underwayC* and inserts $g?$ in *clist*. The preconditions of $C2$ check that $g?$ has a valid reservation, $g?$ is in *clist* and *underwayC* is positive. The postcondition of $C2$ removes $g?$ from the domain of RM, makes the room which was assigned to $g?$, *Available*, removes $g?$ from *clist*, and decrements *underwayC*. Since the cancel transaction is decomposed into two steps, we must specify *CompC*2, a type of compensating step, which semantically undoes the actions of $C1$. *CompC*2 has four preconditions: *res* must be less than *total*, *underwayC* must be positive, $g?$ must be in *clist* and $g?$ must have a valid reservation. The postcondition of *CompC*2 increments *res*, decrements *underwayC* and removes $g?$ from *clist*.

The reserve transaction is broken into steps *Res*1, *Res*2 and *Res*3, similar to $R1$, $R2$ and $R3$ of the *HotelD* specification. We impose an additional constraint that a room cannot be reserved for a guest whose cancelation is in progress; the precondition $g? \notin clist$ in step *Res*3 ensures this. We assume that this example has no *Report* transaction.

Consider the partial history $H_p = < S_{11} >$ where $ty(S_{11}) = C1$. Suppose in the initial state of H_p, $John \notin \mathrm{dom} RM$. The cancel transaction T_1 attempts to cancel the reservation for guest *John*. The execution of step S_{11} results in *John* being inserted in the *clist*. Now step S_{12} cannot execute since the precondition $John \notin \mathrm{dom} RM$ is not satisfied as *John* does not have any reservation. The compensating step C_{12} of type *CompC*2 also checks for these preconditions; since these preconditions are not satisfied, the compensating step cannot be executed. The specification is therefore an invalid specification – it lacks the semantic atomicity property.

The deadlock could be avoided by including the invariant $clist \subseteq guest$ in *DeadlockHotel*. Omission of this constraint allows the database to enter an undesirable state where $c? \in clist \wedge c? \notin guest$, from which neither the next step or the compensating step could be executed.

6.7 SUCCESSOR SETS

After presenting examples of decomposition, we now describe mechanisms to efficiently implement our model. The decomposition process introduces additional database objects (auxiliary variables) and imposes additional constraints (history invariants) on the execution of steps. The additional objects are present primarily to support analysis. For efficient implementation, we want to avoid instantiating the objects. Checking the satisfaction of history invariants before scheduling an operation is expensive and our goal is to avoid such checks in the implementation. Successor sets are the mechanism we use to achieve these objectives.

Definition 6.7 [Successor Set] *The successor set of $ty(T_{ij})$, denoted $SS(ty(T_{ij}))$, is a set of types of steps.*

At this point, the notion of successor sets is purely syntactic. Subsequently, we define the constraints under which a successor set description is correct with respect to a particular decomposition. But first we wish to define the notion of correct successor set histories.

To achieve this goal we introduce the notion of conflict into our model. Two operations *conflict* if both operate on the same data item and at least one is a Write. Two steps T_{ij} and T_{pq} *conflict* if they contain conflicting operations. It is easy to determine the set of conflicting steps once the code for the decomposed transactions is given. At this stage we only have the specification, but we would still like to define a notion of conflict. We define any state variable modified in a postcondition of an operation as being *written* in the specification. Similarly, we define any state variable referenced in a precondition or postcondition as being *read* in the specification.

The read and write set of the steps of the decomposed hotel database, as obtained from the specifications (figure 6.4) is given in Table 6.2. Table 6.3 gives the set of conflicting steps in the Hotel Database.

The definition of conflict allows us to define a notion of correctness with respect to successor set descriptions that is not overly restrictive.

Definition 6.8 [Correct Successor Set History] *H is a correct successor set history if it satisfies the following conditions.*

1. *H is a correct semantic history.*

DeadlockHotel

$$res, underway, underwayC : \mathbb{N};\ clist : \mathbb{P}\,Guest;\ acquired : \mathbb{P}\,Room$$
$$RM : Guest \rightarrowtail\!\!\!\!\!\!\rightarrow Room;\ ST : Room \nrightarrow Status$$
$$\#RM = res - underway + underwayC;\ \mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\,RM \cup acquired$$

Res1

$$\Delta(res, underway)$$
$$res < total;\ res' = res + 1$$
$$underway' = underway + 1$$

Res2

$$\Delta(ST, acquired);\ r! : Room$$
$$ST(r!) = Available;\ ST' = ST \oplus \{r! \mapsto Taken\}$$
$$acquired' = acquired \cup \{r!\}$$

Res3

$$\Delta(RM, underway, acquired)$$
$$g? : Guest;\ r! : Room$$
$$underway > 0;\ r! \notin \mathrm{ran}\,RM$$
$$g? \notin \mathrm{dom}\,RM;\ g? \notin clist$$
$$RM' = RM \cup \{g? \mapsto r!\}$$
$$underway' = underway - 1$$
$$acquired' = acquired \setminus \{r!\}$$

C1

$$\Delta(res, clist, underwayC)$$
$$g? : Guest$$
$$res > 0$$
$$g? \notin clist$$
$$res' = res - 1$$
$$clist' = clist \cup \{g?\}$$
$$underwayC' = underwayC + 1$$

C2

$$\Delta(ST, RM, clist, underwayC)$$
$$g? : Guest$$
$$g? \in \mathrm{dom}\,RM;\ g? \in clist$$
$$underwayC > 0$$
$$ST' = ST \oplus \{RM(g?) \mapsto Available\}$$
$$RM' = \{g?\} \lhd RM;\ clist' = clist \setminus \{g?\}$$
$$underwayC' = underwayC - 1$$

CompRes2

$$\Delta(res, underway)$$
$$res > 0$$
$$underway > 0$$
$$res' = res - 1$$
$$underway' = underway - 1$$

CompRes3

$$\Delta(res, underway, acquired, ST)$$
$$r! : Room$$
$$res > 0;\ underway > 0$$
$$r! \in acquired;\ ST(r!) = Taken$$
$$res' = res - 1$$
$$underway' = underway - 1$$
$$acquired' = acquired \setminus \{r!\}$$
$$ST' = ST \oplus \{r! \mapsto Available\}$$

CompC2

$$\Delta(res, clist, underwayC)$$
$$g? : Guest$$
$$res < total$$
$$g? \in clist;\ g? \in \mathrm{dom}\,RM$$
$$underwayC > 0$$
$$res' = res + 1$$
$$clist' = clist \setminus \{g?\}$$
$$underwayC' = underwayC - 1$$

Figure 6.5 Example Specification Lacking Semantic Atomicity Property

Table 6.2 Read and Write Sets for Steps of Hotel Example

Type of Step	Variables Read	Variables Written
*R*1	*res, total, underway*	*res, underway*
*R*2	*ST, acquired*	*ST, acquired*
*R*3	*RM, underway, acquired*	*RM, underway, acquired*
ReportD	*ST, RM*	
CancelD	*res, ST, RM*	*res, ST, RM*
*CompR*2	*res, underway*	*res, underway*
*CompR*3	*res, ST, underway, acquired*	*res, ST, underway, acquired*

Table 6.3 Conflicting Steps for Hotel Example

Type of Step	Types of Conflicting Steps
*R*1	*R*1, *R*3, *CancelD*, *CompR*2, *CompR*3
*R*2	*R*2, *R*3, *ReportD*, *CancelD*, *CompR*3
*R*3	*R*1, *R*2, *R*3, *ReportD*, *CancelD*, *CompR*2, *CompR*3
ReportD	*R*2, *R*3, *CancelD*, *CompR*3
CancelD	*R*1, *R*2, *R*3, *ReportD*, *CancelD*, *CompR*2, *CompR*3
*CompR*2	*R*1, *R*3, *CancelD*, *CompR*2, *CompR*3
*CompR*3	*R*1, *R*2, *R*3, *CancelD*, *ReportD*, *CompR*2, *CompR*3

2. *If T_i is incomplete in the prefix of H that ends at T_{pq}, and T_{ij} is the last step in T_i such that (i) T_{ij} conflicts with T_{pq} and (ii) T_{ij} precedes T_{pq} in H then $ty(T_{pq}) \in SS(ty(T_{ij}))$.*

In the hotel example, there is one history invariant corresponding to the sensitive transaction isolation property. This history invariant forbids the execution of steps of type *ReportD* when the auxiliary variable $acquired \neq \emptyset$. This history invariant is satisfied as long as a step of type *ReportD* does not appear between steps of type *R*2 and *R*3 of reserve transaction. To ensure this we specify the successor sets as shown in Table 6.4. For the hotel example, the history invariant involving auxiliary variable is captured by the successor set description, and so neither the history invariant nor the auxiliary variables need to be implemented.

With respect to the specifications given with history invariants, not all successor set descriptions are valid. Informally, a successor set is valid with respect to a specification containing history invariants if any correct successor set history can also be generated by the specification containing history invariants. Although desirable, the converse property does not hold in general since first-order logic history invariants have more expressive power than the successor

Table 6.4 Successor Sets for the Hotel Example

SS of Type of Step	Types of Steps in Successor Set
$SS(R1)$	$R1, R2, R3$, *ReportD*, *CancelD*, *CompR2*, *CompR3*
$SS(R2)$	$R1, R2, R3$, *CancelD*, *CompR2*, *CompR3*
$SS(R3)$	$R1, R2, R3$, *ReportD*, *CancelD*, *CompR2*, *CompR3*
SS(*ReportD*)	$R1, R2, R3$, *ReportD*, *CancelD*, *CompR2*, *CompR3*
SS(*CancelD*)	$R1, R2, R3$, *ReportD*, *CancelD*, *CompR2*,*CompR3*
SS(*CompR2*)	$R1, R2, R3$, *ReportD*, *CancelD*, *CompR2*,*CompR3*
SS(*CompR3*)	$R1, R2, R3$, *ReportD*, *CancelD*, *CompR2*,*CompR3*

set mechanism. Formally, we describe valid successor set descriptions with the *valid successor set property*:

Definition 6.9 [Valid Successor Set Property] *A specification S_2 that employs a successor set description is valid with respect to specification S_1 with history invariants if*

1. *any correct successor set history generated by S_2 is also a correct semantic history generated by S_1.*
2. *S_2 satisfies the semantic atomicity property.*

The second condition can be easily satisfied by ensuring that all compensating steps are contained in each successor set description. The hotel example has the valid successor set property, where it turns out that the successor set specification generates exactly the same set of histories as the specification with history invariants.

Suppose an application requires the successful execution property. Since successor set descriptions are less expressive than the first order predicates they replace, the set of histories for S_2 may be a proper subset of the set of histories for S_1. Therefore, the successful execution property must be reverified explicitly on histories generated by S_2.

6.8 CONCURRENT EXECUTION

6.8.1 Correct Stepwise Serializable Histories

For every pair of steps in a correct successor set history, all operations of one step appear before any operations of the other step. However if the steps of a transaction execute atomically and without any interleaving, the database system uses resources poorly. To improve efficiency we introduce the notion of *correct stepwise serializable history*. In a correct stepwise serializable history the steps of transactions need not be executed serially, but nevertheless the effect is the same as that of a correct successor set history.

We develop stepwise serializability by defining history and equivalence in a manner similar to [Bernstein et al., 1987]. A history H defined over a set of transactions **T** involves precisely the operations of steps in **T**, H preserves the order of operations in each step in **T** and any pair of conflicting operations are ordered in H. Two histories H and H' are said to be *equivalent* if they are defined over the same set of steps, they have the same operations, and they order conflicting operations of steps in the same way. That is, for any pair of conflicting operations p_{ij} and q_{kl} belonging to T_{ij} and T_{kl} (respectively), if $p_{ij} \prec_H q_{kl}$, then $p_{ij} \prec_{H'} q_{kl}$. A *correct stepwise serializable history* is one which is equivalent to a correct successor set history. A graph-theoretic characterization of correct stepwise serializable histories is given in [Ammann et al., 1997].

6.8.2 Concurrency Control Mechanism

We now propose a concurrency control mechanism for our model and identify the issues relevant to an implementation.

We make the following assumptions:

1. Lock management is centralized.
2. The steps of a transaction are submitted in order. That is an operation in step $T_{r(s+1)}$ is submitted only after step T_{rs} commits.
3. If a transaction reads and writes the same data entity x, the read operation precedes the write operation.
4. A transaction reads or writes an entity x at most once.
5. The algorithms specified below execute atomically.

Our mechanism uses two phase locking on the steps of the transactions. There are two modes in which a data item may be locked by a step - shared mode or exclusive mode. A step acquires an appropriate lock as a prerequisite for accessing a data item. A step is denied a lock if either another step holds a conflicting lock or if the step fails a test based on successor sets. Locks acquired by a step are released when the step commits or aborts.

For the purposes of this section, we define a step as a sequence of read and write operations followed by a commit or an abort operation,

$$T_{ij} = O_{ij}(x_1), O_{ij}(x_2), \ldots, O_{ij}(x_n), E_{ij},$$

where $O_{ij}(x)$ is either $R_{ij}(x)$ or $W_{ij}(x)$ and E_{ij} is either $C_{ij}(x)$ or $A_{ij}(x)$, and a transaction is a sequence of steps followed by a termination operation,

$$T_i = < T_{i1}, \ldots, T_{in}, TR(T_i) >$$

We require the following data structures in addition to those required by the two phase locking protocol.

1. Active-Set – Set of Active Transactions

   Active-Set(x) – The active set for x keeps the list of all active transactions whose committed steps have accessed x. Whenever any step T_{ij} that reads or writes x commits, the transaction T_i is added to Active-set(x). After the transaction T_i terminates, T_i is removed from Active-Set(x).

2. Int-Set – Interleaving Sets

   Int-Set(T_i,x) – The interleaving set for x is associated with each active transaction T_i that accesses x. The interleaving set gives the types of the steps that can access the data item. If data item x has been accessed by step T_{ij} of T_i and T_{ij} or any step of T_i occurring after T_{ij} commits, then Int-set(T_i,x) is replaced by the successor set of the corresponding committed step.

6.8.2.1 Algorithms. Before a read operation $R_{ij}(x)$ can proceed, step T_{ij} needs a shared lock for x. There are two conditions for T_{ij} to acquire the shared lock: (i) No other step has an exclusive lock on x and (ii) T_{ij} is in Int-Set(T_k,x) for all active transactions T_k whose committed steps have accessed x. If either condition is not satisfied, the lock is not granted and step T_{ij} must try again later. When $R_{ij}(x)$ is retried, it must be re-executed from the first step of the algorithm.

Algorithm for Read

```
Procedure Process-read ($R_{ij}$(x))
begin
  if a step $T_{lm}$ is holding an exclusive lock on x
    exit;          /* Lock unavailable - $T_{ij}$ can retry later */
  for each $T_k \in$ Active-set(x)
    if $ty(T_{ij}) \notin$ Int-set($T_k$,x)
      exit;     /* Lock unavailable - $T_{ij}$ can retry later */
  lock x in shared mode;
  accept($R_{ij}$(x));
end
```

Before a write operation $W_{ij}(x)$ can proceed, step T_{ij} needs an exclusive lock for x. There are two conditions for T_{ij} to acquire the exclusive lock: (i) No other step has any lock on x and (ii) T_{ij} is in Int-Set(T_k,x) for all active transactions T_k whose committed steps have accessed x. If either condition is not satisfied, the lock is not granted and step T_{ij} must try again later. When $W_{ij}(x)$ is retried, it must be re-executed from the first step of the algorithm.

Algorithm for Write

```
Procedure Process-write ($W_{ij}$(x))
begin
  if a step $T_{lm}$ is holding any lock on x
    exit;          /* Lock unavailable - $T_{ij}$ can retry later */
```

```
        for each $T_k \in$ Active-set(x)
            if $ty(T_{ij}) \notin$ Int-set($T_k$,x)
                exit;      /* Lock unavailable - $T_{ij}$ can retry later */
        lock x in exclusive mode
        accept($W_{ij}$(x));
    end
```

A step commits when all the operations of a step are complete. For each data item x locked by the transaction in the current or previous steps, the interleaving set associated with this transaction and data item x is replaced by the successor set of the step. The transaction is included in the list of active transactions that have accessed x. All locks acquired by this step are released.

Algorithm for Step Commit

```
    Procedure Process-stepcommit($C_{ij}$)
    begin
        for each x locked by the transaction in this or previous step do
            Int-set($T_i$,x) = $SS(ty(T_{ij}))$;
        for each entity x locked by the transaction in this step do
            begin
                if $T_i \notin$ Active-set(x)
                    Active-set(x) = Active-set(x) $\cup$ $T_i$;
                Release the lock on x which was acquired by $T_{ij}$;
            end
    end
```

A step may not always complete successfully and may abort. The abort causes all the locks held by the step to be released. The abort of step T_{ij} does not affect the data structures Active-set(x) or Int-set(T_i,x); these data structures are adjusted with the transaction termination processing. Traditional recovery for aborted transactions, such as undo, is required for the aborted step, but details are omitted.

Algorithm for Step Abort

```
    Procedure Process-stepabort($A_{ij}$)
    begin
        /* Restore values written by $T_{ij}$ */
        for each entity x locked by the transaction in this step do
            Release the lock on x which was acquired by $T_{ij}$;
    end
```

Termination removes a transaction from the set of active transactions. Since interleaving sets are associated only with active transactions, the interleaving set Int(T_i,x) is deleted when the transaction terminates.

Algorithm for Transaction Terminate

```
    Procedure Process-terminate($TR(T_i)$)
    begin
    for each entity x which was accessed by $T_i$ do
      begin
        Active-set(x) = Active-set(x) − $T_i$;
        delete the structure Int-set($T_i$,x) ;
      end
end
```

6.8.2.2 Discussion. As with other locking protocols, our mechanism has potential for starvation of transactions and deadlock. Since these issues can be addressed in standard ways, we do not describe detailed algorithms for solving these problems. However, these issues must be dealt with if an implementation of our model is to developed.

A variety of issues pertaining to supporting compensation must also be resolved. One issue is reliably storing data items which may be needed by a compensating step in case a multistep transaction does not complete. A second issue deals with initiating the compensating steps. Garcia-Molina suggests [Garcia-Molina, 1983] that the initiation of the compensating step must be done by the system. Such an approach has the advantage that all transaction aborts, whether user-initiated or failure-related, can be treated in a uniform way. A third issue is recovery from system crash. Transactions that are incomplete at the time of the crash can either be compensated or continued.

6.9 CONCLUSION

In this work, we have provided the database application developer writing the specification conceptual tools necessary to reason about systems in which transactions that ideally should be treated as atomic – for reasons of analysis – must instead be treated as a composition of steps – for reasons of performance. The developer begins with a specification produced via standard formal methods, transforms some transactions in the specification into steps, and assesses the properties of the resulting system. The formal analysis at each step of this process provides assurance that the resulting system possesses the desired properties.

Currently we are investigating how to apply semantic-based transaction decomposition to other areas like multidatabase applications and multilevel secure database systems. These areas impose some additional requirements which in turn pose new challenges to the decomposition process. We plan to investigate how typical applications in these areas can be processed using our model and study the relative advantages/disadvantages of our approach over the existing syntactic approach.

An important question is how well our model scales up to real-world applications. The necessary properties must be demonstrated for applications which must be implemented by our model. In this work, we have used the Object Z specification language and all the analysis are done by hand. However for real-world applications this may not be feasible and it may be necessary to automate to the maximum extent the discharge of proof obligations. Industrial-level tool support for such an endeavor is essential, and the use of existing automated theorem provers and model checkers needs to be investigated.

Acknowledgments

This work was partially supported by Defense Advanced Projects Research Agency under grant number N0060–96–D–3202. The work of S. Jajodia was also supported by National Science Foundation under grant number IRI–9633541 and by National Security Agency under grants MDA904-96-1-0103 and MDA904-96-1-0104. The work of I. Ray was also supported by a George Mason University Graduate Research Fellowship Award.

IV Concurrency Control and Recovery

7 CUSTOMIZABLE CONCURRENCY CONTROL FOR PERSISTENT JAVA

Laurent Daynès, M.P. Atkinson
and Patrick Valduriez

Abstract: We report on the issues raised when designing a customizable locking mechanism for Persistent Java, a type-safe, object-oriented, orthogonally persistent system based on the language Java. Customizable locking mechanisms are supported by locking capabilities. A locking capability is a bookkeeper of locks that automatically acquires locks with a customizable conflict detection mechanism. It implements the concepts of *delegation* of locks and *ignorable conflicts*, automatically keeps track of the dependencies created because of ignored conflicts, and supports the setting of user-defined notifications for conflicts that can't be ignored. Locking capabilities are one of the primitive components of a more general framework that gives the ability to expert application programmers to implement new transaction behaviors in Java. The framework doesn't change the Java language specification, and allows the use of any Java classes to implement the body of transactions without change to either their source or compiled form.

7.1 INTRODUCTION

Persistent programming languages offer an attractive alternative to the increasing number of applications whose needs cannot be satisfied with traditional database support. The requirement of these so called *non-traditional* applications have prompted the development of numerous transaction models whose semantics vary from the traditional transaction model as well as from each other [Elmagarmid, 1992, Barghouti and Kaiser, 1991]. The ever growing proliferation of transaction models, all unable to satisfy all needs, has definitively buried the hope of finding an universal model in the short term, if at all. In the absence of a proper transaction model, most persistent application builders end up investing a significant amount of time developing in-house transaction models to circumvent the proposed transaction support in order to better accommodate the needs of their application.

In order to minimize the cost of realizing new transaction models, application builders must be offered a simple framework which they can use to quickly define the transaction behavior they want and to incorporate it into the persistent programming system. Ideally, these extensions should not require the programmers to have an in-depth knowledge of how transaction processing mechanisms are implemented. Furthermore, each addition of a new transaction model should not require that the system be rebuilt. Instead, the system should be able to dynamically *adjust* itself to incorporate these extensions. Lastly, the user's extensions should be tightly integrated with the system in order to minimize the impact on the overall performance of the system.

This paper reports on our effort to augment Persistent Java (PJava), an alternative platform for the Java language [Atkinson et al., 1996], with such extensible transaction management features. The paper specifically focuses on the issues raised when designing the addition and the implementation of a customizable locking mechanism for Persistent Java.

7.1.1 Overview of Persistent Java

The main goal of the Persistent Java (PJava) project is to leverage Java to support faster development and better maintenance of persistent and transactional applications (e.g. [Jordan, 1996]) via provision of *orthogonal* properties. Providing properties such as persistence and transaction semantics orthogonally has two benefits.

1. Application programming is not polluted with considerations unrelated to the application logic itself, such as persistence or enforcement of some transactional properties. In particular, programmers do not have to explicitly identify the data that may become persistent or may be used in a transactional way. Similarly, the standard Java code that would operate on transient data is used *unchanged* when it operates on persistent data or in a transactional context. The addition of the desired property (e.g., persistence, persistence + transaction) is achieved by simply composing the application code with some context-aware code that encapsulates the particularities of the application requirement (e.g., management of roots of persistence or monitoring of transaction execution).

2. Any Java classes can be used to build applications in a specific operational context (non-persistent Java, persistent Java, persistent and transactional Java) without any change to either the sources or the compiled form of these classes; no extra rewriting/pre-processing or code gener-

ation steps are necessary to execute standard Java classes in PJava and obtain persistence or transaction semantics. Conversely, the source and compiled form of any classes programmed with PJava can be re-used in any standard Java development environment and executed by any standard virtual machine, except for a minority of classes that encapsulate the use of built-in classes specific to PJava (the "context-aware" classes).

The current PJava prototype realizes an alternative platform for the Java language with provision of completely orthogonal persistence for data, meta data (classes) and code (methods). Persistence is added to the Java language with no perturbation to Java's semantics. Consequently, all Java classes can be re-used in persistent applications without any alteration to either their source or their compiled form. The reader is referred to [Atkinson and Morrison, 1995] for an extensive definition of orthogonal persistence and to [Atkinson et al., 1996] for its application to the language Java. From the application programmer's point of view, persistence is simply obtained by composing normal Java classes with a few other persistence-aware classes (in most cases one) that interact with an object that implements the `PJStore` interface. The localized persistent-aware code typically identifies the roots of persistence, binds these root objects to the application's variables, and triggers the stabilization of all updates[1] onto the persistent store.

Our design to add extensible transaction management to Java follows a similar philosophy. Transactions are introduced into Java without changing the language definition and such that programmers don't have to explicitly identify the data manipulated within transactions. The aim is to allow the use of any pre-existing Java classes to program the body of transactions without any alteration to the original source and compiled form of these classes. These transaction bodies can then be executed in the context of any defined transaction models.

In order to achieve extensibility, we augment the PJava virtual machine with a *Customizable Transaction Processing Engine* (or CTPE). The intention is to give knowledgeable Java programmers the ability to define new transaction models by programming customization of the CTPE in Java using predefined *primitive components*. Primitive components are objects that abstract the key mechanisms of individual CTPE's components such as the lock and recovery managers. They give expert programmers control over the low-level mechanisms of the CTPE components without requiring any knowledge of the implementation of these components. Primitive components allow the expert application programmers to define new transaction behaviors in a manner which we believe is both simple and safe. Ordinary Java programmers can then use these transaction models conveniently in their applications.

7.1.2 Customizable Concurrency Control

Programming of customized concurrency controls will be supported by *locking capabilities* in PJava. Locking capabilities implement lock *delegation* and *ignorable conflicts* [Biliris et al., 1994, Barga and Pu, 1995]. They automatically keep track of access dependencies created because of ignored conflicts, allow queries about details of these dependencies (who depends on whom and for which objects), and issue notifications on demand to support application handling of conflicts that can't be ignored. Locking capabilities permit the convenient implementation of a large set of locking protocols. The current design assumes the granularity of locking is an object.

Locking capabilities also provide a comprehensive solution to deal transparently with arbitrary composition of threads with transactions. This is essential to give the ability to compose transactions with arbitrary existing Java code since this code may spawn an arbitrary number of threads. The issue here is to make sure that these threads remain confined within the boundary of the transaction that spawned them and enforce the behavior of their enclosing transaction, except if explicitly programmed otherwise by the transaction model implementer.

The rest of this paper is organized as follows. Section 7.1 gives an overview of our design. Section 7.3 details the programming model offered to ordinary programmers. Section 7.4 describes the framework offered to define new transaction models and how arbitrary Java code may be composed freely with transactions of any model. The customizable locking mechanism of PJava is discussed in section 7.5. Examples of how one can use the framework offered to implement various concurrency control semantics are given in section 7.6. Section 7.7 reviews related work. We conclude with a summary of the status of our design and implementation plans.

7.2 DESIGN CHOICES

Our design choices for augmenting PJava with extensible transaction management capabilities are led by three strong requirements:

- The ability to extend PJava with user-defined transaction models should not compromise the existing safety and security mechanisms of the language Java, and should not introduce new safety or security holes.

- No change may be made to the language definition.

- Data and code used within a transaction must not differ from data and code used in a non-transactional context. We call this *transaction independence.*

The following sections outline the three main principles of our design.

7.2.1 Transactions as Java objects

A transaction defines a unit of work for which some properties must be enforced. The basic interface common to all transaction models is made of operations for demarcating the boundaries of transactions, such as the classic `begin/end/abort` bracketing.

Advanced transaction models extend this common interface with new operations (e.g., operations for re-structuring the scope of transactions such as `split` and `join` [Kaiser and Pu, 1992], or for declaring a transaction as a member of a cooperative group [Fernandez and Zdonik, 1989]). Furthermore, the semantics of the same operation may vary from one transaction model to another. For instance, the operation `end` that indicates the successful termination of a transaction has different semantics depending on whether it is called in a flat transaction, in a sub-transaction in a nested transaction model, or in a member of a group transaction [Fernandez and Zdonik, 1989]. In the classic, flat transaction model, a successful termination requires that the updates made by the transaction be atomically and durably propagated to the persistent store, and made globally visible; in a nested transaction model, the successful termination of a sub-transaction requires that the updates be atomically delegated to its parent transaction, and made visible only to the descendants of its parent transaction; in a group transaction model, the updates may be required to be atomically and durably propagated to the store and made visible only to the other transactions which are members of the same group. This shows the need for a transaction management interface that is both extensible (introduction of new operations) and polymorphic (operations may be redefined).

Defining transactions as first-class objects allows the transaction concept to be introduced into Java without changing the Java language specification. These transaction classes provide a convenient framework for defining an extensible and polymorphic interface for transaction management. Transaction models are implemented as classes and their instances execute transactions according to the semantics that their class defines.

The dynamic loading and binding properties of Java permit new transaction models to be introduced as new transaction classes that*extend*, or *subclass*,[2] existing classes without rebuilding or relinking an operational system. Furthermore, existing applications does not need to be recompiled to use a new transaction class as long as the class supports the operations required by the applications (Figure 7.2 of section 7.3 illustrates how this may be done in Java).

7.2.2 Two-level interface

Our design provides Java programmers with two APIs corresponding to two levels of understanding of transaction management. It presumes two categories of programmers: specialist programmers, with skills in transaction model specification, who implement new transaction classes; and ordinary programmers who program transactional applications using the classes defined by the former group.

The intention is to organize transactional applications in four distinct layers of increasing re-usability and independence with respect to transaction management issues. Figure 7.1 summarizes this layered approach (the size of each layer is not indicative of the volume of classes).

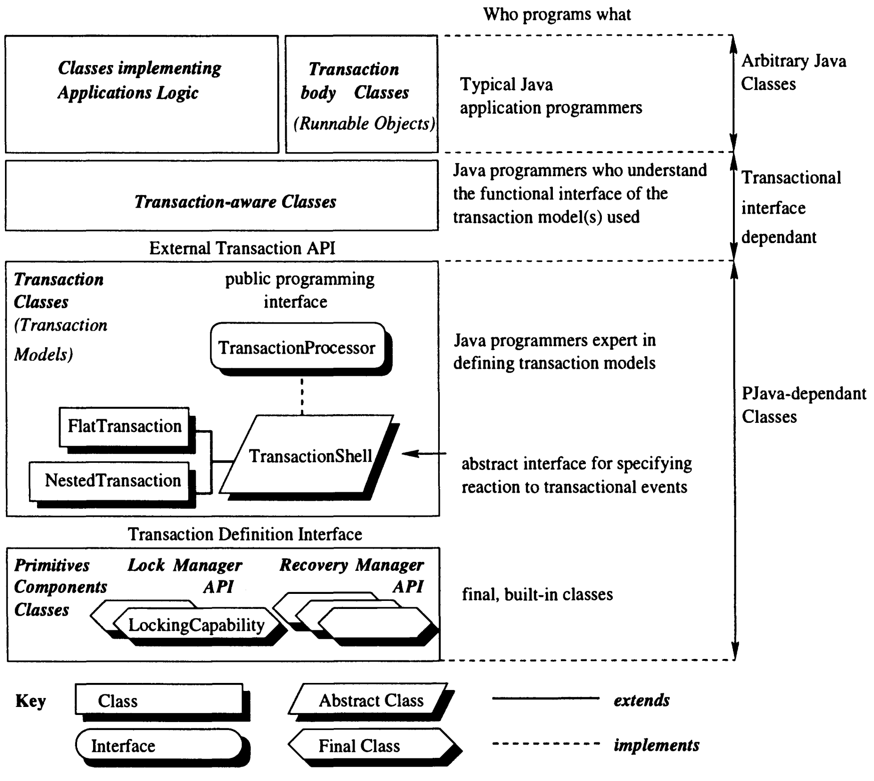

Who programs what
Classes implementing Applications Logic
Transaction body Classes (Runnable Objects)
Typical Java application programmers
Arbitrary Java Classes
Transaction-aware Classes
Java programmers who understand the functional interface of the transaction model(s) used
Transactional interface dependant
External Transaction API
Transaction Classes (Transaction Models)
public programming interface
TransactionProcessor
Java programmers expert in defining transaction models
PJava-dependant Classes
FlatTransaction
NestedTransaction
TransactionShell
abstract interface for specifying reaction to transactional events
Transaction Definition Interface
Primitives Components Classes
Lock Manager API
Recovery Manager API
LockingCapability
final, built-in classes
Key
Class
Abstract Class
extends
Interface
Final Class
implements

Figure 7.1 Extensible transaction management in PJava.

The *external transactional API* (ETAPI) provides a functional view of transaction management to ordinary application programmers. The ETAPI is for programmers who understand the transactional needs of the application. They know which transaction class is best suited for their application, and understand how to use the interface of that transaction class in their application.

Programmers using the ETAPI are responsible for the implementation of transaction-aware classes, which should account for a small portion of the application code. The transaction-aware classes isolate the rest of the application code from classes that depend on classes specific to the ETAPI. Transaction-aware classes typically encapsulate the creation of transaction objects, the definition of the boundaries of transactions, and the invocation of the methods specific to the transaction objects. Hence, above the logical software layer made of transaction-aware classes, there is no discernible difference from ordinary Java programming, except that methods execute transactionally when invoked from within a transaction. The classes implemented on top of the transaction-aware layer can be exported "as is" for execution on virtual machines supporting standard Java classes.

The programming model offered by the ETAPI requires each transaction body to be organized into one or several `Runnable` objects, i.e., objects that implement the `Runnable` interface[3]. `Runnable` objects are the basis for composing arbitrary Java code with arbitrary transaction objects. Composition via `Runnable` objects is similar to the approach taken for threads in Java and compensates for the lack of support for methods as first-class objects. The *Core Reflection* API promised with JDK 1.1 [JavaSoft, 1996] will help to limit the proliferation of `Runnable` classes.

The ETAPI itself consists of a hierarchy of transaction classes, each class implementing a given transaction model. The root of the hierarchy is the abstract class `TransactionShell`. It provides two sets of methods that correspond to the two levels of understanding of transaction management mentioned above. The first set of `public` methods provides a programming interface for defining the boundary of a transaction irrespective of the model that the transaction implements (see section 7.3). The methods of this set implement the interface `TransactionProcessor` and are `final`, therefore they cannot be overridden. The methods of the second set are all `abstract` and `protected`. They define the reactions of the transaction model with respect to transaction management events that may occur during the execution of transactions (see section 7.4). These methods are part of the mandatory methods that a transaction model implementer must define for safety and completeness reasons. Typical application programmers are not expected to define or explicitly use these methods.

Only subclasses of the class `TransactionShell` implement transaction models. They may also augment the basic interface of transactions with new transaction management primitives specific to the model they implement.

The *Transaction Definition Interface* (TDI) provides an implementation view of transaction management. The TDI is for use by the expert programmer who wishes to augment the set of available transaction models in order to satisfy new needs. The TDI consists of *Primitive components* which may be used to implement a subclass of a `TransactionShell`. Primitive components are Java classes and interfaces that expose the visible functions of individual components of the CTPE. For safety reasons, all of the classes that compose the TDI are final. In the current design, the CTPE exposes an interface to only two components: the lock and the recovery manager.

7.2.3 Implicit transaction semantics

Neither of the APIs contain functions to explicitly enforce transactional behavior. Such explicit functions would perform lock acquisition, data dependency tracking and recovery information generation.

Our design uses automation to provide these functions implicitly for the following reasons.

- Programmers are relieved of onerous and error prone tasks such as setting locks and notifying updates explicitly. This improves safety and reduces development-time.
- The majority of code then operates unchanged in a transactional context. This we call *transaction independence*. It greatly increases code re-use as the vast majority of classes do not need to call transaction classes directly.

This implicit mechanism should be contrasted with explicit mechanisms used in some Java bindings to databases and object stores. In those systems, code must be liberally sprinkled with calls explicitly claiming locks, notifying updates, etc. This means that all class re-use depends on being able to import the source form or automatically annotate the compiled form. It means that the application logic may be obscured and that classes cannot be easily exported. Perhaps most seriously, it means that it is easy to misinform the transactional engine by making an erroneous explicit call.

7.2.4 Implementation choices

To achieve implicit transaction semantics, three mechanisms are possible: pre-processing source code, post-processing compiler output[4]) or modifying an existing Java virtual machine (JVM). We have chosen the third approach for the reasons given below.

Pre-processing Java Source

Pre-processing the source code has the apparent advantage that it retains the ability for the code to run anywhere. This advantage is illusory as the code will only run where there is a transactional engine that matches the inserted calls. Such a transactional engine is not currently a standard property of Java. Unfortunately, many useful libraries are available only in class-file format. It is likely that the inserted method calls would have a significant overhead because of the many extra JVM instruction executed. Either maintenance is made more difficult because the application logic is obscured by the extra calls or the build process is made more complex by the extra pass before compilation.

Post-processing Class files

Post-processing class files means that the bytecode sequences in each method are analyzed and other bytecode sequences are inserted into them to perform the transaction control. It has the advantage that it is no longer necessary to obtain source code and that it does not obscure the application logic. Otherwise its merits and demerits are identical with pre-processing, except that a build is now more complex because of an additional pass after compilation. This approach must use only the standard bytecode instructions specified in the architecture neutral format of Java classes [Lindholm and Yellin, 1996] in order to maintain the ubiquitous execution property of Java.

Modifying an existing JVM

Our choice, of modifying an existing JVM, has the advantage that libraries of classes can be imported unchanged and that the application logic is therefore not obscured. Furthermore, changing a JVM allows optimizations that can not be possible in the two previous approaches. On the other hand, this approach has the disadvantage that we are locked in to the particular JVM implementations we are able to change, and that there are therefore some classes that will run only on our JVMs. As observed above, if you want transactional behavior, then you limit your application to run only where there is a transactional engine. If the approach proves effective, as we believe it might, then it could be implemented widely, but this has non-technical implications.

7.2.5 Outline of the modified JVM

The modified JVM identifies at runtime when transaction semantics need to be enforced, and interacts directly with the CTPE's components. For instance, the modified JVM identify instructions that access or modify objects, and replaces them with new instructions that does the required implicit transaction activities in addition to the original instruction semantics. This replacement takes place when the instruction is first executed, in much the same way as quick instructions avoid repeated dynamic binding in [Lindholm and Yellin, 1996]. This techniques avoids increasing the number of JVM cycles.

The modified JVM also keeps track of which `TransactionShell` each Java thread is running under and uses it for interacting with the CTPE. This transactional context specifies to the CTPE the (possibly customized) semantics that must be enforced.

All code must run within the scope of a transaction in PJava and all data manipulations from within a transaction are constrained to conform to that transaction's behavioral requirements. All data types (classes) are treated equally with respect to transaction management. This eliminates the need to discriminate the objects that enforce transaction properties from those that don't.

7.3 PROGRAMMING MODEL

The choice of an interface for defining the boundaries of transactions raises two issues. First, the interface must be flexible enough to encompass the largest range of programming styles. As an example, consider a simple GUI application with a single frame and several buttons to control the execution of a transaction (e.g., start a new transaction, end it, abort it or execute the op-

```
public class ATMBackend {
  private TransactionProcessor _tp;
  // Configuring the Backend server with a transaction implementation
  public ATMBackend( String model) {
    Class transactionModel = Class.forName(model);
    _tp = (TransactionProcessor) transactionModel.newInstance();
  }
  // ...
  public void serverLoop() {
    ATMRequest rq = null;
    Object [ ] rq_args = new Object[1];
    while ( (rq = nextRequest()) ! = null ) {
      switch ( rq.rqid ) {
        // fully specified request. Execute in one go
      case ATMRequest.RQ_EXECUTE:
        rq_args[0] = new Long(rq.amount);
        _tp.start(new MethodInvocation(rq.op,rq.ba,rq_args));
        _tp.claim();
        break;
        // Fragmented request
      case ATMRequest.RQ_BEGIN:
        _tp.start();
        break;
      case ATMRequest.RQ_COMMIT:
        _tp.claim();
        break;
      case ATMRequest.RQ_ABORT:
        _tp.kill();
        break;
      case ATMRequest.RQ_OP:
        rq_args[0] = new Long(rq.amount);
        _tp.enter(new MethodInvocation(rq.op,rq.ba,rq_args));
        break;
      }
    }
  }
}
```

Figure 7.2 An Auto-Teller Machine backend server.

erations selected via the buttons on its behalf). The simple bracketing of an arbitrary block of code with markers such as "begin" and "end" is not sufficient to describe the boundary of the transaction in that case, since the body of the transaction may be composed of several actions spread over the various event handling methods of the GUI application. Similarly, consider a back-end server that dispatches incoming requests to threads available in a pool. A given transaction may send more than one request, each being potentially dispatched to a different thread of the pool each time. Here again, the requirement of the application cannot be satisfied with a simple "begin"/"end" syntactic bracketing.

The second issue is related to the confinement of errors within the boundaries of the transaction that made them. More specifically, any exceptions left uncaught in the body of a transaction must remain confined within that transaction and must be propagated to the failure handling mechanism defined for that transaction. Since the body of a transaction is made of arbitrary Java methods, a transaction body can spawn an arbitrary number of threads. This makes the detection and confinement of failure even more complex.

The class TransactionShell offers a uniform framework for defining the body of a transaction. This framework enables both procedural and event-driven programming styles and deals with arbitrary multi-threaded transactions. The example given in Figure 7.2 illustrates these two styles (exception handling code is omitted for conciseness).

In both cases, transactions are defined by creating an instance of a transaction class. An instance of a transaction class is really just a shell in which to execute a transaction according to the model defined by that transaction's class. A transaction is effectively created when the shell is invoked using its `start` method. If there is no current invocation, a transaction object is nothing but a empty shell. After an invocation completes, the transaction object can be invoked again, starting another transaction.

In the procedural programming style, a transaction instance is directly associated with an object that satisfies the `Runnable` interface. The body of the transaction consists only of the `run` method of the associated `Runnable` object, and the transaction terminates when the execution of this method completes (either normally or because of a failure). The result of the transaction may be obtained using the `claim` method of the transaction object. The `start` method is provided with both synchronous and asynchronous variants, and the `claim` method is provided with both blocking and non-blocking variants.

In the event-based programming style, the transaction object is not directly associated with a `Runnable` object. Instead, the body of the transaction is made of all the `Runnable` objects that *enter* in the transaction between the boundaries explicitly defined by the programmer. When a thread calls the `enter` method of a transaction object *t* with a `Runnable` object *o*, it executes *o* on behalf of *t*. When the enter method returns, the thread is said to *leave* the transaction, i.e., it reverts the transaction it was before.

The example in figure 7.2 illustrates one possible usage of the method enter. For instance, the server may receive four consecutive requests: an RQ_BEGIN, followed by two RQ_EXECUTE, and finally an RQ_COMMIT. Upon reception of the first request, the server invokes the method start of the variable _tp, which hosts an instance of a TransactionShell. This starts a new transaction. Upon reception of each RQ_EXECUTE, the server invokes the method enter of the variable _tp. This effectively makes the server's thread participate in the transaction executed by the transaction shell _tp, for the time necessary to execute the run method of the Runnable object given as an argument to enter (here, an instance of the MethodInvocation class). In the case just described, the server's thread would participate twice in the transaction, once for each RQ_EXECUTE requests.

Threads launched from within the body of a transaction automatically participate in that transaction. Such threads are called *inner threads*. No limitation is imposed on the number of threads that may participate in a transaction concurrently, except if programmed explicitly by a transaction class implementer.

A multi-threaded transaction terminates when the end method of its shell is invoked and all its inner threads, as well as all the threads that entered the transaction prior to the call to end, are completed. Entering a transaction in a terminal state kills that transaction and raises an exception to the thread that attempted to enter the transaction.

With the model just described, programmers are forced to specify the body of their transactions, or part of them, as Runnable objects, and cannot just bracket an arbitrary block of Java code with "begin" and "end" transaction marks. The rationale for this approach is to confine exceptions that are uncaught by transaction bodies to the limit of the transaction. By forcing the encapsulation of every piece of code that participates in the body of a transaction, a TransactionShell can catch all exceptions left uncaught by these transaction bodies simply by invoking the Runnable object within a try / catch Java block, and route the transaction execution to the code that deals with failures.

Achieving the same confinement of exceptions with an approach based on block delimitation makes it necessary to either force the programmer to explicitly catch exceptions and trigger manually the appropriate action (e.g., abort the faulting transaction), or to change the language definition to incorporate transaction bracketing as suggested in [Garthwaite and Nettles, 1996]. Both options are incompatible with our requirements.

The example in Figure 7.2 also illustrates how the dynamicity of the language Java makes it possible for the ATMBackend class to change transaction model at runtime. For instance, the server loop can be augmented with an additional `case` statement for dealing with a new kind of request for changing the transaction model currently used. This additional request just needs a string containing the name of the class implementing the new transaction model. Then, using a mechanism similar to those already used in the constructor of the `ATMBackend` class, the `_tp` variable can be assigned a new instance of the new transaction class, providing this new class implements the `TransactionProcessor` interface.

The class `MethodInvocation` uses the reflexive functionalities of JDK 1.1, described in JavaSoft's draft of the Core Reflection API [JavaSoft, 1996] to support arbitrary method invocation given an object, a string holding a method name and an array of parameters needed for the method invocation.

7.4 TRANSACTION SHELL

New transaction models are introduced by defining subclasses of the abstract class `TransactionShell`. The `TransactionShell` is intended to make the definition of transaction classes simple and safe by:

- enforcing programmed transaction classes to conform to the uniform programming model defined by the public interface of the class `TransactionShell`. Any transaction, irrespective of the model it implements, can then be composed with arbitrary `Runnable` objects.
- automating all of the monitoring of transaction executions. The class `TransactionShell` relieves programmers from implementing the monitoring of all events that may occur during the execution of a transaction and impact on its behavior.
- enforcing the definition of complete transaction behavior by requiring the programmer to fill in mandatory methods that will react to transaction execution events that may happen during the execution of a transaction.
- using default, system-defined concurrency or recovery behaviors if not specified,
- using a default, system-defined recovery procedure if the user-defined one fails (i.e., is either incomplete or erroneous).

The class `TransactionShell` provides two sets of methods that correspond to the two levels of interface mentioned in section 7.1. The external interface is made of concrete public methods that implement the programming model described in the previous section. The internal interface is made of abstract protected methods. Each method specifies a response to a transaction execution event. Declaring these methods as abstract forces the programmer to specify a response to these events and thus guarantees the completeness of the transaction class's implementation. The class `TransactionShell` transparently detects these events and triggers the execution of the corresponding response.

These event handler methods return a boolean. They return `true` if the event is handled and `false` if the transaction class leaves the handling of the event to the class `TransactionShell`. The class `TransactionShell` provides a default response to each kind of event, which makes use of the default transactional attributes specified by the class implementors (see below). For efficiency, subclasses of `TransactionShell` can specify a set of events to ignore (using the `ignoreEvents` of the `TransactionShell`), which means the default handling mechanism will be triggered instead. This avoids unnecessary calls to empty event handlers.

Hence, the task of a transaction class programmer consists of just defining a concrete implementation for each of the `TransactionShell`'s abstract methods, and implementing the transaction management functions specific to the corresponding transaction model.

Table 7.1 List of the principal event handlers of a `TransactionShell`

Events	
Category	Name
transaction's state transitions	`notifyBegin`, `notifyEnd`, `notifyAbort`
inner transaction invocations	`notifyInvokee`, `notifyEndInvokee`, `notifyFailedInvokee`
participant thread activities	`notifyThreadEnter`, `notifyThreadLeave`, `notifyFailedEnteredThread`
inner thread activities	`notifyThreadStart`, `notifyThreadEnd`, `notifyFailedInnerThread`

Table 7.1 lists the principal events sent to transaction objects[5]. There are two categories of events: events related to a transition of the transaction state (e.g., initiation, normal termination or termination due to a failure), and events related to a change of the transaction structure which results from having a programming model that allows a transaction body to be composed of arbitrary participating threads running arbitrary Java code.

Events related to the transition of transaction states are typically used to implement the semantics of the transaction model. Upon transaction initiation, the transaction class must react by assigning default primitive components for concurrency control and recovery management to the notified transaction object. If some components have been omitted during transaction initiation, the notified transaction object is automatically provided with equivalent primitive components set to a system-defined behavior (e.g., strict isolation for concurrency control). Upon transaction termination, the transaction class may define its semantics for publicizing the results of its transaction (i.e., make them visible to all transactions or only some, or some of them to all, make them persistent, or delegate them, etc.).

Events related to change of the transaction structure are further categorized as *per transaction* events and *per participating thread* events.

Per transaction events concern the execution of inner transactions; they inform of the attempt to start a transaction from within the notified transaction, and of the termination of the inner transactions. Upon reception of a inner transaction event, a transaction object may react by inhibiting the transaction semantics of the inner transaction prior to executing its body. In this case, the inner transaction just executes as a normal method call. This may be useful for preventing the composition of transactions of different classes (namely if the interaction between the transaction model of the invoker and those of the invokee is unknown) or for enforcing the "flatness" of a transaction. Inhibition of inner invocations is controlled via a protected method of the class `TransactionShell`.

Per participating thread events concern individual threads that execute on behalf of a transaction. A thread participates in a transaction either because it has explicitly entered the scope of that transaction (via either the `enter` or `start` method of the public interface of a `TransactionShell` object), or because it has been created within a transaction (see section 7.3). Events notifying the participation of threads and the successful or abnormal end of their participation are generated for each kind of thread.

Before a thread participates in a transaction, it must be assigned some transactional attributes. Locking capabilities (discussed in section 7.5) are one example of such transactional attributes. These attributes are primitive component objects that define how a thread enforces the concurrency control and recovery behavior of the transaction it participates in. Assignment of transactional attributes must be done when the transaction object is notified of the participation of a thread. If no attributes are specified, the corresponding `TransactionShell` assigns default attributes to the thread. These default attributes are defined at transaction initiation time.

When a thread leaves the scope of a transaction, the class `TransactionShell` arranges for the automatic re-installation of its previous transactional attributes.

7.5 LOCKING CAPABILITIES

The class `LockingCapability` is the major component offered by the Transaction Definition Interface for customizing concurrency controls. The concurrency control of a subclass of the `TransactionShell` class is specified using instances of `LockingCapability`.

A locking capability, or capability for short, is a book-keeper of locks with a customizable conflict detection mechanism. A `TransactionShell` object can own several capabilities, but a capability belongs to a unique `TransactionShell` called the capability's *owner*. Every thread must be bound to a capability, and several threads can be bound to the same capability (typically, all threads enclosed in the same `TransactionShell` are bound to the same capability). By default, a thread is bound to the default capability of its enclosing `TransactionShell`. A thread may change the capability it is bound to during its execution, typically when it leaves a transaction and enters another one.

When a thread runs, the capability it is bound to automatically acquires the locks protecting the objects the thread operates on. Locks are acquired with respect to the conflict detection mechanism encoded in the capability. Transaction model implementors customize the conflict detection mechanism of each capability by specifying ignore-conflict relationships.

7.5.1 Ignoring Conflicts

Transactions access and manipulate objects of the persistent store by invoking operations on them. Two operations are said to be *compatible* when they do not *conflict*. Two operations conflict if their effects on the state of an object or their return values (if any) are not independent of their execution order. When an invoked operation op_i conflicts with an operation op_j in progress, a dependency[6] is formed if op_i is allowed to execute. Such dependencies reveal possible inconsistent states which may induce either an abortion of the dependent transaction or a specific commit ordering [Chrysanthis and Ramamritham, 1994]. The traditional ACID transaction model usually prevents such dependencies from happening, while "extended" transaction models allow some of these dependencies to happen temporarily.

A transaction management system must keep track of the ongoing operations and of dependencies that have been induced by the conflict. PJava uses a *customizable lock manager* for this purpose.

A lock manager detects conflicts as follows. Objects are associated with locks[7]. To perform an operation op_i on an object O, the lock protecting O must be acquired in a *locking mode* corresponding to op_i. The compatibility of locking modes (and thus of operations) is defined by a two dimensional compatibility table: one dimension corresponds to the current mode of lock, the other corresponds to the mode requested. The entry of the compatibility table corresponding to the current state of the lock and the mode of the lock request determines whether there is a conflict. If the request does not conflict, the requester is added to the set of *owners* of the lock.

PJava considers only read/write locking modes which are easy to detect transparently at the level of the virtual machine: each JVM instruction that operates on an object can be categorized as either read or write.

PJava's customizable lock manager allows a lock request to specify, in addition to the locking mode requested, a set of *ignore-conflict* relationships. An ignore-conflict relationship is a way to specify that one lock request can ignore an incompatible owner of the lock when diagnosing a conflict with the requested lock. For instance, a lock request issued from a transaction T_1, specifying a ignore-relationship with T_2 (we say that T_1 is *non-conflicting* with T_2) will ignore any conflict with T_2 when deciding whether the lock can be granted.

Ignore-conflict relationships are specified using a *labeled directed graph* where vertices are locking capabilities and edges are *ignore-conflict* relationships. Edges are directed and labeled as either *transitive* or not. We use $C_i \overset{t}{succ} C_j$ to denote a transitive edge directed from C_i to C_j and $C_i \overset{\neg t}{succ} C_j$ a intransitive edge from C_i to C_j. By default, edges are *transitive*.

This labeling of edges restricts the set of predecessors a locking capability can ignore conflicts with. We call this set, $Pred(C)$ for a capability$C$, the set of *effective predecessors* of C. Thus, given a graph of locking capabilities, a locking capability ignores conflicts with all its effective predecessors in that graph. For instance, given the graph of locking capabilities illustrated on figure 7.3, we have:

$$C_p \overset{t}{succ}\, C_q \overset{\neg t}{succ}\, C_r \Rightarrow Pred(C_r) = \{C_q\}$$
$$C_p \overset{t}{succ}\, C_q \overset{t}{succ}\, C_s \Rightarrow Pred(C_s) = \{C_p, C_q\}$$

Hence, C_s can ignore conflicts with both C_p and C_q, while C_r can ignore conflicts only with C_q.
More formally, the set of predecessors of a capability$C$ is defined as:

$$Pred(C) = Pred_{\neg t}(C) \cup [\bigcup_{\forall C_i \in Pred_t(C)} (\{C_i\} \cup Pred(C_i))] \qquad (7.1)$$

where

$$Pred_{\neg t}(C) = \{\ C_i \mid \exists\ C_i \overset{\neg t}{succ}\, C\ \} \qquad (7.2)$$

$$Pred_t(C) = \{\ C_i \mid \exists\ C_i \overset{t}{succ}\, C\ \} \qquad (7.3)$$

$Pred_{\neg t}(C)$ denotes the set of immediate predecessors that forbid transitivity; $Pred_t(C)$ denotes the set of immediate predecessors that allow transitivity.

We also define $Owner(l,M)$ as the set of locking capabilities which own lock l in mode M, and $I_{owner}(l,M)$ as the set of owners of lock l in a mode *Incompatible* with mode M. For instance, in the case of read/write locking mode[8], we have :

$$I_{owner}(l, Read) = Owner(l, Write) \qquad (7.4)$$

$$I_{owner}(l, Write) = Owner(l, Write) \cup Owner(l, Read) \qquad (7.5)$$

Lastly, we define $NCW(C)$ as the set of capabilities which are *Non-Conflicting With C*:

$$NCW(C) = \{C\} \cup Pred(C) \qquad (7.6)$$

With these definitions, a request for a lock l in mode M is granted to a locking capability C if:

$$I_{owner}(l,M) \subseteq NCW(C) \qquad (7.7)$$

As already mentioned, ignored conflicts create dependencies. More specifically, a dependency is created for each C_i such that $C_i \in (I_{owner}(l,M) \cap Pred(C_i))$. PJava keeps track of these dependencies and leaves to the `TransactionShell` programmer the interpretation and the elimination of these dependencies. Locking capabilities can be queried about their dependencies at any time. An exception is raised when a `TransactionShell` tries to release the locks of (one of) its locking capabilities that depend on at least one other capability.

There are three ways to eliminate dependencies: abort the transaction that owns the dependent capability, wait for a specific commit order before releasing the lock, or transfer the responsibility for the locks, and thereby, the visibility of the state of the objects these locks protect, to one of the transactions the dependency comes from. The latter is called *delegation* of locks.

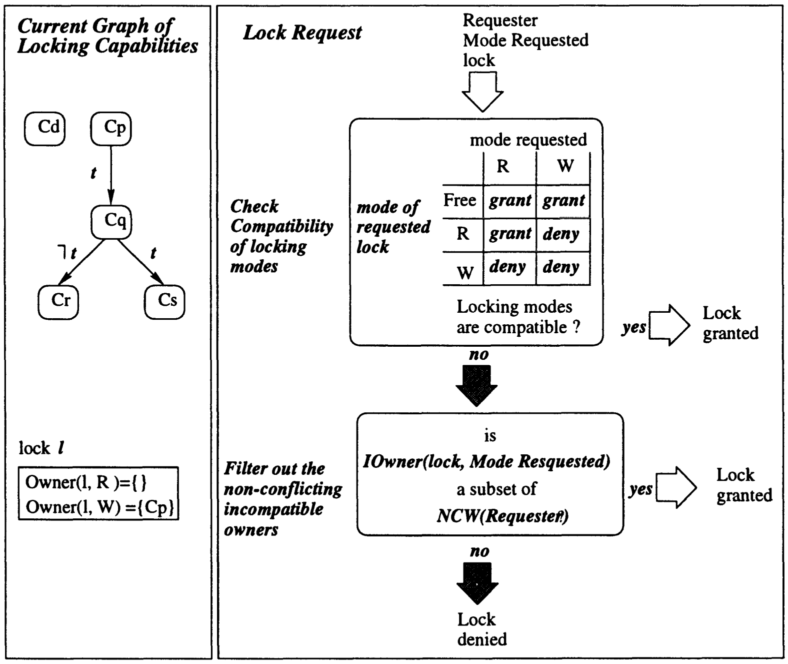

Current Graph of Locking Capabilities
Cd
Cp
t
Cq
¬t
t
Cr
Cs
lock l
Owner(l, R)={ }
Owner(l, W) ={Cp}
Lock Request
Requester
Mode Requested
lock
Check Compatibility of locking modes
mode of requested lock
mode requested
R
W
Free
grant
grant
R
grant
deny
W
deny
deny
Locking modes are compatible ?
yes
Lock granted
no
Filter out the non-conflicting incompatible owners
is IOwner(lock, Mode Resquested) a subset of NCW(Requester)
yes
Lock granted
no
Lock denied

Figure 7.3 Example of graph of locking capabilities and how it customizes the lock manager's conflict detection mechanism.

7.5.2 Delegation

Delegation of locks allows one locking capability to *atomically* transfer the responsibility for its locks to another capability. Transferring lock responsibility means changing the ownership of the delegated locks, and thus transferring the control over the visibility of the objects the delegated locks protect. It also means transferring the dependencies that have been created for acquiring these locks. For instance, if a locking capability C_1 delegates its exclusive lock on an object O to a capability C_2, C_1 is no longer able to access O after the delegation, until C_2 releases O's lock or delegates it back to C_1. Moreover, if C_1 acquired the lock on O by ignoring a conflict with a capability C_3, C_1's dependency on C_3 for O is also transferred, such that, after delegation, C_2 depends on C_3.

We speak of *global delegation* when a capability transfers the responsibility for all its locks at once, and *partial delegation* when it transfers the responsibility for only a subset of its locks. The class `LockingCapability` provides both forms of delegation. The method for global delegation takes just one parameter: the delegatee `LockingCapability`. A partial delegation takes an additional parameter to enumerate the objects whose locks must be delegated.

Global delegation is suitable for transaction models with well-defined development, that is, where the set of objects whose visibility will be delegated at the end of the transaction is known in advance. This is the case for the nested transaction model [Moss, 1981] and the colored action model [Shrivastava and Wheater, 1990]. Partial delegation is required for supporting dynamic restructuring of transactions [Kaiser and Pu, 1992], necessary in open-ended activity where developments are unpredictable and the set of objects that must be delegated is known only at the time when the need for restructuring the transaction occurs.

7.5.3 Notification

A transaction model programmer can specify notifications to be sent when its customized conflict detection mechanism diagnoses a conflict. Every locking capability can specify one *conflict notification handler*. Any object that implements the `ConflictNotificationHandler` interface can be used as a handler. This interface is essentially made of a method that takes fives parameters: the capability the handler is bound to, the mode in which it holds the lock, the locking granule the lock protects (an object in the current design), the capability that requested a conflicting lock, and the mode of the requested lock.

A `ConflictNotificationHandler` is typically used to mediate with end-users as part of the conflict resolution algorithm. Used together with the `LockingCapability`'s methods for restructuring the visibility of transactions, it allows the support of powerful multi-user collaborative environments.

7.5.4 Summary

In summary, programmers customize PJava's lock manager to achieve a given concurrency control using the following steps:

- Instance(s) of the LockingCapability required for implementing the desired concurrency control must be created for each instance of the TransactionShell using that concurrency control.

- The position of the new LockingCapability's instances in the current graph of locking capabilities must be specified. This is done using the class LockingCapability's methods for specifying immediate predecessors that allow or disallow the transitivity of the ignore-conflict (*succ*) relationship. These specifications effectively customize the conflict detection algorithm of PJava's lock manager.

- A ConflictNotificationHandler must be bound to a LockingCapability when its TransactionShell implements a model that requires notification for this instance of LockingCapability. The object implementing the ConflictNotificationHandler interface (typically, the TransactionShell itself) can use an appropriate method of the LockingCapability to resolve the notified conflict (e.g., to delegate the conflicting lock).

- binding each thread that enters the transaction associated to the TransactionShell to one of the instances of LockingCapability owned by that TransactionShell.

- querying all remaining dependencies, and determining the best way to eliminate them in order to end the TransactionShell.

Most of the time, all the management of concurrency control for transactional purposes is confined within TransactionShells. The neat effect of this mechanism is that all classes that implement the transaction bodies are not concerned at all with concurrency control issues. This allows PJava to use any existing Java classes to implement the body of any kind of transaction without changing a line of their code. Inversely, most of the code written in the context of PJava can be exported to a normal Java client.

7.6 REALIZING TRANSACTION MODELS

This section illustrates with several examples how one may use the framework we have described in this paper for realizing different transactional behaviors. Our tutorial examples focus primarily on concurrency control aspects.

7.6.1 Flat Transactions

Our first example shows the realization of a simple flat transaction model that has ACID properties. In the following, we call the resulting transaction class `FlatTransaction`.

The class `FlatTransaction` must extend the class TransactionShell in order for its instances to be known from the transaction processing engine of PJava. The implementor of the class `FlatTransaction` must then deal with two main issues: (i) the implementation of the semantics of its transaction model, namely the ACID semantics, and (ii), the definition of the responses to all the events that may occur during the execution of an instance of `FlatTransaction`; that is, how to implement the abstract event handler methods inherited from the class `TransactionShell`.

As said in section 7.4, these events relate either to a transition of the transaction state, or to a change in the transaction structure (i.e., launching of inner transactions or inner threads, etc.). The former directly concerns the implementation of the transaction semantics, while the latter is related to how well instances of `FlatTransaction` can compose with arbitrary Java code.

A first solution would consist of triggering the abort of the transaction upon the occurrence of any events notifying an attempt to change the structure of the transaction. This over-simplistic solution allows only single-threaded flat transactions and significantly reduces the usability of the class `FlatTransaction`. A better design is to accept in the transaction any new threads that attempt to participate, and to assign these threads the transactional attributes required for enforcing the transaction's properties. Nested is prohibited just by transforming into simple method calls any inner invocations of transactions.

More sophisticated schemes may be implemented by checking the class of the inner invoked transaction object, and selecting the appropriate action according to that class. For instance, some flat transaction models allow inner invocation of nested top-level transactions when they realize benevolent side-effects such as splitting overflowed indexes [Gray and Reuter, 1993]. If an instance of a class known to implement such benevolent side-effects is invoked from within a flat transaction, then the inner invocation may be allowed.

From a concurrency control point of view, the ACID properties require strict isolation between transactions. The conflict detection algorithm of an instance of `LockingCapability` created without inxxconflict, ignore-conflict relationships ignore-conflict relationships (i.e., without any predecessor in the graph of locking capabilities) realizes strict isolation. Thus, implementing strict isolation of an instance of `FlatTransaction` just requires the creation of a single `LockingCapability` without any predecessors nor successors in the graph of locking capabilities, and the binding to this capability of all the threads participating in the transaction.

To summarize, the `FlatTransaction` implementation consists of filling the inherited abstract methods as follows:

- Upon transaction initiation (i.e., method `notifyBegin`), an instance of the class `LockingCapability` is created and the current thread is bound to that locking capability. From now on, the newly created locking capability will acquire the lock of any object used by that thread.

- Upon notification of a transaction's successful termination (i.e., `notify-End`), all locks acquired by the transaction's locking capability are released. Locks are also released upon notification of transaction failure (i.e., `notifyAbort`). The difference in the implementation of the two methods lies in the management of updates which is outside of the scope of this paper.

- Upon any notification of a participating thread (i.e., methods `notify-ThreadStart` and `notifyThreadEnter`), the thread is bound to the notified transaction's `LockingCapability`.

- Upon notification of a failed participating thread (i.e., methods `notify-FailedEnteredThread` and `notifyFailedInnerThread`), `kill` method inherited from `TransactionShell` is invoked on itself to kill the transaction. The `kill` method arranges for rollback of each thread that was participating in the transaction to the point where they were before they entered the transactions. Inner threads are simply destroyed.

- Upon any notification of an inner transaction invocation, the inner transaction behavior is inhibited. This is done by calling the `inhibit` method inherited from `TransactionShell`.

- Ignore all other event notifications at instantiation time and provide the corresponding methods with an empty implementation that just returns `true`.

7.6.2 Nested Transactions

The previous example emphasizes the handling of dynamic changes to the structure of transactions and only shows a very simple usage of locking capabilities to implement concurrency control.

We now focus on the uses of the class `LockingCapability` for building advanced concurrency control semantics by demonstrating the use of locking capabilities to build an increasingly sophisticated closed nested transaction model. We refer the reader to [Härder and Rothermel, 1993] for a comprehensive description. Our examples do not make any restrictions on which transaction within a hierarchy of nested transactions is able to execute some code.

Our initial example is a very simple nested transaction model without any parallelism and restricted to single-threaded transactions. This model allows only synchronous invocation of transactions. Thus a single thread supports the execution of an entire hierarchy of nested transactions. This model may be implemented using a single subclass of TransactionShell which we call `STNestedTransaction`. Enforcing single-threading is done by aborting instances of `STNestedTransaction` upon any notification of new participating threads to these instances.

The concurrency control of the class `STNestedTransaction` can be formulated using the following locking rules:

1. A transaction *T* may acquire a lock in mode *M* if all the transactions that hold the lock in a mode incompatible with *M* are ancestors of *T*.

2. When a sub-transaction commits, it delegates all of its locks to its parent.

3. When a top-level transaction commits, it releases all of its locks.

4. When a transaction *T* aborts, it releases all of its locks (which includes the locks *T* has acquired itself and the locks delegated to *T* by its committed sub-transactions).

These rules state that a sub-transaction can see all the intermediate actions of its ancestor transactions; when a sub-transaction commits, it incorporates its effects (and those of its committed descendants) into its parent transaction, and makes them visible to all of its parent's descendants. Sub-transactions shield the surrounding world from the actions they perform (rule 4): if a sub-transaction aborts, it re-installs the visibility states of the objects it has manipulated as they were prior to its execution. Since a parent transaction never runs concurrently with its sub-transactions, transactions can always share objects with their ancestors without further concurrency control [Härder and Rothermel, 1993].

Given the above locking rules, the class `STNestedTransaction` may be implemented as follows. Each instance of `STNestedTransaction` is given an instance of `LockingCapability` at initiation time (i.e., method `notifyBegin`). If an instance *T* of `STNestedTransaction` is a sub-transaction (i.e., it has been invoked from another `STNestedTransaction`) the capability of *T* must take the capability of *T*'s parent as a immediate transitive predecessor. If *T* is not a sub-transaction, its capability must not have any predecessors. Once the capability of a `STNestedTransaction` has been created, the current thread is bound to it and started.

The graph of locking capabilities resulting from several nested invocations of `STNestedTransaction` is illustrated in Figure 7.4. The example shows the successive states of the graph after several nested invocations of transactions. The last step includes the start of another top-level `STNestedTransaction`.

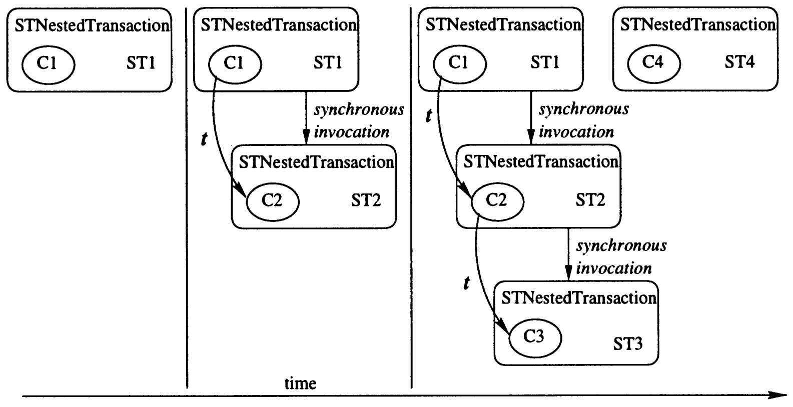


Figure 7.4 Dynamic construction of the graph of locking capabilities implementing the concurrency control of the class STNestedTransaction.

Each invocation re-arranges the graph of locking capabilities in order to customize the conflict detection algorithm of the capability of each sub-transaction T such that conflicts with ancestors of T are ignored (rule 1). For instance, the invocation of the $ST3$ object, instance of STNestedTransaction, results in the installation of a transitive edge from $C2$ to $C3$. The set of capabilities that are non-conflicting with $C3$ becomes then $NCW(C3) = \{C1, C2, C3\}$. This effectively makes the transaction $ST3$ ignoring any conflict with both transactions $ST1$ and $ST2$.

Furthermore, each hierarchy of nested transactions is strictly isolated from each other, since there is no edge between the capabilities of transactions from different hierarchies. This is the case for the transaction $ST4$ in our example.

Upon notification of successful termination (notifyEnd), instances of ST-NestedTransaction either delegate all the locks of their capability to the capability of their parent transaction if they are sub-transaction, or release the locks of their capability

Generalizing the class STNestedTransaction to support only sibling parallelism is straightforward from a concurrency control point of view since the same locking rules hold. The complexity of the implementation lies in the management of the transaction structure itself. Generalizing the class STNested-Transaction further to support both parent-child and sibling parallelism require some changes to the locking rules previously defined since a transaction can run concurrently with its sub-transactions.

The locking rules proposed for this kind of parallelism requires the management of two sets of locks per transaction T [Härder and Rothermel, 1993]: one set for the locks T acquires during its execution (called *held* locks), and one for the locks T's sub-transactions delegate to T (called *retained* locks).

The rationale for this distinction is that a parent transaction can not allow its parallel sub-transactions to access its own objects without endangering the correctness of its computation. On the other hand, the locks delegated to a parent transaction T by its committed sub-transaction must be grantable to the other running descendants of that transaction. The locking rules for this model can then be formulated as follows:

1. A transaction T may acquire a lock in mode M if (1) no other transaction holds that lock in a mode incompatible with M, (2) all transactions that retain the lock in a mode incompatible with M are ancestors of T.
2. When a sub-transaction commits, it delegates all of its locks (held or retained) to its parent which retains the delegated locks (i.e., keeps them in its retained set).
3. When a top-level transaction commits, it releases all of its locks (held or retained).
4. When a transaction aborts, it releases all of its locks (held and retained).

These locking rules may be implemented using two locking capabilities per transaction T: one for T's retained locks (called the *retainer* capability), the other for T's held locks (called the *holder* capability). Figure 7.5 exemplifies via a small example how these capabilities are used.

Upon its initiation, a transaction creates two locking capabilities. The retainer capability is set as a transitive predecessor of the holder capability. Furthermore, the retainer capability of the parent transaction is made a transitive predecessor of the retainer of the initiated transaction.

The graph of locking capabilities built for a two-level nested transaction is depicted on the top-left part of Figure 7.5. Notice that the threads participating in a transaction are bound to the holder capability of that transaction. This graph customizes the conflict detection algorithm of the locking capability C_{h4} such that conflicts with C_{r4}, C_{r2} and C_{r1} are ignored (e.g., $NCW(C_{h4}) = \{C_{r4}, C_{r2}, C_{r1}\}$); that is, only conflicts with locks retained by T_4, T_2 and T_1 are ignored by T_4. Thus, a thread participating in T_4 is not able to acquire a lock acquired by any thread of its ancestor T_1, but can acquire any of the locks retained by T_1 (i.e., locks that were delegated to T_1 by its committed descendants).

Upon notification of a successful termination, a sub-transaction delegates all of the locks of its holder capability to its retainer capability, which in turn delegates all of its locks to the retainer capability of its parent transaction.

This example demonstrates that locking capabilities can be used "passively" in order to act as a "database of locks". Such passive databases of locks are useful to implement domains of visibility such as group in engineering transaction models, or color in the colored action model of [Shrivastava and Wheater, 1990].

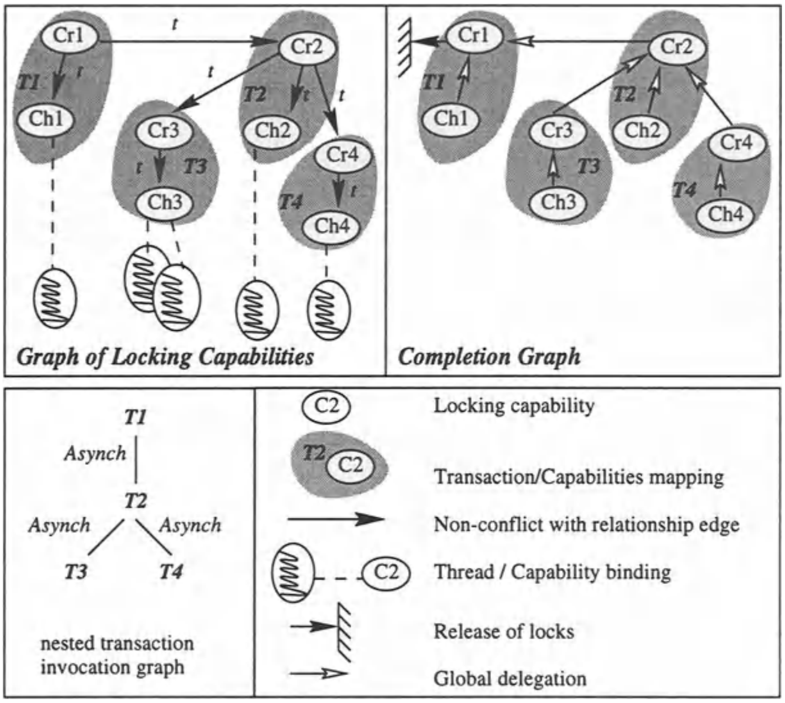

Cr1
Cr2
Cr3
Cr4
Ch1
Ch2
Ch3
Ch4
T1
T2
T3
T4
t
Graph of Locking Capabilities
Completion Graph
T1
Asynch
T2
Asynch
Asynch
T3
T4
nested transaction invocation graph
C2
Locking capability
Transaction/Capabilities mapping
Non-conflict with relationship edge
Thread / Capability binding
Release of locks
Global delegation

Figure 7.5 Graph of locking capabilities for a nested transaction model that allows both parent-child and sibling parallelism.

7.7 RELATED WORK

Our work is closely related to the previous efforts for incorporating the transaction concept into a general-purpose object-oriented programming language. Projects that have investigated these issues includes Argus [Liskov, 1988], Avalon/C++ [Eppinger et al., 1991], and Arjuna [Shrivastava and Wheater, 1990].

Argus extends the programming language CLU. It allows computations to run as atomic transactions. Transactions can be nested, though only sibling parallelism is supported. Transactions are supported directly by the language which incorporates control structures such as `topaction` or `action` for specifying (sub or top-level) transactions and `coenter` for allowing synchronous invocations of multiple transactions. Transaction properties apply only on *atomic* objects. Atomic objects are like ordinary objects except that transaction properties are automatically enforced for them.

The Avalon/C++ and Arjuna systems differ from Argus mainly in their usage of the class inheritance mechanism to provide transactional capabilities. Both systems are based on the C++ object-oriented language. In both systems, user-defined objects must inherit from system provided classes[9] in order to benefit from the transactional capabilities of the system. Programmers must then explicitly program the enforcement of the transactional properties, such as setting locks, using the methods inherited from these classes. Avalon/C++ is implemented on top of Camelot and supports nested transaction models. Arjuna has proposed the usage of the more powerful multi-colored action model [Shrivastava and Wheater, 1990], though only nested transactions have been implemented as far as we know.

All three systems make transactional properties dependent on the type of objects, and therefore introduce a dichotomy that impedes re-usability. All three systems also offer a way to customize concurrency properties of some objects by allowing the programming of user-defined atomic types. However, no framework is offered to define new transactional behaviors.

More recently, several approaches for introducing transactions into the language Java have been described [Atkinson and Jordan, 1996]. [Garthwaite and Nettles, 1996] proposes an extension of the language Java with a new control structure called `transaction`. A `transaction` statement defines a new scope; control is transferred at the end of that scope if an explicit `rollback` or `commit` statement is specified within the block. Uncontrolled leave of the block results in a default action (usually rollback). Concurrency control must be handled by the programmer via explicit setting of locks. The main drawbacks of this approach are its lack of flexibility and the necessity of changing the definition of the Java language. The authors motivated the latter as being better for integration with other similar control structures found in Java, though our design demonstrates that the same effect can be achieved without changing the language definition.

[dos Santos and Theroude, 1996] proposes a binding between Java and relational databases on top of JDBC to supplement Java with persistence. Transaction services are provided via a class `Transaction` that implements a simple flat transaction model with ACID properties. The class `Transaction` provides `begin` and `commit` methods. The body of a transaction consists of the code between these two method calls.

These two approaches to add transactions to Java share the following shortcomings:

- there is no comprehensive solution for composing arbitrary Java code with the proposed transaction constructs. In particular, the problem of composing Java threads with transactions is not addressed despite the fact that threads are an essential construct of Java.
- Only the ACID, flat transaction model is supported.
- The enforcement of the transaction properties relies on the programmers who are required to explicitly request locks or note updates in their code.

This reliance on users breaks the safety of Java and platform-independence of the application code.

Another approach that uses the *Meta-Object protocols* to introduce "non-functional" properties, such as persistence and transactions, transparently to the application programmers has been proposed by [Wu and Schwiderski, 1996]. The idea is to subclass each application class that requires the addition of non-functional properties with a *reflection class*. This reflection class overrides the methods of the application class so that method calls are wrapped with calls to a meta-object before and after the application class's code is executed. End-user classes deal with the reflection classes rather than the original application classes. Reflection classes and bindings to meta-objects are generated via pre-processing techniques. The meta-objects are implemented in Java.

[Wu and Schwiderski, 1996] proposes to use this approach to transparently supplement Java applications with user-defined concurrency control using meta-objects that implement locking. The drawbacks of this technique are (1) the loss of efficiency because of the extra Java method calls to meta-objects, (2) the loss of independence since a Java class cannot be re-used if access to its sources is not provided, (3) the proliferation of Java classes because of the generation of reflection classes. Furthermore, the meta-object protocol of [Wu and Schwiderski, 1996] requires classes to be strictly encapsulated. Direct access to instance variables must be precluded because meta-objects are not able to intercept direct manipulation to the objects they are bound to.

7.8 CONCLUSION

A design for adding extensible transaction management features to Persistent Java (PJava) has been presented. It augments PJava with an extensible pool of transaction classes, and gives expert programmers the ability to extend this pool to accommodate the needs of new applications using a Transaction Definition Interface. This interface is made of primitive components intended to ease the programming of new transaction classes. Ordinary application programmers can then select the transaction class best suited to their needs. Selection of the proper transaction class may be done at development time or at runtime using Java's dynamic binding properties.

The primitive components for programming concurrency control of transactions have also been presented. The main component is the class `LockingCapability`. It provides a simple and safe interface to a customizable locking mechanism which supports ignoring of conflicts, delegation of locks, automated tracking of data dependencies created when ignoring of conflicts is exploited, and user-defined notification of conflicts. These concepts are nicely integrated with the Java language and do not require any change to the language.

The main advantage of our approach is to offer *transaction independence* irrespective of the transaction model used: any Java classes can be used to implement the body of a transaction without any change to either the sources or the compiled form of these classes. In particular, locking capabilities and transaction shells provide a comprehensive solution to allow the arbitrary composition of threads with transactions transparently to the application. This is particularly valuable when implementing transaction bodies using Java classes delivered from third parties that cannot export the source of their classes for legal reasons. It also improves the productivity of the programmers who don't need to explicitly identify the data that may be used in a transactional way, or the code that may operate in a transactional context.

The design lacks flexibility with respect to granularity issues. At the moment, it is assumed that the transaction properties are enforced at the object granularity and for all data manipulations. However, the sizes of objects in Java are too small to realize locking efficiently. We are currently investigating additional primitive components that would enable programmers to express larger granularities while maintaining transaction independence.

Our immediate concern is to devise a lock manager that will support locking capabilities with minimal impact on the overall performance of PJava. Like most persistent object systems, PJava is optimized for navigational accesses and memory residence of active objects. Lock management implementation techniques defined today for either traditional disk-oriented [Gray and Reuter, 1993, Eppinger et al., 1991] or main-memory database systems (e.g. [Garcia-Molina and Salem, 1992, Gottemukkala and Lehman, 1992]) do not meet our needs. The former are too slow, and the latter require the database to reside permanently in main memory. Recent proposals for implementing features such as ignoring of conflicts or delegation also rely on these techniques [Biliris et al., 1994, Barga and Pu, 1995] and, therefore, don't meet our needs either.

Our solution to circumvent these problems will capitalize on our previous work on the design and implementation of efficient locking techniques for persistent object systems [Daynès et al., 1995, Daynès, 1995]. These techniques have shown performance measures encouraging enough to cope with the performance of persistent object systems. Our plan is to adapt these mechanisms to implement locking capabilities in our second prototype of PJava.

Notes

1. Atomic propagation of updates onto the persistent stable store in PJava's parlance.

2. Class extension is the mechanism for obtaining subclass in Java [Arnold and Gosling, 1996].

3. An interface in Java specifies a collection of methods without implementing their bodies [Arnold and Gosling, 1996]. When a class implements an interface, it must provide implementation of all the methods described in that interface. Interfaces provide encapsulation of method protocols without restricting the implementation to one inheritance tree.

4. Java compilers generate class files (one per class) which contain the methods in the form of sequences of Java bytecoded instructions interpreted by Java Virtual Machines.

5. Events related to primitive components, such as conflict notification events, are sent to the primitive components rather than to the transaction objects they are assigned to.

6. These dependencies are categorized as *dependencies due to behavior* in [Chrysanthis and Ramamritham, 1994].

7. Objects are chosen as the locking granule in our design.

8. PJava considers only read/write locking. However nothing precludes the use of arbitrary locking modes defined according to some semantic criteria with the mechanism just described.

9. Arjuna provides a class `StateManager` for recovery and `LockManager` for both recovery and concurrency control. The equivalent Avalon/C++ classes are, respectively, the class `Recoverable` and the class `Atomic`.

Acknowledgments

The work at Glasgow on the PJava project is supported by a grant from Sun MicroSystems Inc. and by grant GR/K87791 from the British Engineering and Physical Sciences Research Council. We are grateful to Mick Jordan and Susan Spence for their comments and careful reading of ealier drafts of this document.

8 TOWARD FORMALIZING RECOVERY OF (ADVANCED) TRANSACTIONS

Cris Pedregal Martin and Krithi Ramamritham

Department of Computer Science
University of Massachusetts, Amherst
Massachusetts, USA

Abstract: Current literature on database transaction recovery reveals a semantic gap between high-level requirements (such as the all-or-nothing property) and the low-level descriptions of how these requirements are implemented (in terms of buffers and their policies, volatile and persistent storage, shadows, etc.). At the same time, fast growing demands for recovery in both traditional and advanced transaction models require an increased understanding of the relationships between requirements and mechanisms, and the ability to craft recovery more flexibly and modularly. In this chapter we address these challenges, introducing a framework to unify the different components of recovery as well as providing the concepts and notation needd to reason about recovery protocols. We apply our framework to formalize the properties of ARIES, a production-quality recovery protocol, and show how it can accommodate ARIES/RH, a variant of ARIES that supports delegation.

8.1 INTRODUCTION

Recovery support in database transaction processing systems (TP) is provided to ensure consistency and correctness under logical as well as physical failures. Even when we confine ourselves to the Failure Atomicity (FA, the all-or-nothing) property of transactions, several considerations determine *how* recovery is achieved. For instance, different versions of ARIES [Mohan et al., 1992a], and especially the case study reported in [Cabrera et al., 1993] demonstrate the need for different policies and hence different recovery protocols and mechanisms – depending on the size of the objects, frequency of access, and the system architecture, among other considerations. Furthermore, when failure atomicity is to be achieved in parallel and distributed platforms, traditional recovery approaches do not perform well since they lead to unnecessary transaction aborts [Molesky and Ramamritham, 1995]. Finally, the growing importance of advanced applications and nontraditional transaction models as well as relaxed correctness criteria places new semantics and performance demands on recovery.

These important challenges show the need for new approaches to recovery; in particular, it is necessary to develop systematic methods to craft recovery both for the traditional FA correctness criterion, and for advanced transaction models and applications, which demand even more flexibility from the recovery subsystem. In the current state of the art in recovery, however, good design and implementation is hampered by the gap between the abstract description of the desired (high-level) recovery properties, and the very detailed implementation-oriented knowledge of how to build systems that support those properties. Specifically, there is a wide semantic gap between high-level requirements (such as the all-or-nothing property) and the low-level descriptions of how these requirements are implemented (in terms of buffers and their policies, volatile and persistent storage, shadows, etc.).

To address these problems, we introduce a framework to *unify* the different components of recovery as well as provide the concepts and notation needed to *reason about* recovery protocols.

The framework conceptualizes recovery in the context of transaction processing systems by identifying the essential ingredients of recovery and precisely prescribing their relationships thus stating various recovery properties of such systems.

By formalizing recovery properties at each abstraction level, we allow the description of abstract properties (such as the Failure Atomicity requirement) without reference to a particular implementation, and of concrete mechanisms without reference to the abstract properties they support. This separation of the *what* from the *how* allows the use of abstraction both to understand and explain recovery schemes, and to precisely state and prove the properties with which they must comply. The only related work we are aware of is [Kuo, 1996], which formalizes an-ARIES based data manager in terms of input/output automata but closer of abstraction of a particular implementation. In contrast, our formalism is broader, as it encompasses advanced transaction models, and it strives to define appropriate recovery abstractions and thus lead to a hierarchical formalization of recovery and the concomitant separation of concerns provided by different levels of abstraction.

In this chapter we apply our framework to ARIES, a production-quality, practical recovery protocol which supports traditional failure atomicity. We also broaden the scope by applying it to ARIES/RH, a variant of ARIES that supports delegation. Delegation [Chrysanthis and Ramamritham, 1994] allows a transaction to transfer responsibility over one or more of its operations to another transaction. This broadens the visibility of the delegatee, and allows control over the recovery properties of the transaction model. Thus, delegation adds substantial semantic power to a conventional Transaction Management System. Examples of Advanced Transaction Models that can be synthesized using delegate are Joint Transactions, Nested Transactions, Split Transactions, and Open Nested Transactions [Chrysanthis and Ramamritham, 1994]. See section 8.3.3 for more details on delegation.

The remainder of this chapter is organized as follows. In section 8.2 first we introduce the formal framework, presenting the ingredients of recovery and their properties in terms of histories. Then we state our assumptions and the necessary formal definitions.

In section 8.3 we use the elements of section 8.2 to formally specify various recovery properties. We begin with the requirements for Failure Atomicity and Durability, which abstractly describe *what* one expects to hold in a system that offers recovery; we also extend these requirements to take Delegation into account. Then we formalize the *assurances*, which make explicit certain usual assumptions about the semantics of the basic mechanisms; for example, no aborted operation will be later committed by the recovery mechanisms. Finally we specify the recovery *mechanisms*, the lowest level of the abstraction hierarchy. The mechanisms describe *what* recovery is built on; for example, the semantics of the persistent log.

In section 8.4 we examine a concrete recovery protocol, ARIES, and show the application of our framework to make its properties precise; we also formalize ARIES/RH, the variant of ARIES that supports delegation through rewriting of history. Finally, in section 8.5 we discuss the work involved in relaxing some of the assumptions of this chapter, and conclude with a summary.

8.2 THE FORMAL MODEL

We want to describe recovery in transaction processing systems in terms of its properties at different levels of abstraction. *Recovery properties* are statements that characterize the expected behavior of the system as a whole or some of its components. For example, at the topmost abstraction level, a recovery property of interest is Failure Atomicity, which we express as conditions on the occurrence of commits and aborts in an abstract history. At lower levels we express more specialized recovery properties in terms of more specialized entities, such as the persistent portion of the log. In this section we present our framework in terms of the various recovery properties, grouped by level of abstraction, and their relationships, both within a level, and across levels (when certain properties "ensure" or "restrict" others).

Our framework consists of recovery *ingredients* grouped in four levels of abstraction; for clarity, we use different names for the recovery properties at each level. We state the properties as predicates over histories and their projections; we introduce histories in the next section. Here we only give an overview (see Figure 8.1); in subsequent sections we define them precisely. At the top level

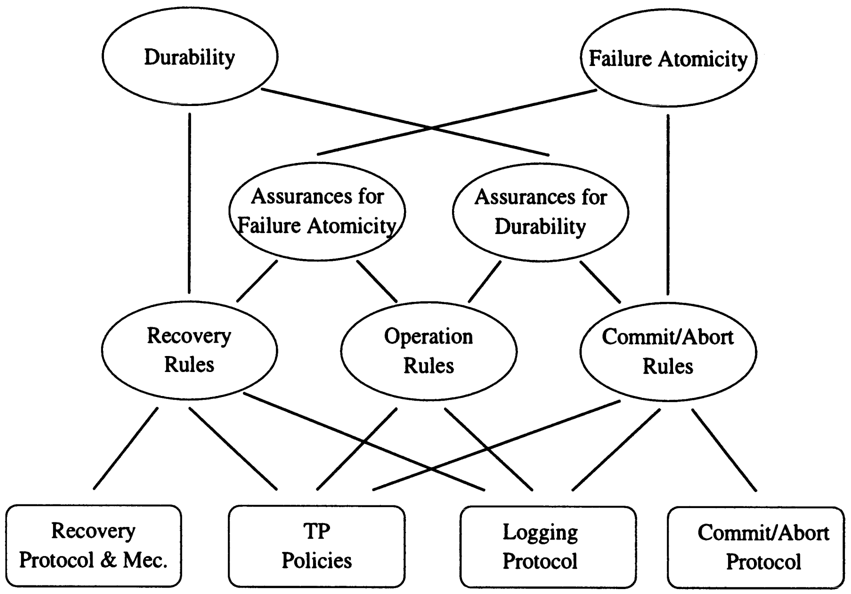


Figure 8.1 Recovery Ingredients

we have the recovery *requirements*, such as Failure Atomicity and Durability. Requirements are the properties that applications and users expect from a system that correctly supports recovery. Below the requirements lie three groups of *rules*:

CT/AB: rules to commit/abort transactions and operations,

XOPS: rules to execute operations,[1] and

XREC: rules to effect recovery.

Failure atomicity is primarily the concern of CT/AB; durability is primarily the concern of XREC. Thus the specifications of the abort and commit protocols are needed to demonstrate failure atomicity while the specifications of the recovery protocol are needed to demonstrate durability.

These three rule groups correspond to an intuitive breakdown into components, but we must also account for the interaction between rules, which we do with an intermediate level of properties which we term assurances. Specifically, ensuring failure atomicity imposes certain restrictions on XOPS and XREC to assure that they will also work toward achieving failure atomicity, while durability requires certain *assurances* on CT/AB and XOPS so that they will also work toward achieving durability. That these assurances hold must be demonstrated given the specifications of the corresponding rules; assuming the rules and the assurances one proves that the requirements are met.

The ingredients comprising the next level are specific protocols and policies (see Figure 8.1). They embody the semantics of basic mechanisms, such as the log, and algorithms for recovery. In this chapter we concentrate on the integration of a specific recovery protocol (ARIES, and its variant with delegation).

Specifically, we want to show that a given protocol meets certain requirements. This can be done through a process of refinement. For instance, given that recovery protocols operate in phases, we specify the properties of each phase. We then show that these protocol properties satisfy the rules and along with the assurances given by CT/AB and XOPS satisfy the requirements associated with the crash recovery protocol. The details of each phase (say, specified via pseudo-code) can then be used to demonstrate that the properties associated with each phase in fact hold.

The salient aspects of our framework include:

- It enables the formal specification of the correctness of transaction executions during normal run-time as well as during recovery after a crash.
- It provides a systematic delineation of the different components of recovery.
- It allows the formalization of the behavior of recovery – through a process of refinement involving multiple levels of abstraction. This leads to a demonstration of correctness.

8.2.1 Modeling Recovery through Histories

Our goal is to frame recovery in terms of how different views of the events – the histories – in a transaction system are related to each other. Informally, one can visualize a transaction system history as an execution trace – a chronological sequence – of transaction operations on data objects, such as updates, and transaction management events, such as commit. (We define precisely histories and their different events in the next section.)

We model recovery in a transaction processing system by examining the properties of its different histories; each history applies to different entities in a transaction processing system. These histories are arranged in a hierarchy and are related to each other by *projections*, and it is the properties of these projections that describe the particulars of a recovery scheme (see Figure 8.2). The histories are as follows:

- The history $\mathcal{H}$ records all the events that occur in the system – including crashes. Clearly, this is an abstraction.
- $\mathcal{L}$ denotes the history known to the system, one that is lost in the event of a crash. $\mathcal{L}$ is a projection of $\mathcal{H}$; it contains the suffix of $\mathcal{H}$ starting from the most recent crash event. ($\mathcal{L}$ can be visualized as the system log.)

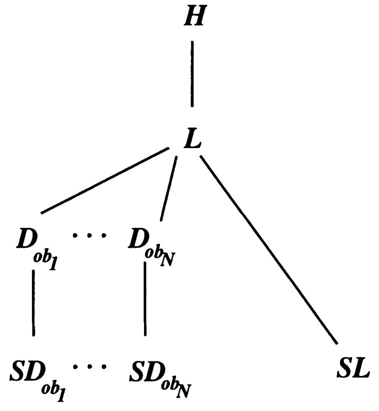


Figure 8.2 Histories in a Database Transaction System

- $\mathcal{SL}$ denotes the history known to the system in spite of crashes. This is a projection of $\mathcal{L}$. ($\mathcal{SL}$ can be visualized as the portion of the log that has been moved to stable storage).

- $\mathcal{D}_{ob}$ is a projection of $\mathcal{L}$ containing just the operations on *ob*. It denotes the state of *ob* known to the system. ($\mathcal{D}_{ob}$ can be visualized as the volatile state of *ob*).

- $\mathcal{SD}_{ob}$ is the state of *ob* that survives crashes. It is a projection of $\mathcal{D}_{ob}$; it contains the prefix of $\mathcal{D}_{ob}$. ($\mathcal{SD}_{ob}$ can be visualized as the stabilized state of *ob*).

Assumptions. For ease of explanation, we focus first on database systems:

1. that use atomic transactions,

2. that perform in-place updates and logging for recovery, and whose operations are atomic, and

3. that use serializability as the correctness criterion for concurrent transaction executions.

Then, in section 8.3.3 we relax the restrictions (1) and (3) by showing how to add the delegation primitive to the framework. Delegation allows the synthesis of advanced transaction models, whose correctness criteria relax and extend conventional serializability and Failure Atomicity.

In this hierarchy of histories we ignore the presence of checkpoints. In Section 8.5, we discuss the extensions to the formal model that can deal with further relaxations of these restrictions.

8.2.2 Events, Histories, States

Consider a database as a set of data objects each of which has a state that can be modified by operations executed on behalf of transactions. These objects may be stored in persistent storage (e.g., magnetic disk) or in volatile storage; we generally assume that all objects exist in persistent storage (some possibly in an outdated version), but some may be "cached" in faster volatile memory. Usually the system only manipulates objects in volatile memory, and this is what raises the recovery issues.

Definition 8.1 [Object and Transaction Events]

Invocation of an operation on an object is termed an object event. *The type of an object defines the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t. We write p_t when ob is clear from context or irrelevant. (For simplicity of exposition we assume that a transaction does not invoke multiple instances of $p_t[ob]$.)*

$Commit(t)$ and $abort(t)$ denote the commit and abort of transaction t, respectively. $Commit[p_t[ob]]$ and $abort[p_t[ob]]$ denote the commit and abort of operation p performed by transaction t on object ob, respectively. These are all transaction (management) events. *When a transaction event is not issued by a transaction, we add a superscript; e.g., R when an operation is issued by the recovery system.*

Definition 8.2 [Crash, Recovery and Recovery-interval]

A crash *event denotes the occurrence of a system failure; a* rec *event denotes that the system has recovered from a failure. All events are totally ordered with respect to both* crash *and* rec *events. Different crashes and recoveries in a history are indicated by a subscript, as in rec_k. Notice that during each (say, the k^{th}) recovery phase there may be multiple crashes, that we indicate with a superscript. Thus $crash_k^1$ is the first crash, and before rec_k there may be several crashes $crash_k^2, ..., crash_k^n$.*

We define the k^{th} recovery-interval *to be the part of the history (see below) bounded by $crash_k^1$ and rec_k. To reduce clutter we usually write $crash_k$ for $crash_k^1$ when it is clear from context.*

Remark: We assume throughout this chapter that recovery is completed before any normal processing is allowed to restart (but see Section 8.5). This is reflected in this formalism by the existence of a single system-wide recovery event *rec* that represents the completion of a particular recovery phase. To model recovery concurrent with normal processing it suffices to introduce a *set* of per-object recovery events, each of which represents that its corresponding object has been successfully recovered.

Definition 8.3 [Histories]

A history $\mathcal{H}$ *[Bernstein et al., 1987, Chrysanthis and Ramamritham, 1994] is a partially ordered set of events invoked by transactions. Thus, object events and transaction management events are both part of the history $\mathcal{H}$. We write $\varepsilon \in \mathcal{H}$ to indicate that the event ε occurs in the history $\mathcal{H}$. Notation $\rightarrow_{\mathcal{H}}$ denotes precedence ordering in the history $\mathcal{H}$ (we usually omit the subscript $\mathcal{H}$) and $\Rightarrow$ denotes logical implication.*

We write $\alpha \rightarrow_{\mathcal{H}}^{\neg\varepsilon} \beta$, where events $\alpha, \varepsilon, \beta \in \mathcal{H}$, to indicate that event ε does not appear between α and β (other events may appear). Formally: $\alpha \rightarrow_{\mathcal{H}}^{\neg\varepsilon} \beta \Leftrightarrow \alpha \rightarrow_{\mathcal{H}} \beta \wedge \forall e\, ((\alpha \rightarrow_{\mathcal{H}} e \rightarrow_{\mathcal{H}} \beta) \Rightarrow e \neq \varepsilon)$.

Definition 8.4 [Projections and States]

A projection $\mathcal{H}^P$ *of a history $\mathcal{H}$ by predicate* P *is a history that contains all events in $\mathcal{H}$ that satisfy predicate* P*, preserving the order. For example, the projection of the events invoked by a transaction t is a partial order denoting the temporal order in which the related events occur in the history. We abuse notation and write $\mathcal{H}^{-E}$ to denote the projection that removes all events in set E. For example, we are often interested in "projecting out" all uncommitted operations.*

$\mathcal{H}^{\varepsilon}$, is the projection of history $\mathcal{H}$ until *(totally ordered) event ε (it includes ε). $\mathcal{H}^{\varepsilon-}$ is $\mathcal{H}^{\varepsilon}$ excluding event ε.*[2]

Let $\mathcal{H}^{(ob)}$ denote the projection of $\mathcal{H}$ with respect to the operations on a single object ob.[3] Thus, a state s of an object is the state produced by applying the history $\mathcal{H}^{(ob)}$ to the object's initial state s_0 ($s = state(s_0, \mathcal{H}^{(ob)})$). For brevity, we will use $\mathcal{H}^{(ob)}$ to denote the state of an object produced by $\mathcal{H}^{(ob)}$, implicitly assuming initial state s_0.

Definition 8.5 [Uncommitted and Aborted Transaction Sets]

We denote by $Ut_{\mathcal{H}}$ the set of uncommitted *transactions in history $\mathcal{H}$: $t \in Ut_{\mathcal{H}} \Leftrightarrow commit(t) \notin \mathcal{H}$. The set of* aborted *transactions $At_{\mathcal{H}}$ in history $\mathcal{H}$: $t \in At_{\mathcal{H}} \Leftrightarrow abort(t) \in \mathcal{H}$. Similarly we define the set of* pending (uncommitted and unaborted) transaction operations *$Pp_{\mathcal{H}}$, the set of* aborted operations *$Ap_{\mathcal{H}}$ and the set of* recovery operations *$Rp_{\mathcal{H}}$. We drop the subscript, t, when it is clear from context.*

Definition 8.6 [Physical and Logical States]

The physical state *of an object* ob *after history $\mathcal{H}$ is the state of* ob *after $\mathcal{H}^{(ob)}$ is applied to the initial state of* ob. *The* physical database state *after $\mathcal{H}$ is the physical state of all the objects in the database after $\mathcal{H}$ is applied. This is denoted by $\mathcal{H}_P$.*

Consider the history $\mathcal{H}^{-Rp \cup Ap}$ that results from removing from a history $\mathcal{H}$ all object operations performed by the recovery system and all aborted operations. The logical database state, *denoted by $\mathcal{H}_L$, is the physical state that results*[4] *from $\mathcal{H}^{-Rp \cup Ap}$.*

Definition 8.7 [Equivalence of Histories]

Two histories $\mathcal{H}', \mathcal{H}''$ are equivalent *when the (logical or physical) state of the database after the execution of $\mathcal{H}'$ is the same as the state after the execution of $\mathcal{H}''$ on the same initial state. Different equivalence relations result when the logical (L) or physical (P) state of the database are considered for each of $\mathcal{H}'$ and $\mathcal{H}''$. We define three: $\mathcal{H}'_P \equiv \mathcal{H}''_P$ $\mathcal{H}'_P \equiv \mathcal{H}''_L$ $\mathcal{H}'_L \equiv \mathcal{H}''_L$.*

Two histories $\mathcal{H}', \mathcal{H}''$ are operation commit equivalent *when they are equivalent and all operations committed in one are committed in the other and vice-versa. We denote them $\mathcal{H}'_P \equiv^c \mathcal{H}''_P$ $\mathcal{H}'_P \equiv^c \mathcal{H}''_L$ $\mathcal{H}'_L \equiv^c \mathcal{H}''_L$.*

8.3 REQUIREMENTS, ASSURANCES & RULES

In transaction processing systems that adopt the traditional transaction model, transactions must be *failure atomic*, i.e., satisfy the all-or-nothing property. Failure atomicity requires that (a) if a transaction commits, the changes done by *all* its operations are committed[5] and (b) if a transaction aborts unilaterally (logical failure) or there is a system failure before a transaction commits, then *none* of its changes remain in the system. *Durability* requires that changes made by a transaction remain persistent even if failures occur after the commit of the transaction.

Thus, the goals of recovery are to ensure that enough information about the changes made by a transaction is stored in persistent memory to enable the reconstruction of the changes made by a committed transaction in the case of a system failure. It should also enable the rolling back of the changes made by an aborted transaction by keeping appropriate information around. These two goals must be accomplished while interfering as little as possible with the normal ("forward") operation of the system.

In this section we use the formalism of section 8.2 to state the properties that characterize recovery at different levels of abstraction, from abstract to concrete (see Figure 8.1). We begin by specifying the *requirements* of Failure Atomicity and Durability, and how they are affected by the introduction of Delegation. We then discuss *rules* and *assurances* that enable the construction of recovery, and the associated restrictions they place on the recovery system. Finally, we discuss specific recovery *mechanisms*. This sets the stage for the discussion of a specific protocol (ARIES) in Section 8.4.

8.3.1 Durability

Durability requires that committed operations should persist in spite of crashes.

1. When recovery is complete (after the recovery-interval $(crash_k^1, rec_k)$), the logical state is equivalent to the state produced by committed operations just before $crash_k^1$:

$$\forall k (\mathcal{H}_L^{crash_k^1 -} \equiv^c \mathcal{H}_L^{rec_k})$$

2. After recovery, the physical state of $\mathcal{L}$ mirrors the logical state of $\mathcal{H}$ at that point:

$$\forall k(rec_k \in \mathcal{H} \Rightarrow \mathcal{L}_P^{rec_k} \equiv \mathcal{H}_L^{rec_k})$$

8.3.2 Failure Atomicity

Transaction t is *failure atomic* if the following two conditions hold:

All Operations invoked by a committed transaction are committed:

$$(commit(t) \in \mathcal{H}) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in \mathcal{H}) \Rightarrow (commit[p_t[ob]] \in \mathcal{H})).$$

Nothing Operations invoked by an aborted transaction are aborted:

$$(abort(t) \in \mathcal{H}) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in \mathcal{H}) \Rightarrow (abort[p_t[ob]] \in \mathcal{H})).$$

8.3.3 Failure Atomicity and Delegation

Delegation allows a transaction to transfer responsibility for an operation to another transaction. After the delegation, the fate of the operation, i.e., its visibility and conflicts with other operations, are dictated by the scope and fate of the delegatee transaction. In this section we give just the essential definitions.

Traditionally, the transaction invoking an operation is also responsible for committing or aborting that operation. With delegation the invoker of the operation and the transaction that commits (or aborts) the operation may be different. Delegation is useful in synthesizing advanced transaction models because it broadens the visibility of the delegatee, and because it controls the recovery properties of the transaction model. The broadening of visibility is useful in allowing a delegator to selectively make tentative and partial results, as well as hints such as coordination information, accessible to other transactions. The control of the recovery makes it possible to decouple the fate of an operation from that of the transaction that made the operation; for instance, a transaction may delegate some operations that will remain uncommitted but alive after the delegator transaction aborted. Examples of Advanced Transaction Models that can be synthesized using delegate are Joint Transactions, Nested Transactions, Split Transactions, and Open Nested Transactions [Chrysanthis and Ramamritham, 1994]. For extensive treatments of delegation, see [Chrysanthis and Ramamritham, 1994]; delegation in the context of recovery is examined in [Pedregal Martin and Ramamritham, 1997].

Definition 8.8 [Invoking Transaction]

A transaction t that issues an operation p on object ob is called the invoking *transaction, and we denote it with a subscript:* $p_t[ob]$. *We drop the subscript when it is obvious or irrelevant.*

Definition 8.9 [Responsible Transaction] *A transaction t responsible for an operation p is in charge of committing or aborting p, unless it delegates it:* ResponsibleTr($p[ob]$) $= t$ *holds from when t performs* $p[ob]$ *or t is delegated* $p[ob]$ *until t either terminates or delegates* $p[ob]$.

Notice that without delegation, the transaction responsible for an operation is always the invoking transaction.

Definition 8.10 [Delegation]

We write $delegate(t_1, t_2, p_{t_0}[ob])$ *to denote that* t_1 *delegates operation p (originally invoked by* t_0*) to transaction* t_2*. For this delegation we have:*

Precondition ResponsibleTr($p[ob]$) $= t_1$.

Postcondition: ResponsibleTr($p[ob]$) $= t_2$.

Adding Delegation. We now examine the consequences of adding the notion of delegation to the basic framework. This is an important extension as the semantics of delegation allows the synthesis of advanced transaction models whose correctness criteria relax serializability in various ways. In the presence of delegation, we say that transaction t is *failure atomic* if the following two modified conditions hold:

All' All operations a committed transaction is responsible for are committed:

$$\begin{aligned}&(commit(t') \in \mathcal{H}) \Rightarrow \\ &\qquad \forall ob\ \forall p\ \forall t\ ((p_t[ob] \in \mathcal{H}\ \wedge\ ResponsibleTr(p_t[ob]) = t') \Rightarrow \\ &\qquad\qquad (commit[p_t[ob]] \in \mathcal{H})),\end{aligned}$$

Nothing' All operations an aborted transaction is responsible for are aborted:

$$\begin{aligned}&(abort(t') \in \mathcal{H}) \Rightarrow \\ &\qquad \forall ob\ \forall p\ \forall t\ ((p_t[ob] \in \mathcal{H}\ \wedge\ ResponsibleTr(p_t[ob]) = t') \Rightarrow \\ &\qquad\qquad (abort[p_t[ob]] \in \mathcal{H})),\end{aligned}$$

Changes with the addition of delegation. In the absence of delegation, the transaction that issued an operation remains responsible for it. Therefore, the abort/commit of one dictates the abort/commit of the other, respectively. When a transaction t delegates an operation p to another transaction t' it decouples p's fate from its own (in the sense of committing or aborting). This causes some changes; however, most of the recovery properties remain unchanged, because they are formulated in terms of operations, not transactions.

We now focus on the next level of specification, which is concerned with *assurances*. These are properties that the various components must preserve to allow the more abstract requirements to be satisfied. The components correspond to well-understood mechanisms and protocols, properties of which are rarely stated explicitly.

8.3.4 Assurances for Failure Atomicity

Here we describe the restrictions imposed on recovery mechanisms to provide assurances for Failure Atomicity. They are described as restrictions as they limit what can be done by the recovery mechanism to obtain the necessary assurances. Usually these restrictions are implicitly assumed by recovery schemes; they reflect the broad notion that the recovery mechanism is "well-behaved," i.e., that it does not abort committed operations or vice-versa, and that it only operates during the recovery phase.

1. No aborted operation should be committed by the recovery system:

   $$\forall p \forall t \forall ob(abort[p_t[ob]] \in \mathcal{H} \Rightarrow (commit^R[p_t[ob]] \notin \mathcal{H}))$$

2. No committed operation should be aborted by the recovery system:

   $$\forall p \forall t \forall ob(commit[p_t[ob]] \in \mathcal{H} \Rightarrow (abort^R[p_t[ob]] \notin \mathcal{H}))$$

3. Outside of a recovery-interval, object, commit, and abort operations cannot be invoked by the recovery system:

   $$\forall t \forall p \forall ob(\varepsilon \in \{p_t^R[ob], commit^R[p_t[ob]], abort^R[p_t[ob]]\}) \Rightarrow \forall k(rec_k \rightarrow^{\neg\varepsilon} crash^1_{k+1})$$

   We define rec_0 to precede all events in $\mathcal{H}$ so that $k = 0$ covers the interval before the first crash.

4. If the recovery system aborts a transaction operation, then it will eventually abort the transaction:

   $$\forall t \forall p \forall ob(abort^R[p_t[ob]] \in \mathcal{H} \Rightarrow abort^R[t] \in \mathcal{H})$$

Delegation Assurances. The only restriction that needs reformulating is (4).

4.' If the recovery system aborts an operation, then it will eventually abort the operation's responsible transaction:

   $$\forall p, ob, t(abort^R[p_t[ob]] \in \mathcal{H} \Rightarrow abort^R[ResponsibleTr(p_t[ob])] \in \mathcal{H})$$

8.3.5 Assurances for Durability

Here we describe the assurances provided to the recovery component so that it can achieve durability. The first assurance is central to the semantics of having a reliable logging mechanism. The rest can be seen as "technical" (i.e., for the completeness of the formalism): the next three make explicit the usual assumptions of "good behavior," and the last one ensures the base case for induction proofs (on the length of histories).

1. All operations between two consecutive crashes $crash_i$ and $crash_j$ (or between the initial state and $crash_1$) which appear in $\mathcal{H}^{crash_j-}$ also appear in $\mathcal{L}^{crash_j-}$, and they appear in the same order.

2. No operations are invoked by other systems during the recovery period (the recovery system may invoke operations to effect recovery). Formally:

   $\forall p \forall t \forall ob \forall S(S \neq R \wedge \varepsilon \in \{p^S[ob], commit^S[p_t[ob]], abort^S[p_t[ob]]\}) \Rightarrow$
   $\qquad \forall k, i(crash_k^i \rightarrow^{\neg\varepsilon} rec_k)$

3. No other part S of the transaction system commits an operation which was previously aborted. Formally:

   $\forall S \forall p \forall t \forall ob(S \neq R \wedge abort[p_t[ob]] \in \mathcal{H} \Rightarrow$
   $\qquad \neg(abort[p_t[ob]] \rightarrow_{\mathcal{H}} commit^S[p_t[ob]]))$

4. No other part of the system aborts an operation which was previously committed.

   $\forall S \forall p \forall t \forall ob(S \neq R \wedge commit[p_t[ob]] \in \mathcal{H} \Rightarrow$
   $\qquad \neg(commit[p_t[ob]] \rightarrow_{\mathcal{H}} abort^S[p_t[ob]]))$

   S refers to different components of the transaction processing system.

5. History and log are both empty at the beginning: $\mathcal{H}^0 = \phi = \mathcal{L}'$.

8.3.6 Recovery Mechanisms Rules

Here we specify the mechanisms that support recovery in terms of rules. For example, if an operation was uncommitted before a crash, it will not be committed by the recovery system.

1. After recovery, history $\mathcal{L}$ reflects the effects of all committed operations, all aborted operations, all transaction management operations and all system operations (which includes undos of aborted operations). Those operations invoked by transactions, which have neither been committed nor aborted, are given by $Pp_{\mathcal{L}^{crash_k^1-}}$ which we denote *Actops*. None of these operations is reflected.

   $\forall k(\mathcal{L}_P^{rec_k} \equiv^c (\mathcal{L}_P^{crash_k^1-})^{-Actops})$

2. During recovery, an operation performed by a transaction which is neither committed nor aborted before the crash is aborted by the recovery system.

   $\forall p \forall t \forall ob \forall k(p_t[ob] \in (Actops) \Rightarrow$
   $\qquad (crash_k^1 \rightarrow_{\mathcal{H}} abort^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k)))$

3. An operation invoked by a transaction committed before a crash is not aborted by the recovery system.

   $\forall t \forall p \forall ob \forall k(commit[p_t[ob]] \in \mathcal{L}^{\rfloor\nabla\dashv\int\langle^{\infty}_{\|}-} \Rightarrow$
   $\qquad \neg(crash_k \rightarrow_{\mathcal{H}} abort^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k))$

4. If an operation invoked by a transaction was uncommitted before a crash, it is not committed by the recovery system.

$$\forall t \forall p \forall ob \forall k (commit[p_t[ob]] \notin \mathcal{L}^{\rfloor\nabla\dashv\int\langle^{\infty}_{\|}-} \Rightarrow \\ \neg(crash_k \rightarrow_{\mathcal{H}} commit^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k))$$

5. The recovery system does not invoke any operations outside the recovery-interval.

$$\forall p, ob, t(\varepsilon \in \{p_t^R[ob], commit^R[p_t[ob]], abort^R[p_t[ob]]\}) \Rightarrow \\ \forall k(rec_k \rightarrow^{\neg\varepsilon} crash_{k+1})$$

6. If the recovery system aborts an operation invoked by a transaction in a recovery interval, it also aborts the transaction before the end of that recovery interval.

$$\forall p, ob, t, k((crash_k \rightarrow abort^R[p_t[ob]] \rightarrow rec_k) \Rightarrow \\ (crash_k \rightarrow abort^R[t] \rightarrow rec_k))$$

8.3.7 Logging and Commit/Abort Protocols

The commit/abort and logging protocols guarantee the following:

AB-UNDO The undo of an operation is equated with the abort of the operation:

$$\forall p \forall ob \forall t (p_t[ob] \in \mathcal{L} \Rightarrow (undo^R(p_t[ob]) \in \mathcal{L} \Leftrightarrow abort^R(p_t[ob]) \in \mathcal{L}))$$

LOG-CT All the committed operations are in the stable log at the time of a crash:

$$\forall i \forall p \forall t \forall ob (commit(p_t[ob]) \in \mathcal{H}^{crash_i^1 -}) \Rightarrow (p_t[ob] \in \mathcal{SL}^{\rfloor\nabla\dashv\int\langle^{\infty}_{\rangle}-})$$

8.4 A SPECIFIC RECOVERY PROTOCOL

In this section we indicate how to apply our framework to a specific recovery protocol, ARIES, and how, when we extend ARIES with delegation (resulting in ARIES/RH) our framework adapts and covers the new extensions. First, we give an informal overview of ARIES and ARIES/RH. Second, we specify the assurances that ARIES and ARIES/RH assume from other components of recovery. Third, we specify the correctness properties satisfied by ARIES and ARIES/RH. Then, we show that the second and the third together conform to the rules that recovery protocols in general must satisfy. For brevity we present just a sample of the proofs.

8.4.1 Overview of ARIES and ARIES/RH

We first review ARIES to establish context and terminology, and then we explain the modifications necessary for ARIES/RH [Pedregal Martin and Ramamritham, 1997]. The ARIES recovery method follows the repeating history

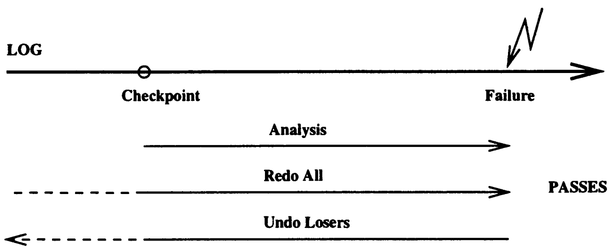


Figure 8.3 ARIES passes over the log

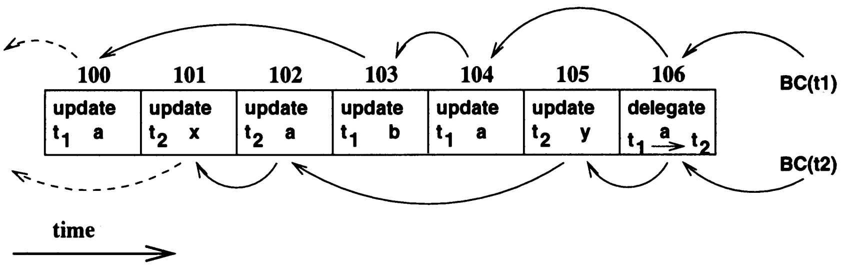


Figure 8.4 Backward Chains in the log

paradigm and consists of three phases[6] (see figure 8.3). Immediately after a crash, ARIES invalidates the volatile database. Analysis identifies which transactions must be rolled back (losers) and which must be made persistent (winners). Redo repeats history, redoing all transaction operations that had taken place up to the crash. Finally, using the analysis information, undo removes the operations from loser transactions.

ARIES keeps, for each transaction, a *Backward Chain* (BC, see figure 8.4). All the log records pertaining to one transaction form a linked list BC, accessible through *Tr_List*, which points to the most recent one. ARIES inserts *compensation log records* (CLRs) in the BC after undoing each log record's action.[7] Applying $delegate(t_1, t_2, ob)$[8] is tantamount to removing the subchain of records of operations on *ob* from BC(t_1) and merging it with BC(t_2). Next we discuss ARIES/RH, which supports delegation without modifying the log. First we present the data structures, and we explain the normal processing. We then examine recovery processing, first the forward (analysis & redo) pass and then the backward (undo) pass.

8.4.1.1 Data Structures. We must know which operations on which objects each transaction t is responsible for, i.e., its $Op\_List(t)$. For that we use the *Transaction List* and expand each transaction's *Object List* found in conventional Database Systems; we also add a `delegate` type log record.

field name	*function*
LSN	position within the LOG
Tor	transaction id of delegator
TorBC	delegator's backward chain
Tee	transaction id of delegatee
TeeBC	delegatee's backward chain

Figure 8.5 Fields of the delegate log record

Tr_List. The Transaction List [Bernstein et al., 1987, Gray and Reuter, 1993, Mohan et al., 1992a] contains, for each Trans-ID, the LSN for the *most recent* record written on behalf of that transaction, and, during recovery, whether a transaction is a *winner* or a *loser.*[9]

Ob_List. For *each* transaction t there is an Object List $Ob\_List(t)$. In terms of Op_List: $Ob\_List(t) = \{ob \mid \exists p_{t_0}[ob] \in Op\_List(t)\}$, i.e., the objects for which there is an operation for which t is responsible. The operation $p_{t_0}[ob]$ may have been invoked by t_0 and the responsibility transferred to t via delegation.

When transactions are responsible for specific operations (not a whole object), a certain object may appear in more than one *Ob_List* (but the associated operations will be different).[10] We identify the *operations* that a transaction is responsible for by introducing the notion of *scope.*

For each object *ob* in $Ob\_List(t_1)$ there is a *set* of scopes *Scopes*, that covers the operations to *ob for which* t_1 *is currently responsible.* A scope is a tuple (t_0, l_1, l_2) where t_0 is the transaction that actually did the operations (the invoking transaction), l_1 is the first, and l_2 the last LSN in the range of log records that comprise the scope.

Delegate Log Records. We add a new log record type: `delegate`. Its type-specific fields (see figure 8.5) store the two transactions and the object involved in the delegation.

8.4.1.2 Normal Processing.

We sketch how ARIES/RH extends ARIES by showing how to handle delegations and operations. Other transactional events are modified as well; the reader is referred to [Pedregal Martin and Ramamritham, 1997] for a complete account.

- $p_t[ob]$

  1. ADJUST SCOPES. If this is the first operation of t to *ob* since either t started or last delegated *ob* we must open a new scope. Otherwise, there is an active scope of t on *ob* that we must extend.

```
if $ob \notin Ob\_List(t)$ then $Ob\_List(t) \leftarrow Ob\_List(t) \cup \{ob\}$ ;
if $(t,\_,\_)$[11] $\notin Ob\_List(t)[ob]$
  then create new scope
  else extend existing scope
```

- $delegate(t_1, t_2, ob)$

1. WELL-FORMED? Verify that $ob \in Ob\_List(t_1)$, which tests, for this case, the precondition: $pre(delegate(t_1, t_2, op[ob])) \Rightarrow (ResponsibleTr(op[ob]) = t_1)$.
2. PREPARE LOG RECORD(S).
   Record delegator, delegatee.

   $Rec.tor \leftarrow t_1;\ Rec.tee \leftarrow t_2;$

   Link this log record into t_1's and t_2's backward chains.

   $Rec.torBC \leftarrow \mathrm{BC}(t_1).PrevLSN;\ \ Rec.teeBC \leftarrow \mathrm{BC}(t_2).PrevLSN.$
3. TRANSFER RESPONSIBILITY. Move operations on ob from $Op\_List(t_1)$ to $Op\_List(t_2)$.
   Add ob to delegatee's *Ob_List* and record that ob was delegated by t_1.

   $Ob\_List(t_2) \leftarrow Ob\_List(t_2) \cup \{ob\}\ ;\ \ Ob\_List(t_2)[ob].deleg \leftarrow t_1.$

   Pass delegator's Scopes for ob to the delegatee and remove ob from the delegator's *Ob_List*.
4. WRITE DELEGATION LOG RECORD(S).
   Write log record and mark it as the current head of the backward chains of delegator and delegatee.

   $\mathrm{LOG[CurrLSN]} \leftarrow Rec\ ;\ \ \mathrm{BC}(t_1) \leftarrow \mathrm{CurrLSN}\ ;\ \ \mathrm{BC}(t_2) \leftarrow \mathrm{CurrLSN}.$

8.4.1.3 Crash Recovery. In the rest of this section, we present the recovery phase of ARIES/RH, which includes a forward pass and a backward pass.

Forward Pass. For brevity, we describe only the results of the forward pass of recovery. Details can be found in [Pedregal Martin and Ramamritham, 1997]. Before the first pass of recovery starts, $Winners = Losers = \phi$. At the end of the forward pass *Winners*, *Losers*, and *Object List*s are up to date, including the scopes of the operations. Specifically, after the Forward Pass the state is:

- *Ob_Lists* are restored to their state before the crash, for all transactions.
- *Winners* has all the transactions whose operations must survive (i.e., which had committed before the crash). *Losers* has those whose operations must be obliterated.
- *LoserObs* includes all objects in the *Ob_List*s of loser transactions. We compute it after the forward pass ends, as $LoserObs = \bigcup_{t \in Losers} Ob\_List(t)$.

Backward Pass. To undo loser transactions, ARIES continually undoes the operation with maximum Log Sequence Number (LSN), ensuring monotonically decreasing (by LSN) accesses to the log, with the attendant efficiencies.

ARIES undoes all the operations *invoked* by a loser transaction. In the presence of delegation, what we need instead is to undo *all the operations that*

were ultimately delegated to a loser transaction. Notice that by undoing the *loser* operations instead of the operations invoked by loser transactions, we are in fact applying the delegations, as we undo according to the fate of the final delegatee of each operation.[12]

We show in [Pedregal Martin and Ramamritham, 1997] that it suffices to keep information on operation scopes to efficiently undo loser operations. There we also discuss how undo and delegation are integrated in the backward pass. Operation and delegation are the only records that require special processing. As with ARIES, ARIES/RH also visits each log record at most once and in a monotonically decreasing way. This reduces the cost of bringing the log from disk.

8.4.2 Formalizing some properties of ARIES and ARIES/RH

Policies. ARIES assumes the STEAL and NO-FORCE policy combination. That is, the restrictions associated with NO-STEAL and FORCE, which we formalize next, do not apply.

NO-STEAL requires that no uncommitted operations be propagated to the stable database. If an operation is stable, its transaction must have committed. Formally:

$$\forall \mathcal{D}^{(ob)} \in prefix(\mathcal{L}^{(ob)}), \quad \forall \varepsilon \in \mathcal{D}^{(ob)}$$
$$(p_t[ob] \rightarrow_{\mathcal{D}^{(ob)}} \varepsilon) \Rightarrow (commit(t) \rightarrow_{\mathcal{L}^{(ob)}} \varepsilon)).$$

Notice that this specification of NO-STEAL does not impose an ordering or logging strategy; nor does it say how to record that a transaction is considered committed.

FORCE prescribes that updated objects must be in the persistent database for a transaction to commit. Formally:

$$\forall \mathcal{D}^{(ob)} \in prefix(\mathcal{L}^{(ob)}), \quad \forall \varepsilon \in \mathcal{D}^{(ob)}(commit(t) \rightarrow_{\mathcal{L}^{(ob)}} \varepsilon)) \Rightarrow$$
$$(p_t[ob] \rightarrow_{\mathcal{D}^{(ob)}} \varepsilon).$$

Operation execution, Commit, and Abort.

WAL: No operation to the stable database can be installed before a corresponding record of the operation is stored in the persistent log. This is called the Write-Ahead Log (WAL) rule. Formally:

$$\forall \mathcal{D}_{(ob)} \in prefix(\mathcal{L}_{(ob)}) \, \forall \varepsilon \in \mathcal{D}_{(ob)} (p_t[ob] \rightarrow_{\mathcal{D}_{(ob)}} \varepsilon) \Rightarrow (p_t[ob] \rightarrow_{S\mathcal{L}_{(ob)}} \varepsilon))$$

Semantics of Transaction Abort: If a transaction s is aborted, no other transaction t can operate on the same object until s's operations are aborted. Formally:

$$\forall s \forall t \; (q_s[ob] \rightarrow_{\mathcal{L}} p_t[ob] \wedge abort(s) \rightarrow_{\mathcal{L}} p_t[ob]) \Rightarrow abort[q_s[ob]] \rightarrow_{\mathcal{L}} p_t[ob]$$

Commit: The system considers a transaction committed when it has persistently logged all the operations and the commit record for the transaction. Formally:

$$\forall L \in \mathit{prefix}(\mathcal{L})\ \forall \varepsilon \in \mathcal{SL}$$
$$(\mathit{commit}(t) \rightarrow_L \varepsilon) \Rightarrow (\mathit{commit}(t) \rightarrow_{\mathcal{SL}} \varepsilon) \wedge \forall p_t \in L(p_t \rightarrow_{\mathcal{SL}} \varepsilon)$$

Winners, Losers, LoserObs.

- $t \in \mathit{Winners} \Longleftrightarrow (\mathit{Commit}(t) \rightarrow \mathit{Crash})$
  t is a winner if it committed before the crash.

- $t \in \mathit{Losers} \Longleftrightarrow (\mathit{Begin}(t) \rightarrow \mathit{Crash} \wedge \nexists \mathit{Commit}(t) \in \mathcal{H})$
  t is a loser if it was active but did not commit before the crash.

  Losers: an active transaction is by default a loser. If there is a commit record before the crash, its transaction is moved to *Winners*. Note that these sets are disjoint.

- $\mathit{LoserObs} = \bigcup_{t \in \mathit{Losers}} \mathit{Ob\_List}(t)$
  i.e., $ob \in \mathit{LoserObs} \Rightarrow \exists t \in \mathit{Losers} : ob \in \mathit{Ob\_List}(t)$
  LoserObs is the set of all objects for which there is a loser transaction that is responsible for an operation to that object. This means that a loser object has at least one operation that will be undone.

Specification of ARIES. In the following, $\mathit{post}(P)$ refers to the postcondition that a particular phase P (one of *analysis, redo, undo*) of ARIES satisfies.

1. After a crash, $\mathcal{L} = \phi$

2. $\mathit{post}(\mathit{analysis}) \Rightarrow$
   $\forall p \forall ob \forall t(((p_t[ob] \in \mathcal{SL} \wedge \mathit{commit}(p_t[ob]) \notin \mathcal{SL}) \Leftrightarrow p_t[ob] \in \mathit{Losers}))$

3. $\mathit{post}(\mathit{analysis}) \Rightarrow$
   $\forall p \forall ob \forall t(p_t^R[ob] \in \mathcal{SL} \Leftrightarrow p_t^R[ob] \in \mathit{Losers})$

4. $\mathit{post}(\mathit{analysis}) \Rightarrow$
   $\forall p \forall ob \forall t((p_t[ob] \in \mathcal{SL} \wedge \mathit{commit}(p_t[ob]) \in \mathcal{SL}) \Leftrightarrow p_t[ob] \in \mathit{Winners})$

5. $\mathit{post}(\mathit{redo}) \Rightarrow (\mathcal{L} = \mathcal{SL})$

6. $\mathit{post}(\mathit{undo}) \Rightarrow$
   $\forall p \forall ob((p[ob] \in \mathit{Losers}) \Rightarrow$
   $(\mathit{undo}^R(p[ob]) \in \mathcal{L}) \wedge \forall q \forall ob(q[ob] \rightarrow_{\mathcal{L}} p[ob] \Rightarrow$
   $\mathit{undo}^R(p[ob]) \rightarrow_{\mathcal{L}} \mathit{undo}^R(q[ob])))$

   Here $p[ob]$ and $q[ob]$ indicate operations that may be done by a transaction or the system.

7. $\forall p \forall t \forall ob(\mathit{undo}^R[p_t[ob]] \Leftrightarrow \mathit{abort}^R[p_t[ob]])$

8. $\mathit{post}(\mathit{undo}) \Rightarrow \mathcal{L}^{\nabla\rceil\lceil\wr} \in \mathit{prefix}(\mathcal{L})$

9. ARIES is not active outside the recovery period.

Formalizing ARIES/RH. Because our framework is operation-based and not transaction-based, extending the formalization (preceding) and the proofs (following) for ARIES to ARIES/RH only entails reasoning about chains of delegations, represented by scopes.

8.4.3 Proof Sketches

With the logging and commit/abort protocols and the recovery rules from Section 8.3, we show examples of proving that ARIES specifications conform to the specification of recovery protocols.

- ARIES Specification 9 can be used to show that the recovery Specification 5 holds.
- As a more involved example, LOG-CT ensures that at a crash, all committed operations are indeed in $\mathcal{SL}$. From ARIES Specifications 2, 3 and 6, we can infer that all uncommitted transaction operations and recovery system operations are undone. Further, these are the only operations that are undone. Recovery system operations include undos of aborted operations. Hence, operations that are to be aborted are also undone. Further these operations are undone in an order consistent with ARIES Specification 6. Hence, we can infer Recovery Rule 1.

Proving that an implementation of the ARIES protocol satisfies ARIES specifications involves:

1. modeling the dirty page table, the transaction table, checkpoints, and different types of LSNs.
2. expressing the requirements stated above in terms of the properties of these entities with respect to the transaction management events and object events (i.e., during normal transaction processing) as well as during recovery steps.
3. given the pseudo-code that provides the details of transaction processing in terms of these concrete entities, demonstrating that the correctness requirements on these entities in fact hold.

8.5 FURTHER WORK AND SUMMARY

We showed how our recovery framework can be used to deal with the basic recovery methods for atomic transactions that work in conjunction with in-place updates, the Write-Ahead Logging (WAL) protocol and the no-force/steal buffer management policies. Also, for ease of exposition, we assumed that recovery processing was completed before new transactions were allowed. We also showed how to add delegation, and how the specifications and implementations were modified.

The building blocks developed in Section 2, namely, histories, their projections, and the properties of the (resulting) histories are sufficient to deal with situations where these and other assumptions are relaxed, suggesting further work.

Beyond in-place updates. Some recovery protocols are based on the presence of shadows in volatile storage. Updates are done only to shadows. If a transaction commits, changes made to the shadow are installed in the stable database. If it aborts, the shadow is discarded. To achieve this each object *ob* in such an environment is annotated by its version number ob^1, ob^2,..ob^n where each version is associated with a particular transaction. When intention lists are used, some protocols make use of intention lists whereby operations are explicitly performed only when a transaction commits. The properties of these protocols can be stated by defining projections of history $\mathcal{H}$ for each active transaction along with a projection with respect to committed transactions.

Considering object to page mapping issues. The model of Section 2 assumed that the object was both the unit of operation as well as the unit of disk persistence. In general, multiple objects may lie in a page or multiple pages may be needed to store an object. To model this, one more level of refinement must be introduced: the operations on objects mapping to operations on pages.

Reducing delays due to crash recovery. Checkpointing is used in practice to minimize the amount of redo during recovery. We can model checkpoints as a projection of the history $S\mathcal{L}$ and, using that, redefine the requirements of the redo part of the protocol. Some protocols allow new transactions to begin before crash recovery is complete. After the transactions that need to be aborted have been identified and the redo phase is completed, new transaction processing can begin. However, objects with operations whose abortions are still outstanding cannot be accessed until such abortions are done. This can be modeled by unraveling the recovery process further to model the recovery of individual objects and and by placing constraints on operation executions.

Avoiding unnecessary abortions. In a multiple node database system, the recovery protocol must be designed to abort only the transactions running on a failed node [Molesky and Ramamritham, 1995]. This implies that not all transactions that have not yet committed need be aborted. To model this, the crash of the system must be refined to model crash of individual nodes and the recovery requirement as well as the protocols must be specified in a way that only the transactions running on the crashed nodes are aborted.

Summary

We have used histories, the mainstay of formal models underlying concurrent systems, as the starting point of our framework to deal with recovery. The novelty of our work lies in the definition of different categories of histories, different with respect to the transaction processing entities that the events in a history pertain to. The histories are related to each other via specific projections. Correctness properties, properties of recovery policies, protocols, and mechanisms were stated in terms of the properties of these histories. For instance, the properties of the transaction management events and recovery events were specified as constraints on the relevant histories. The result then is an axiomatic specification of recovery. We also gave a sketch of how the correctness of these properties can be shown relative to the properties satisfied by less abstract entities. Further, we showed how to extend the framework and prove correctness when we include delegation, whose semantics allows the construction of advanced transaction models. We concluded discussing the directions in which to proceed to broaden the scope of our work.

Notes

1. This is affected by both concurrency control policies and recovery policies.

2. Formally, $\mathcal{H}^{\varepsilon \mathcal{H}^{\epsilon}} = \mathcal{H}^{\varepsilon -} \circ \varepsilon$ where $\circ$ is the usual composition operator.

3. $\mathcal{H}^{(ob)} = p_1[ob] \circ p_2[ob] \circ \ldots \circ p_n[ob]$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations.

4. Notice that $\mathcal{H}^{(ob)} = \mathcal{H}_P$ and $\mathcal{H}^{(ob)-Rp \cup Ap} = \mathcal{H}_L$.

5. This is one of the reasons we prefer to have ways by which the commitment of an operation can be dealt with in addition to the commitment of transactions. Furthermore, we desire a formalism that can uniformly deal with recovery in advanced transaction models (where a transaction may be able to commit even if some of its operations do not).

6. Some variants of ARIES merge the two forward passes into one, thus we also use only one forward pass.

7. To avoid undoing an operation repeatedly should crashes occur during recovery.

8. Notation *delegate*(t_1, t_2, ob) indicates delegation of *all* operation of t_1 on *ob* to t_2.

9. For each transaction *t*, *Tr_List*(t) contains the head of the BC(t), e.g., in fig. 8.4, BC(t) is *Tr_List*(t).

10. For example, this can occur in the case of non-conflicting operations, such as increments of a counter.

11. To reduce clutter, '_' denotes a field that we do not change or are not interested in.

12. In ARIES, all loser operations are those invoked by loser transactions, so ARIES/RH reduces to ARIES when there is no delegation.

V Transaction Optimization

9 TRANSACTION OPTIMIZATION TECHNIQUES

Abdelsalam Helal, Yoo-Sung Kim, Marian H. Nodine, Ahmed K. Elmagarmid and Abdelsalam A. Heddaya

Abstract: Replication introduces a tension between query optimization and remote access control in a distributed database system. If we view a transaction as a partially-ordered set of queries and updates, then factors that affect quorum selection for the fragments accessed by a transaction as a whole are currently orthogonal to factors that affect the replica selection during the planning of individual queries. Therefore, the two processes may act at cross-purposes to one another. Query optimization considers an individual query and selects a set of fragments that minimizes the computation and communication cost and allows computation to be pushed into the local site. Transaction management, on the other hand, selects quorums (sets of replicas to retrieve) based on replica availability and on mutual consistency constraints such as quorum intersection among write operations or between read and write operations. Thus, transaction optimization narrows the "optimal" solution space for the queries it contains. Hence, transaction management should cooperate with query optimization to optimize transaction processing.

In this book chapter, we discuss why and how to optimize transactions. We present a novel transaction optimization strategy that integrates query optimization techniques with transaction management. The proposed strategy chooses quorums of maximum intersection, while minimizing a communication and/or computation cost function. It also attempts to maximize the number of up-to-date copies of read quorums, so as to maximize the optimization space of the individual queries.

9.1 INTRODUCTION

Distributed query optimization and transaction management have typically been separated into independent modules [Helal et al., 1996b]. A query, written in a convenient non-procedural language such as the relational SQL or QUEL [Korth and Silberschatz, 1991], refers to logical elements of the distributed database, be they relations or views. The query is submitted first to a query optimizing compiler (QOC) that transforms it into a procedural program operating on physical data that constitute fragments of the logical relations. This procedural program is generated carefully so as to minimize the I/O, communication, and computation costs of executing the query. A traditional compiler then produces the final executable form of the query. The task of managing query execution falls on the shoulders of the transaction manager (TM), which constitutes part of the runtime system, and whose functionality is invoked via procedure calls inserted by the QOC. The TM ensures the ACID properties of the transaction by performing concurrency control, recovery, and atomic commitment.

In a replicated distributed database [Helal et al., 1996a], fragments of data are replicated across several databases to increase its availability in a failure-intolerant environment. Here, availability concerns dictate that many consistent replicas be available at many sites. When a transaction accesses a set of fragments, efficiency concerns dictate that those accesses be clustered in as small a set of sites as possible. This process is simplest when there are few replicas to consider. Thus, the current goals of transaction management and replication conflict.

Distributed transactions that access replicated data may suffer from variable processing delays. Depending on the degree of replication and on the data fragmentation strategy, a transaction may have to access a large number of physical fragments. This leads to

1. an increased number of round-trip messages,
2. an increase in the total message CPU processing time (for send and receive at the transaction initiation site), and
3. an increase in the number of execution threads used to spawn off multiple concurrent communication.

We refer to each execution thread used to control a physical fragment as a *shred*, and the processing overhead due to a transaction's shreds as the *shredding effect*. Physical fragments are also referred to as shreds in this paper. While facilitating higher degree of concurrency and availability, fine-grain fragmentation and large-scale replication are responsible for the shredding effect.

The challenge that we address in this book chapter is how to alleviate the shredding effect while maintaining the same high levels of concurrency and availability. We view transactions as partially-ordered sets of queries and/or updates. We argue that in a replicated, distributed database and with transactions of this form, the QOC and the TM must communicate to ensure efficient transaction execution. We introduce a component of the QOC called the Transaction Optimizer (TO), that interacts with the TM to select the best set of replicas to access during the processing of a specific transaction. This allows the two modules to work together to ensure the transactional cognizant optimality of the query execution. That is, it chooses an execution that minimizes the communication and/or computation overhead of a transaction based on its overall collection of queries and updates (transactional) and a knowledge of current site availability (cognizant).

9.1.1 What is Wrong with the Current Architecture?

Architecturally, the split between QOC and TM handicaps the QOC dramatically, in that the QOC cannot take advantage of the run-time information about failure and performance that is readily available to the TM. This is especially true in the presence of replication in the distributed system. For example, since the QOC knows nothing about the availability of specific replicas, it may choose to access replicas that are unavailable due to site or communication failures. The QOC may also assume that the set of replicas are all up-to-date while in fact a replica or two could be lagging behind. When the TM discovers this information at runtime, it cannot invoke the QOC to revise the scope of its optimization space.

Another example of this handicap arises from the ability of the transaction program to execute independent queries and updates under a single transaction. The precise choice of the queries to be so composed may not occur until runtime, for instance when the transaction program is allowed the power of conditional branching. Because the QOC optimizes each query separately, it may very well choose for different queries to touch physical copies whose union is strewn across much of the distributed system. We call each physical copy touched by a transaction a *shred.* Each shred requires a separate thread of control, thus a transaction with many shreds on different sites incurs a higher communication and computation overhead.

This shredding effect in replicated distributed database systems introduces another reason for transaction optimization. In a distributed system, transaction management imposes a finer granularity on the transaction queries, resulting in more constituent parts for the transaction. This leads to the shredding of the transaction as a single control unit into multiple threads of costly controls. Our goal in this book chapter is to show how this effect can be mitigated by selecting quorums to cluster accesses to the same database and thus maximize our ability to piggyback shred control messages together.

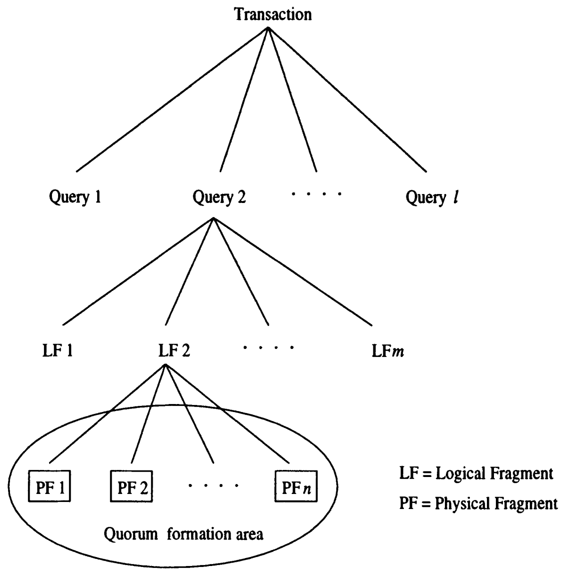


Figure 9.1 Transaction composition

Before discussing the implication of this effect, we give an example. Figure 9.1 shows a transaction which has several queries. Each query accesses several data items. A read set R and a write set W are imposed on each transaction. For example, a single query transaction, $A \bowtie B$ creates $R = \{A, B\}$, and $W = \emptyset$. Each element in $R \cup W$ is called a *logical fragment*. For each logical fragment, a data dictionary is accessed to obtain the corresponding set of *physical fragments*. Physical fragments could be disjoint entities of a larger data object, or identical replicas of the same data object. In this book chapter, we only consider physical fragments as identical replicas of logical fragment.

Assume that a global transaction has l queries and each query accesses m logical fragments, on the average. If each logical fragment access requires a quorum of n physical fragments, the global transaction is actually transformed into $l \times m \times n$ parts. If all the physical fragments are stored at different sites, the transaction manager may have to contact n sites up to m times for each query.

Now consider the issue of optimizing the individual queries. During the query decomposition and optimization phase, each query will be considered separately, and sites chosen for each quorum will be based on optimal placement with respect to efficient query execution and pushing subquery execution down to the local sites. The query optimizer does not consider that a replica it needs may be accessed from some other site because it was read by some other query in the transaction, and consequently may select a different replica. Alternatively, it may be desirable to retrieve a replica from a site that is already

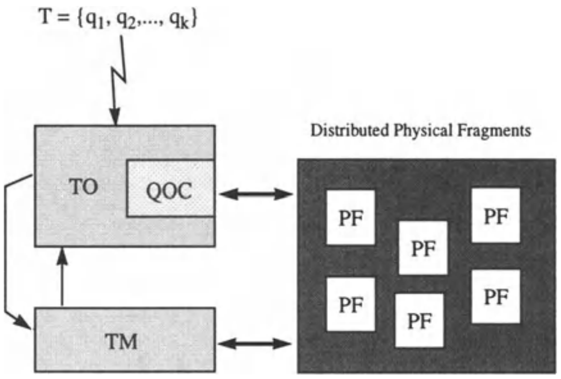

$T = \{q_1, q_2, \ldots, q_k\}$
Distributed Physical Fragments
TO
QOC
TM
PF
PF
PF
PF
PF
PF

Figure 9.2 Query optimizer – transaction manager interaction

being accessed by the transaction, but the query optimizer does not have the information to do this.

For example, consider a transaction T_1 that contains five queries, each of which accesses four logical fragments, where each logical fragment requires a read quorum of five physical replicas. If the replicas are distributed across hundreds of sites, then this transaction has 100 shreds. With improper quorum selection, the transaction manager may access up to 100 different sites during processing of our example query. Each additional site accessed by a transaction increases the processing overhead for ACID protocols such as two-phase commit. These protocols are communication-intensive in distributed database systems. If we can cluster the shred accesses for each transaction, accessing physical fragments that are available at the same site together, we can minimize the effects of transaction shredding and optimize the total transaction processing cost.

Optimizing the number of communications for shred control in a transaction requires knowledge of transaction processing protocols. Minimizing the number of shreds of a transaction is a constrained problem. In the transaction composition of Figure 9.1, given that all physical fragments are replicas, a quorum consisting of a known minimum number of fragments must be included [Ozsu and Valduiez, 1991, Gifford, 1979, Thomas, 1979]. Improper quorum selection by the transaction manager may limit the success of our optimization against the shredding effect.

9.1.2 How Should We Change the Architecture?

In Figure 9.2, we propose a new architecture, in which the QOC and the TM interact. This architecture introduces a new module to the QOC, called the Transaction Optimizer (TO), that interacts with the TM to access site availability information.

Based on this architecture, we introduce a novel transaction optimization strategy for minimizing communication and execution cost of the queries of the same transaction, and for minimizing the shredding effect caused by transaction management in replicated distributed database systems. The proposed transaction optimization strategy consists of two phases of interaction between the QOC and the TM. The first phase, called the pre-access phase, is a static phase where some decisions are made prior to any remote communication. In this phase, the QOC indicates to the TM which logical fragments it needs for each query in the transaction. The TM maps the logical fragments onto a set of physical fragments, and selects the candidate sites for all queries of the transaction in a way that maximizes the transaction optimization space. The TM then responds with availability and consistency information to the TO. The pre-access optimization applies ACID restrictions in the choice of the sites. This for example could include ensuring that the quorums formed are sufficiently large.

In the second phase, or post-access optimization phase, the TO uses the availability and consistency information from the TM to rewrite the queries and to design an efficient overall execution plan. This phase starts after initial communication is attempted with the remote sites initially included by the optimization process. All physical fragments that were selected during the pre-access phase and are currently available are locked, and at least one physical fragment for each logical fragment is up-to-date. The transaction optimizer generates the execution plan for each of its queries based on the optimization space determined in the pre-access phase, including the discovered realities of data availability, accessibility, and most importantly, up-to-dateness.

Using the proposed two-phase optimization technique, the fine-grain decomposition effect of transaction processing (the shredding effect) can be controlled by a subsequent reversal composition step that groups together shredded elements belonging to different queries but destined to the same remote sites. As a result of using our transaction optimization technique, the total processing cost of transaction processing in replicated, distributed database systems can be minimized in the presence of concurrency control, replication control, and recovery control constraints.

9.1.3 Chapter Organization

The chapter is organized as follows. The rest of the introduction is devoted to a discussion of related work. Section 9.2 defines the problem and describes examples that motivate the need of transaction optimization. We specialize our discussion on the effects of transaction optimization in replicated distributed database systems. In Section 9.3, we describe the basic ideas behind our transaction optimization strategy and give a detailed description of our two-phase optimization algorithms. We also present examples that clarify the interaction between the transaction manager and the query optimizing compiler. Section 9.4 discusses the different ways the QOC can use information provided by the TM to globally optimize individual queries. Finally, conclusion of this work is given in Section 9.5.

9.1.4 Related Work

To our knowledge, the only work that relates to ours was proposed by Mohan in [Mohan, 1992]. Like this work, Mohan argued for the importance of the cooperation between transaction management and query optimization. Unlike our work that focuses on distributed transaction management including replication [Helal et al., 1996a], Mohan focused on concurrency control, where locking information was made available to the query optimizer, so that the latter makes intelligent decisions. For example, at some instances, locking was avoided by taking advantage of the isolation level of the optimized query in execution. A major difference between our optimization and Mohan's is that his comes from sacrificing the isolation property of transactions. In our optimization, ACID properties [Gray, 1981, Gray and Reuter, 1993] of transactions are maintained. In another less relevant work, Samar [Samaras et al., 1995] proposed a two-phase commit optimization in commercial distributed environments.

The classic survey paper for relational query optimization is [Jarke and Koch, 1984]. Also, a condensed survey and a good bibliographical resource on the query optimization problem is given in [Ioannidis, 1996].

9.2 PROBLEM DEFINITION

As evident in the example given in Figure 9.1, if the shredding effect of transaction management goes unoptimized, processing and communication resources will be wasted due to the repeated overhead associated with every shredded element. In general, the transaction manager should optimize transactions to choose quorums so as to cluster accesses to physical fragments that can be made from the same site.

Quorum selection heuristics can be used in conjunction with a particular set of optimization objectives. These objectives could be

1. to minimize the number of communication messages,

2. to balance the load and reduce any skewed access in the system, or

3. to minimize response time even at the expense of broadcast messages.

Each one of these objectives impacts how quorums are selected. However, quorum selection is constrained also by the operational definition of the quorums themselves (the quorum intersection rules [Thomas, 1979, Gifford, 1979]).

The following example demonstrates an optimization that can be done during quorum selection. Assume that a distributed database system consists of seven sites, where S_i stands for site i ($1 \leq i \leq 7$). Logical data items A, B, C, have 5, 4, and 3 physical fragments, respectively. The physical fragments (replicas) of A are stored at S_2, S_3, S_5, S_6, and S_7. Those of B are stored at S_1, S_3, S_4, and S_5, and those of C are stored at S_4, S_5, and S_6. Assume that the majority-consensus protocol is used for replica control. Further, assume that we have a transaction T_1, consisting of queries that access logical fragments A, B, and C. According to quorum parameters (weight of fragments and read/write thresholds), T_1 must access at least three-copies quorum for A, two-copies quorum for B, and one-copy quorum for C. These assumptions are summarized in Table 9.1.

Assume the replicas of A at S_2, S_6, and S_7, and those of B at S_1 and S_3, and those of C at S_4 are selected for T_1 as the candidate sites for the formation of the required quorums. The transaction manager of T_1 then must contact a total of six remote sites to form the required quorums. Using different replicas, quorum formation of A could include copies at S_3, S_5, and S_6, and that of B could include copies at S_3 and S_5, and that of C could include copies at S_5. Compared to six sites, this selection of quorums results in communication with only three sites. The selection of quorums therefore affects the size of the set of spanning sites. Because this optimization concerns all of the quorums that must be chosen for all queries in T_1, we call this process of minimizing the number of sites accessed *transaction optimization.*

Table 9.1 Allocation table of replicas and required quorums for T_1.

Sites \ DB items	S_1	S_2	S_3	S_4	S_5	S_6	S_7	size of required quorum
A		√	√		√	√	√	3
B	√		√	√	√			2
C				√	√	√		1
1st Quorum	B	A	B	C		A	A	6 sites
2nd Quorum	—	—	A,B	—	A,B,C	A	—	3 sites

The goal of the transaction optimization process is as follows: Each transaction accesses a set of logical fragments Θ_i, $1 \leq i \leq m$ during the course of executing its queries and updates. Let $d(\Theta_i)$ be the set of sites which contain physical replicas for logical fragment Θ_i. Then let the subset of these that are available be represented by the function $A(d(\Theta_i))$. If the transaction requires a quorum of l replicas, then let $(Q(A(d(\Theta_i))))$ be the set of all sets of size l containing fragments from sites $A(d(\Theta_i)))$. This is the set of all available quorums (by site). Then, for each logical fragment i, we want to select one of these quorums, $q_i = q(Q(A(d(\Theta_i))))$ in such a way that minimizes the size of

$$\bigcup_{i=1}^{m} q_i.$$

If several such options are available, we can arbitrate by choosing the quorums that are best from the standpoint of optimizing the individual queries.

In the following section, we present our transaction optimization strategy that aims at minimizing the transaction shredding effect, and that also minimizes the set of spanning sites . We show how the transaction manager cooperates with the transaction optimizer to achieve this optimization while maintaining the consistency of the data.

9.3 A NOVEL TRANSACTION OPTIMIZATION STRATEGY

In this section, we describe a new transaction optimization strategy that is followed for every operation on logical fragments. For each logical fragment, the pre-access phase attempts to lock a superset of its quorum (define this as a *quorum superset*) that will best suit the set of queries and updates that access it. This may involve giving less preference to copies with high communication cost, choosing a subset of the available replicas, or preferring a superset that has the highest affinity with other quorum supersets and extends the spanning set of sites accessed by the transaction as little as possible. Once all quorum supersets of the same transaction are decided, piggyback messages are sent to the remote sites. These messages attempt to access the version information for each copy, locking the fragment appropriately for use by the queries and updates in the transaction and returning its version information. Note that at least one of the replicas of each logical fragment accessed will be current. Also, the TM will detect if no message is returned in a reasonable time, and mark that site as being unavailable.

Based on the replies, the post-access optimization proceeds. The replies indicate the up-to-dateness (or the degree of mutual consistency) among the copies. If all copies are up-to-date, better optimization will be possible. On the other hand, if only one copy is up-to-date, optimization becomes limited, and the cost of execution-time copying of the most up-to-date copy to other replicas should be weighed against the loss of parallelism. The post-access phase first weighs the relative merits of accessing the different fragments at each site, factoring up-to-dateness, and retrieval cost for all available fragments into a single cost metric that it uses for comparison. The result of this evaluation is a prioritized list of sites to use when accessing the physical fragments in the query.

Once the prioritized list is generated, the post-access phase sets up one or more waves of piggyback messages (depending on the structure of the transaction). First, it processes the queries, then the updates. For each query, it isolates out the monodatabase subqueries, taking into account the cost information from the pre-access phase and the prioritized list. It adds each monodatabase subquery to the piggyback message to the site it must execute on. It then analyzes the updates together. Given an update requiring a write quorum of q, it piggybacks updates onto the messages for at least q sites, favoring sites that already have messages being sent to them, and sites where multiple updates can be executed. As with the pre-access phase, "extra" updates may be generated if piggyback messages are being sent to more than q of the logical fragment's sites, as these updates can be done easily and will increase the consistency of the data.

In the following, we give the details of the pre-access and the post-access optimization phases.

9.3.1 Pre-Access Optimization

The first phase (pre-access phase) of our transaction optimization attempts to select a set of likely sites to fulfill the quorum requirements for each logical fragment. It attempts to lock all physical fragments relevant to the transaction for reading or writing as needed, and returns the version and size information for each such physical fragment. The choice of sites to access is made with the following objectives in mind:

1. Minimize the size of the spanning set (number of remote sites that must be contacted on behalf of the transaction).
2. Minimize the cost of the spanning set (communication and computation cost function) of the remote sites.
3. Minimize the total number of messages exchanged on behalf of the transaction.
4. Maximize (at minimal or no cost) the ability to attain mutual consistency among the copies of the logical fragments accessed by the transaction given the above constraints. This optimization supports the subsequent post-access optimizations that will be discussed in Section 9.3.2.

The pre-access phase of the optimization is implemented by the Algorithm 1 whose details are shown in Figure 9.3. This algorithm takes as its input an AllocationTable like the one shown in Table 9.2. In this table, for each LF_i, RemainingQuorum (shown in Algorithm 1 but not in Table 9.2) is the number of copies yet to be selected for the read/write quorum. It is initialized to the required number of copies for a quorum (possibly with one or more extra copies if the network is failure-prone). For each S_k, EffectiveReplicas is the number of physical fragments at the site whose quorums have yet to be filled. It is initialized to the number of logical fragments it has available, and RoundTripTime is set to the last known round-trip-time to the site S_k.

During computation, Algorithm 1 computes the RemainingCost as follows:

Algorithm 1
Pre-Access Optimization (AllocationTable) returns PiggybackMsgSet.
Input:
1. Set of logical fragments $\{LF_i, 1 \leq i \leq m\}$.

Output:
1. ResultSet, a minimum cost spanning set over all LF_i.
   Resultset is the union of all selected quorums for LF_i.
2. A set of piggyback messages 1-to-1 mapped to ResultSet.

```
ResultSet = $\varnothing$.
WHILE some RemainingQuorum($LF_j$) in ($1 \leq j \leq m$) $\neq 0$ DO
   Select $S_k$ with minimum RemainingCost.
      If a tie, favor a $S_k$ already in ResultSet.
   ResultSet = ResultSet $\cup$ $\{S_k\}$
   FOR EACH $LF_i$ that has a replica at $S_k$ DO
      Mark the entry of $S_k$'s column of $LF_i$'s row.
      RemainingQuorum ($LF_i$) -= 1.
      IF (RemainingQuorum ($LF_i$) = 0) DO
         FOR EACH column $S_l$ that has a replica of $LF_i$
            EffectiveReplicas ($S_l$) -= 1.
      END                // if size of RemainingQuorum becomes 0
   END                                        // for each $LF_i$ at $S_k$
END   // while there is a ReminingQuorum somewhere to select
                  // Piggyback requests to get one message per site
FOR EACH $S_k$ in ResultSet DO
   Piggyback all marked entries of $S_k$ into one message.
   FOR EACH unmarked entries of $S_k$ for some $LF_m$
      Append $LF_m$ to $S_k$'s piggyback message.
END                                              // piggback loop
RETURN set of piggyback messages.
END
```

Figure 9.3 Pre-access optimization algorithm

Table 9.2 Example of allocation table input to pre-access algorithm.

Sites \ DB items	S_1	S_2	S_3	S_4	S_5	S_6	S_7	size of required quorum
A		$\surd$	$\surd$*		$\surd$*	$\surd$*	$\surd$	3
B	$\surd$		$\surd$*	$\surd$	$\surd$*			2
C				$\surd$	$\surd$*	$\surd$**		1
Effective Replicas	1	1	2	2	3	2	1	
Round Trip Time	30	30	30	30	30	30	30	

$$\text{RemainingCost}(S_k) = \frac{\text{RoundTripTime}(S_k)}{\text{EffectiveReplicas}(S_k)}.$$

This cost metric favors sites with a short round trip time (as remembered from previous accesses to the site) and a large number of physical fragments relevant to the transaction. When no relevant copies are left at a site, RemainingCost increases to ∞.

Note that with the input given by Table 9.2, the quorum supersets selected by Algorithm 1 would be, for logical fragment A, sites S_3, S_5 and S_6, for logical fragment B, sites S_3 and S_5, and for logical fragment C, sites S_5 and "extra" site S_6. Those marked in the table with a "*" are selected during the execution of the first *while* loop, while those marked with a "**" are piggybacked onto the message during the second step of the bottom *for* loop.

The output of Algorithm 1 is a set of physical fragment names, grouped by site. At this point, piggybacked request messages are sent to each site to lock each fragment according to the type of access, and return version and size information for each locked fragment. Round trip time is also returned for future pre-access optimizations.

To fulfill its first objective, Algorithm 1 chooses quorums by favoring the ones that largely intersect with the union of all quorums of the same transaction. Choosing quorums this way maximizes the affinity results in a minimal size spanning set. To fulfill its second objective, Algorithm 1 attempts to build the minimal spanning set under the additional constraint of minimizing the cost function associated with accessing the data. The cost function could take into account communication cost, processing cost, and other factors.

To fulfill its third objective, Algorithm 1 uses a piggybacking technique where different quorum elements destined to the same remote site are grouped together in a single piggyback message. The intuition behind this is to replace multiple communication exchanges between two end points by exactly one round trip of communication exchange. This optimization sets the upper bound on the number of requests that can be induced by a single global transaction to be linear in the number of database sites, regardless of the size or number of queries in the transaction. Piggybacking therefore counters the communication effect of transaction shredding.

To fulfill its fourth objective, Algorithm 1 attempts to extend the set of replicas locked during pre-access to include extra copies at no additional communication cost. This is because the additional replicas are co-located with physical fragments that are already being accessed, and can be locked using the same piggyback message. The entry marked "**" in Table 9.2 is an example of such a replica.

The locking during the pre-access phase of additional copies at the sites where the piggyback messages are sent helps the optimization process during the post-access phase both for queries and updates. For queries, which do only reads, requesting more copies than needed provides higher availability. For example, having more copies available gives the query optimizer more latitude in selecting copies, giving it more opportunity to select copies from the same site as inputs to the same operation and then pushing the operation down into the site. It is also useful to have extra copies in case one of the sites fails or becomes inaccessible, or to improve response time by using the first read quorum that materializes. In the case of a write operation, piggybacking can be used to bring more copies into sync. This leads to a higher likelihood that more elements of subsequent read quorums are up-to-date. This, in turn, leads to more parallelism that can be exploited by the QOC.

9.3.2 Post-Access Optimization

Once a transaction manager at a remote site receives a piggyback message, the piggyback is unpacked and the quorum elements are decoded, and processed individually. We skip concurrency control details (see [Ozsu and Valduiez, 1991, Bernstein and Goodman, 1984, Ceri and Pelagatti, 1984]), and continue from the point where the home transaction manager (where the transaction originated) receives a reply from one of the piggyback messages. At this point, access has already been made to physical data fragments and they are locked. The reply message consists of the acknowledgment of the request and a confirmation that the fragment is locked. In addition, a timestamp (or version numver) and size for each physical fragment is included (only the size of the most up-to-date fragment is important). When the home transaction manager receives all the timestamps of a quorum, it decides on the up-to-dateness and the size of the fragments. It constructs an AllocationTable similar to the one shown in Table 9.3 before it begins the post-access optimization phase, in which the actual execution of the queries of the transaction begins. The "*" marks essential replicas while the "**" marks extra replicas as discussed in Algorithm 1. The • marks up-to-date copies while the ∘ marks copies that are lagging behind. Algorithm 2 whose details are shown in Figure 9.4 gives the details of the post-access optimization phase.

Table 9.3 Example of allocation table input to post-access algorithm.

Sites \ DB items	S_3	S_5	S_6	up-to-date fragment size	size of required quorum
A	$\bullet\surd*$	$\circ\surd*$	$\bullet\surd*$	12K	3
B	$\circ\surd*$	$\bullet\surd*$		8K	2
C		$\bullet\surd*$	$\circ\surd**$	18K	1
Effective Replicas	2	3	2		
Round Trip Time	30	30	30		

The objectives of the post-access phase are as follows:

1. Ensure that accesses by different queries to the same logical fragment by the same transaction also access the same physical fragment, except when query operations are pushed down into the individual databases.

2. Reduce the number of sites accessed, under replication.

3. Reduce the communication cost.

4. Optimize and decompose each query.

In order to minimize the communication cost associated with executing a query, the post-access phase specifies an *effective retrieval cost* for each site, and selects quorums to minimize the total effective retrieval cost:

$$\begin{aligned}\text{EffectiveRetCost}(S_i) &= \text{RoundTripTime} + \\ &\quad (\text{ByteTransferTime} * \sum_{\text{EffectiveReplicas}(S_i)} \text{ReplicaSize})\end{aligned}$$

We perceive the actual queries and updates in a transaction to take place in waves, where the queries and updates for each wave are piggybacked together to the same site as much as possible, and the number of waves is minimized. For each wave, the piggybacked messages to each site are generated as shown in Algorithm 3 whose details are shown in Figure 9.5.

The post-access phase takes a heuristic approach towards accomplishing its objectives. It works towards ensuring that accesses by different queries to the same logical fragment by the same transaction also access the same physical fragment as much as possible by always using the order SitePerm when assigning operations to sites. A prefix of SitePerm contains the sites in the actual spanning set for the transaction. No site outside of the spanning set will be selected for access unless some query has a very efficient plan that requires the use of that site. In this case, updates may also be propagated to that site. Barring this circumstance, all accesses will be focused on the sites in the spanning

```
Algorithm 2
Post-Access Optimization (AllocationTable) returns SitePerm.
Input:
    1. Minimum cost spanning set from Algorithm 1.
    2. Up-to-dateness information for all copies accessed
            during the pre-access phase.
    3. Cost parameters measured during the pre-access phase.
Output:
    1. SitePerm:  A Prioritized list of sites in the spanning set.
    2. AllocTable:  AllocationTable with cost information added.


  /* Construct an Allocation Table */
  Make AllocTable [num replicas × spanning set size].
  FOR EACH site DO
     EffectiveReplicas = num physical fragments in the spanning set
                 accessed by the transaction.
     RoundTripTime = value measured in the pre-access phase.
  END
  FOR EACH logical fragment, ReplicaSize = value retrieved
                 in the pre-access phase.
  FOR EACH physical fragment,  Up-to-dateness = value retrieved
in
                 pre-access phase.
  /* Make sure you have available quorums for logical fragments */
  SitePerm = ⟨⟩.
  WHILE some RemainingQuorum($LF_j$) in ($1 \leq j \leq m$) $\neq 0$ DO
     Compute the EffectiveRetCost of each column.
     Select $S_k$ with minimum EffectiveRetCost.
     SitePerm = ⟨ SitePerm, {$S_k$} ⟩.
     FOR EACH $LF_i$ that has a replica at $S_k$ DO
        Mark the entry of $S_k$'s column of $LF_i$'s row.
        RemainingQuorum ($LF_i$) -= 1.
        IF (RemainingQuorum ($LF_i$) = 0) DO
           FOR EACH column $S_l$ that has a replica of $LF_i$
              EffectiveReplicas ($S_l$) -= 1.
        END              // if size of RemainingQuorum becomes 0
     END                                   // for each $LF_i$ at $S_k$
  END    // while there is a ReminingQuorum somewhere to select
     RETURN SitePerm, AllocTable.
END
```

Figure 9.4 Post-access site prioritization algorithm

```
Algorithm 3
    Post-Access Message Piggybacking (SitePerm,AllocationTable)
                    returns PiggybackMsgSet.
Input:
    1. SitePerm from Algorithm 2.
    2. Allocation table from Algorithm 2.
Output:
    1. PiggybackMsgSet: One message per site.

  PiggybackMsgSet = ∅.
  //PiggybackMsgSet is a set of the ordered pairs $\{< S_k, msgs >\}$,
                    where $S_k$ is the site for which $\{msgs\}$ are sent
  FOR EACH query $q_i$ in the current wave DO
    NewPiggybackMsgs = QueryOpt($q_i$, SitePerm, AllocationTable);
          //QueryOpt functionality is discussed in the next section.
    PiggybackMsgSet = PiggybackMsgSet ∪ NewPiggybackMsgs.
    Coalesce messages in PiggybackMsgSet that go to the same site.
  END                                          // For each query

  FOR EACH update $u_j$ in the current wave DO
    UpdateQuorumCount = num replicas needed for a write quorum.
    FOR EACH site $S_k$ that already has a piggyback message DO
      If $S_k$ has a copy of the logical fragment $u_j$ updates DO
        Add an update message to $S_k$'s piggyback message.
        UpdateQuorumCount -= 1.
      END
    END
    IF UpdateQuorumCount $\leq$ 0, move to next update.
    ELSE FOR EACH site $S_l$ that does not already have a piggyback
            message, in SitePerm order, DO
      Add an update message to the piggyback message for $S_l$.
      UpdateQuorumCount -= 1.
      IF UpdateQuorumCount $\leq$ 0, move to the next update.
    END
  END                                          // For each update

  RETURN PiggybackMsgSet.
END
```

Figure 9.5 Post-access message piggybacking

set. Thus, it also accomplishes the objective of reducing the number of sites accessed, under replication.

The post-access phase reduces the communication cost by minimizing the number of sites accessed, and also by using the cost metric EffectiveRetCost in selecting the physical fragments to access during a specific wave. Given a reasonable query optimizer, the processing cost for decomposing and optimizing each query should also be reduced.

9.4 QUERY OPTIMIZATION ISSUES

9.4.1 Query Decomposition and Site Assignment

Query optimizers operate on individual queries, reordering their operations so that they can be executed more efficiently. In a distributed database setting, a query optimizer does the following:

1. The query optimizer pushes down *select* and *project* operations, and reorders *joins*, so that the query is expressed as a query over a set of results of monodatabase queries, with one monodatabase query per site;

2. It then reorders the query from the previous step such that all of the interdatabase joins are done in some optimal order; [Bodorik and Riordon, 1988] maintains that near optimal results are achieved by reordering the joins to minimize the total size of the partial results; and

3. Finally, it assigns the interdatabase operations to specific sites.

Query optimization in a distributed database without replication is already an NP-hard problem, due primarily to the complexity of reordering the interdatabase join operations. In a replicated, distributed database, the first step of collecting together operations on the same database into monodatabase subqueries becomes far more complex, as replicated input logical fragments need to be assigned sites to minimize the cost of processing the query. To do a complete job, the optimizer must consider all possible combinations of logical fragment/site assignments, optimizing the query for each assignment, and selecting the optimal one based on minimizing the expected cost. Clearly, good heuristics need to be found for this phase of the query optimization as well. However, as this book chapter focuses on transaction optimization, we will merely examine the ways in which the QOC can exploit the information provided by the TM.

Some heuristics that the QOC might use in assigning reads of logical fragments to sites include the following:

- Is there a copy in some site this query is already accessing?

- If there is not a copy in some site this query is already using, what is the most desirable site (from SitePerm, output of Algorithm 2) that has a copy of this logical fragment?
- For a copy selected according to one of the above criteria, is this copy up-to-date? If not, is it better to bring that copy up-to-date or use a different copy that is already up-to-date?

Note that answering some of these questions requires the use of metrics defined over the cost information gathered from the pre-access phase. The exact nature and use of these metrics is specific to the query optimizer. However, fragment size and round-trip-time are factors in determining how much it costs to bring a fragment up-to-date. The first access to a site should be penalized by the whole round-trip-time. Also, round-trip-time and fragment size are factors in the cost of retrieving a relation from a site. We expect these and other metrics to be useful during query optimization.

9.4.2 Interim Replication

Interim replication allows a query optimizer to identify where the up-to-date replicas for some logical fragment reside, and make a temporary copy of that fragment in a different site to expedite the processing of a query. For example, if $A \bowtie B$ is the original query, A and B are large relations, and the result is small, it could be worth the overhead to temporarily replicate B at the site where A resides, and perform the join operation locally at that site.

The idea of interim replication itself is a time optimization that belongs to the classical query optimization domain. However, interim replication may or may not be advisable for a given query. Since our pre-access phase ensures that at least one up-to-date copy of a logical fragment is locked, a query optimizer can consider interim replication as a possibility when optimizing a specific query, as the actual data is guaranteed to be available.

Additionally, though, we have shown how the process of updating "extra" copies in the post-access phase increases the number of up-to-date copies in the database. Therefore, under our optimization strategy, interim replication will be minimized.

9.5 CONCLUSIONS

Query optimization and transaction processing usually work against each other. In a replicated distributed database, performing query optimization of transactions consisting of sets of queries and updates requires cooperation between the query optimizer and the transaction manager. This is because physical fragments (replicas) are stored in several sites, and a global transaction has a large number of *shredded elements*, or threads of control for specific (groups of) physical fragments at different sites. If the number of shredded elements is not optimized, the communication cost for transaction processing may be wasted due to the duplication of processing in query optimization and communication, and the repetition of transaction processing protocols for all shred elements of the transaction.

In this book chapter, we introduced an architecture that allows the transaction manager and the query optimizer to cooperate to reduce the number of shredded elements. Specifically, we introduced a two-phase transaction optimization strategy that minimizes the number of remote sites (spanning set) involved in a transaction, and consequently the total number of messages required for transaction processing. We introduced quorum affinity and showed how to chose a set of quorums with maximum intersection (Algorithm 1). We also introduced piggybacking of requests concerning different quorum elements residing in the same remote site. This way multiple requests by the same transaction to different copies that reside in the same site are grouped into a single piggyback message, thus bounding the total number of messages that are generated by a transaction to be linear in the number of sites (n). Piggybacking is especially needed in replicated distributed database system to mitigate the *shredding effect*.

We have shown how and where the proposed transaction optimization strategy uses classical query optimization (in the second phase), and how it cooperates with transaction management (in the first phase) to achieve better optimization.

We also utilized piggybacking by inserting additional updates to additional copies that are not part of the quorums. Such insertion (or piggyback expansion) increases the degree of mutual consistency among the copies and in the same time incurs very little additional overhead. At any degree of mutual consistency, transaction execution is ACID, and one-copy serializability is guaranteed. The higher the degree of mutual consistency, the more likely it is that future transactions will be able to find a nearby, up-to-date read quorum, thus aiding future optimization efforts.

VI ECA Approach

VII OLTP / OLAP

10 AN EXTENSIBLE APPROACH TO REALIZING ADVANCED TRANSACTION MODELS

Eman Anwar, Sharma Chakravarthy
and Marissa Viveros

Abstract: Use of databases for non-traditional applications has prompted the development of an array of transaction models whose semantics vary from the traditional model, as well as from each other. The implementation details of most of the proposed models have been sketchy at best. Furthermore, current architectures of most DBMSs do not lend themselves to supporting more than one *built-in* transaction model. As a result, despite the presence of rich transaction models, applications cannot realize semantics other than that provided by the traditional transaction model.

In this paper, we propose a framework for supporting various transaction models in an *extensible* manner. We demonstrate how ECA (event-condition-action) rules, defined at the *system level* on significant operations of a transaction and/or data structures such as a lock table, allow the database implementor/customizer to support: i) currently proposed extended transaction models, and ii) newer transaction models as they become available. Most importantly, this framework allows one to customize transaction (or application) semantics in arbitrary ways using the same underlying mechanism. *Sentinel*, an active object-oriented database system developed at the University of Florida, has been used for implementing several extended transaction models.

10.1 INTRODUCTION

The suitability of the traditional transaction model (with ACID properties) for serving the requirements of business-oriented applications has been long established. However, as the use of database management systems (DBMSs) encompass newer and non-traditional applications, it is necessary to re-examine the appropriateness of the traditional transaction model. This reassessment reveals that the ACID properties are too restrictive and in some cases inadequate for serving the requirements of non-traditional applications. Transactions in traditional DBMSs are implicitly assumed to be *competing* for resources instead of *cooperating* for accomplishing a larger task. This fundamental assumption imposes restrictions on the use of the traditional transaction model for applications other than the one it was intended for. For example, a strong demand for cooperation exists in CAD and software engineering environments. The traditional transaction model prohibits any form of cooperation by requiring the isolation of uncommitted transaction results. As another example, in a workflow application, some of the (sub)tasks that deal with invoices may have to satisfy the ACID properties (on a small portion of the database) whereas other tasks may work on their own copy of the data objects and only require synchronization.

The current solution for meeting the diverse requirements of novel applications has been the proposal of advanced or extended transaction models such as nested transactions, Sagas, ConTract model, and Flex transactions [Moss, 1981, Garcia-Molina and Salem, 1987, Reuter, 1989, Elmagarmid et al., 1990]. These transaction models relax the ACID properties in various ways to better model the parallelism, consistency, and serializability requirements of non-traditional applications. Unfortunately, there is no universal transaction model that satisfies the requirements of all known classes of applications. Rather, each transaction model tends to be *application-specific*, i.e., serve the requirements of a *particular class* of applications. Consequently, since a DBMS typically supports *only one* transaction model, a DBMS can only serve the requirements of a particular class of applications. Therefore, it is critical that the solution to this problem aims at a framework which readily supports multiple transaction models, as well as support them on the same DBMS. Choice of a transaction model is usually based on application needs and is best made *at application development time*, not at database development/configuration time. This approach, if successful, will obviate the need for developing DBMSs suited for specific application classes. It is equally important to avoid hardwiring the semantics of all known transaction models (the kitchen-sink approach), as this increases runtime checking as well as the footprint of the transaction manager. In summary, there is a need for: i) a DBMS to be configured with different transaction models as needed by the application at run time, ii) a DBMS to support more than one transaction model at the same time, and iii) configuring/selecting a transaction model by the user. Below, we identify the goals of our research.

10.1.1 Goals

- Provide a uniform framework which allows for the specification and enforcement of various transaction models[1] (including the traditional model) in the *same* underlying DBMS. More importantly, many scenarios require the specification and enforcement of dependencies amongst transactions where these dependencies do not necessarily conform to any particular transaction model. Consequently, we aim at a general-purpose framework where it is possible to express and enforce *both* existing transaction models as well as arbitrary transaction semantics in the form of transaction dependencies. Transaction dependencies need to be supported to accommodate workflow applications.

- Survey of the literature reveals a general tendency to propose advanced transaction models at the conceptual level only without paying much attention to the actual implementation details in terms of data structures and primitive operations that are common to various transaction models. One of the main objectives of our approach has been to implement various transaction models to understand additional data structures/operations required for each transaction model as well as to provide a platform for analyzing the behavior of applications adhering to various transaction models. In essence, for various transaction models, the methodology proposed in this paper provides, at the implementation level, what ACTA [Chrysanthis and Ramamritham, 1990] provides at the conceptual level. In other words, our approach can be viewed as taking the conceptual formalism of transaction models provided by ACTA [Chrysanthis and Ramamritham, 1990] and transforming them to their operational form to gain insights into the implementation issues.

- Currently, a large number of transaction models as well as variants of existing models exist. This proliferation is partly due to the diverse requirements needed by different applications as well as researchers extending a transaction model to its conceptual limit. We aim at using the actual implementation of transaction models as a platform for understanding their similarities and differences. This in turn may identify relationships amongst transaction models such as one transaction model subsuming another. Consequently, it may be possible to reduce the number of transaction models which need to be considered.

- Understand the interactions of different transaction models. In other words, what are the semantics of running multiple concurrent applications each adhering to a different transaction model?

10.1.2 Related Work

Several alternative approaches to supporting various transaction models have been proposed by the research community. Some of these approaches have been incorporated into research prototypes although commercial DBMSs incorporate very few of these research results [Mohan, 1994].

- Carnot [Attie et al., 1992] has taken the approach of providing a general specification facility that enables the formalism of most of the proposed transaction models that can be stated in terms of dependencies amongst significant events in different subtransactions. CTL (Computational Tree Logic) is used for the specification and an actor based implementation has been used for implementing task dependencies.
- ASSET [Biliris et al., 1994] identifies a set of primitives using which a number of extended transaction models can be realized. Implementation of the primitives has been sketched.
- TSME [Georgakopoulos et al., 1994] provides a transaction processing system toolkit which allows for the specification and enforcement of transaction dependencies. Rules are used as a mechanism for enforcing these dependencies.
- [Barga and Pu, 1995] adopt a layered approach to realizing various transaction models. The notion of *adapters* are used to extend entities such as the lock manager with operations and data structures to support different transaction models.
- ACTA [Chrysanthis and Ramamritham, 1990] proposed a conceptual-level framework for specifying, analyzing, and synthesizing extended transaction models using dependencies.
- CORD [Heineman and Kaiser, 1997] proposed a DBMS architecture with a Concurrency Control Language (CCL) that allows a database application designer to specify concurrency control policies to tailor the behavior of a transaction manager. They have designed a rule-based CCL, called CORD, and have implemented a runtime engine that can be hooked up to a conventional transaction manager to implement the sophisticated concurrency control required by advanced database applications.
- A proposal for supporting advanced transaction models by extending current transaction monitors' capability [Mohan, 1994].

In this paper, a pragmatic solution is proposed by adapting the active database paradigm for modifying the system behavior (as opposed to the application behavior) using sets of ECA rules. The basic idea is to allow the database administrator (DBA) to *build* or equivalently *emulate* the desired transaction behavior by using ECA (event-condition-action) rules to either: i) modify the behavior of a transaction model supported by the system or ii) support different transaction models (including the traditional one) by providing rule sets on primitive data structures. Our approach differs from current approaches in that we use the active database paradigm as a mechanism for supporting extended transaction models in a novel way[2]. Our approach also models and enforces *auxiliary* semantics (other than those defining transaction semantics) useful for a number of applications within the same framework. For example, to reduce the possibility of rollbacks and unnecessary waits by transactions, it might be necessary to define semantics which specify thresholds on the number of long-lived transactions in the system.

This paper is structured as follows. Section 2 outlines our approach based on the active database paradigm as well as presenting details of several alternative ways for supporting extended transaction models in an active database environment. Section 3 describes Zeitgeist's transaction manager (an OODBMS developed at Texas Instruments) and how we incorporated active capability at the *systems level* into Zeitgeist producing Sentinel. The implementation details for realizing transaction models using this platform are given in Section 4. A discussion of the extensibility of our approach is presented in Section 5 while conclusions and future directions for research are given in Section 6.

10.2 OUR APPROACH

A transaction performs a number of operations during the course of its execution – some specified by the user and some performed by the system to guarantee certain properties. The *semantics* of the operations performed by the system differ from one transaction model to the other. For instance, the semantics of the *commit* operation in the traditional transaction model entails updating the log, making all updates permanent in the database, and releasing all locks held. This is in contrast to the commit of a subtransaction (in the nested transaction model) where all locks are inherited by the parent and the updates *not* made permanent until all superior transactions commit. As another example, a transaction in the traditional transaction model can acquire an *exclusive-lock* on an object if no other transaction holds *any* lock on that object. This is different from the nested transaction model where a subtransaction may acquire an *exclusive-lock* on an object even if one of its ancestor transactions holds a lock on that object. Moreover, some transactions perform operations which are very specific to that transaction model (and not shared by other transaction models). As an example, in the Split transaction model, a transaction may perform the operation *split* which causes the instantiation of a new top-level transaction and the delegation of some uncommitted operations to it.

It is apparent that in order to support different transaction models in the *same* DBMS, one should not hardwire the semantics of operations such as commit, abort, read and write.[3] Instead, a flexible mechanism is needed for associating computations with these system operations, as well as with some operations performed by the system on behalf of users, where these computations define the semantics of these operations *depending on the transaction model chosen for executing a specific application.* Furthermore, for this mechanism to be effective and extensible, it should be independent of the programming model and the environment. And this is precisely what active capability *supported at the system level* offers.

Briefly, active DBMSs couple database technology with rule-based programming to achieve the capability of reacting to database stimuli, commonly referred to as *events.* Active DBMSs consist of *passive* databases and a set of event-condition-action (ECA) rules. An ECA rule specifies an action to be executed upon the occurrence of one or more events, provided a condition holds. Therefore, by treating the *significant* operations performed by transactions as *events* and executing particular computations (i.e., condition checking and action execution) when these events are detected, it is possible to realize various transaction semantics. For example, consider the *acquire-exclusive-lock* operation. By treating this operation as an event, it is possible to associate with it one or more condition-action pairs C_1A_1, $C_2A_2, \ldots, C_nA_n$, where each C_iA_i defines the semantics of this operation in a particular transaction model. Hence, to obtain the semantics of lock acquisition in the nested transaction model, the C_iA_i defining its semantics should be *activated.* Consequently, obtaining the transaction semantics of a particular transaction model entails activating the correct C_iA_i pair for each operation defined in the desired transaction model.

There are many advantages to using active capability as a mechanism for realizing various transaction models. First, the utility of active capability for supporting application specific behavior has been well established, as can be observed by the presence of this capability in almost all commercial models, its introduction into SQL3, and the number of research prototypes being developed. The availability of expressive event specification languages (e.g., Snoop, Samos, Ode, Reach) that allow sequence, conjunction and time related events can be beneficial for modeling some of the synchronization aspects of workflow and other transaction models.

Currently, most of the DBMSs (commercial or otherwise) provide active capability at the *application level.* This allows the user/application developer to specify events and rules on application level data (objects) to add additional behavior to applications asynchronously. None of these DBMSs support changes to the *DBMS behavior* asynchronously, a feature necessary for supporting different transaction models. Once this capability is available, it can be used for other purposes, such as restricting the number of concurrent transactions, reorganization of B-trees, etc. However, the presence of active capability at the application level does not guarantee that it can be used at the system level as well.

In fact, depending upon the approach taken it may not even be easy/possible to port the design to the systems level without great difficulty.

We will illustrate later that our approach to the design of active capability allowed us to port it to the systems level with relative ease. In fact, the implementation discussed in this paper supports both application level and system level active capability in a uniform manner. To the best of our knowledge, most of the commercial systems as well as research prototypes do not support active capability at the systems level.

10.2.1 Realizing Transaction Models using ECA rules

Active database paradigm can be used in a number of ways to support flexible transaction models. Below, we examine these alternatives and discuss the merits of each approach, ease of its implementation, and the extent to which it can support extended transaction models. The alternatives for supporting different transaction *given a DBMS* can be broadly classified into the following approaches:

1. Provide a set of rules that can be used from within applications to get the desired transaction semantics. This approach assumes that the underlying DBMS supports *some* transaction model. In this approach, the desired transaction semantics is obtained by enabling the rule sets provided. For example, we assume that there is a set of rules for sagas that can be enabled by a command giving the user the semantics of the saga transaction model. Without any loss of generality we shall assume that rules are in the form of ECA rules, i.e., event, condition and action rules (along with coupling modes, event contexts, priority etc.). This approach certainly enhances the functionality of the system and is a concrete approach towards supporting extended transaction models.

   One advantage of this approach is that new rule sets can be defined (of course by a DBA or a DBC) and added to the system. It may also be possible to add additional rules to slightly tweak the semantics of a transaction model. A limitation is that the set of rules defined are over the events of the transaction model supported by the system, e.g., commit, abort, etc. Consequently, only those transaction models which are very similar to the underlying transaction model can be expressed using this alternative. To elaborate, the Split transaction model cannot be expressed, if the underlying transaction model is the classical ACID model, since the *split* transaction primitive is not provided by the traditional transaction model.

2. Identify a set of critical events on the underlying data structures used by a DBMS (such as the operations on the lock table, the log, and deadlock and conflict resolution primitives) and write rules on these events. This

approach does not assume any underlying transaction model. This approach can be used to support different transaction models including the traditional transaction model. In this approach, system level ECA rules are defined on data structure interfaces to support flexible transactions.

A distinct advantage of this approach is that it will be possible to support workflow and newer transaction models irrespective of whether they are extensions of the traditional transaction model. To elaborate, the rules are now defined on low-level events which act on the data structures directly thereby providing finer control for defining transaction semantics. For instance, a rule can be defined on lock table events such as *acquire-lock* and *release-lock*. This is in contrast to defining rules on high-level events such as commit, abort etc. Another advantage is that a DBMS can be configured using a subset of the transaction models available at the system generation time. This approach may be able to offset the performance disadvantage currently observed in active database systems. The system designer will be in a better position (relatively) to support or extend transaction models.[4]

This approach is similar to the one taken in [Unland and Schlageter, 1992]. They introduce a flexible and adaptable tool kit approach for transaction management. This tool kit enables a database implementor or applications designer to assemble application-specific transaction types. Such transaction types can be constructed by selecting a meaningful subset from a starter set of basic constituents. This starter set provides, among other things, basic components for concurrency control, recovery, and transaction processing control.

3. This is a generator approach using the second alternative. In this approach a high-level specification of a transaction model (either by the user or by the person who configures the system) is accepted and automatically translated into a set of rules. The specification is assumed at the compile time so that either rules or optimized versions of code corresponding to the rules are generated. The advantage of this approach is that the burden of writing rules is no longer on the DBA.

In this paper, we use the second approach which we believe is versatile and meets most of our goals mentioned earlier. Our approach for supporting a given transaction model T_x using active capability is essentially a three step process:

1. Identify the set of operations executed by transactions in the model under consideration. Both application visible and internal operations are taken into account. For example, application visible operations such as *begin transaction* and internal operations such as *acquire lock* are considered. Some of these operations are treated as *events*, i.e., their execution is *trapped* by the active DBMS. It should be emphasized that not all

events detected are associated with operations implemented in the system. Rather, these events can be abstract or external events.

2. The second step involves identifying the condition which needs to be evaluated when an event occurs (e.g., checking for conflicts at lock request time) and the action to be performed if the condition evaluates to true (e.g., granting the lock to the requesting transaction). The events, conditions and actions yield pools of events, conditions, and actions, respectively, which are stored in the DBMS. These pools, depicted in Figure 10.1, form the building blocks from which rules are constructed.

3. The final step involves combining an event, a condition and an action to compose an ECA rule. Each ECA rule defines the semantics of a smaller unit of the transaction model under consideration. For instance, an ECA rule may define the semantics of the *acquire lock* operation. This process is repeated until a *rule set* defining the entire semantics of a transaction model, is built. We allow for the cascading of rule execution. This occurs when the *action* component of a rule raises event(s) which may trigger other rule(s).

This approach allows sharing of the building blocks in several ways. Events, conditions, and actions are shared across rules sets composed for different transaction models. In addition, intermediate rules can also be shared by other rules. Although Figure 10.1 shows a single level for clarity, a hierarchy of rules is constructed from the building blocks. The overlap of events, conditions and actions for different rule sets clearly indicates the modularity and reusability aspect of our approach. This is further substantiated in the section on implementation.

To summarize, our approach encapsulates the semantics of a transaction model into a set of ECA rules. These rules are derived from the analysis of each transaction model as well as examination of their similarities and differences. This encapsulation is done at the level of significant operations (e.g., begin-transaction, commit) that can be treated as events and/or at the level of internal operations on data structures (e.g., lock-table). Once the semantics of a transaction model is composed in terms of these building blocks, rules are written for each block. The availability of begin and end events are useful to model the semantics without having to introduce additional events. Also, the availability of coupling modes and composite events are used to avoid explicit coding of control as much as possible.

The mechanism described above can also be applied for customizing auxiliary behavior. By trapping the operations that are executed by applications, it is possible to perform *auxiliary* actions as required by the user/system designer (i.e., other than those defining the transaction semantics). A good example of this is in systems where optimal performance is achieved when the number

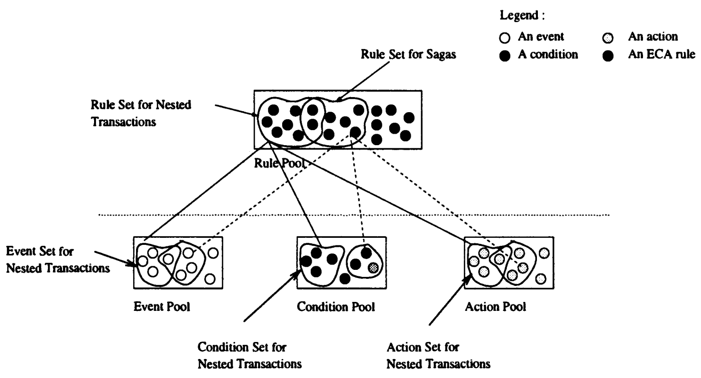

Legend :
○ An event
● A condition
○ An action
● An ECA rule
Rule Set for Sagas
Rule Set for Nested Transactions
Rule Pool
Event Set for Nested Transactions
Event Pool
Condition Pool
Action Pool
Condition Set for Nested Transactions
Action Set for Nested Transactions

Figure 10.1 Rule Composition

of transactions in the system does not exceed a particular threshold (e.g., load balancing and buffer sizes). Therefore, it is necessary to check the number of transactions and not allow the threshold to be exceeded. This can be accomplished by trapping the operation *begin transaction* and checking the number of active transactions at that point. If the number is found to be less than the threshold, then allow the transaction to continue execution, otherwise either abort the transaction or make it wait. Similarly, in banking applications there may be a limit on the number or amount of withdrawals in a day. By defining a rule which is triggered upon detection of the *begin* operation, it is possible to check the number or amount of withdrawals appropriately and either continue or abort the transaction. To summarize, not only does the active database paradigm allow for the specification of transaction semantics but arbitrary semantics as well in an extensible manner.

10.3 IMPLEMENTATION DETAILS

Sentinel, an active Object Oriented Data Base Management System (OODBMS) developed at UF, was used as the platform for implementing the traditional transaction model, nested transactions and Sagas using ECA rules. Sentinel was developed by incorporating *system level* active capability into Zeitgeist, an object-oriented DBMS developed at Texas Instruments. In the following sections, we first begin by describing Zeitgeist with special emphasis given to its transaction manager. We then proceed by explaining how active behavior was incorporated at the *systems level* into Zeitgeist and then used to realize various transaction models.

10.3.1 Zeitgeist

Zeitgeist is an OODBMS which uses the C++ language to define the OO conceptual schema as well as to manipulate the internal object structure. Persistence is achieved by translating C++ objects into record format and storing them in an underlying storage manager, Ingres 6.0.3 in this case. A background process creates a shared memory segment (where the transaction manager's data structures are maintained) as well as resolves any deadlocks. Concurrency control is provided by maintaining lock information in the shared memory segment and regulating its access using semaphores. Recovery is provided by the storage manager.

```
class zgt_tx {

    public:

        friend class zgt_ht;
        friend class wait_for;

        long tid;   // the transaction identifier
        int pid;    // the process identification number of the transaction
        long sgo;  // the storage group number where the object on which the transaction is blocked is stored
        long obno; // the object number of the object on which the transaction is blocked
        char status;  // the current status of the transaction
        char lockmode; // the lockmode requested for the object on which the transaction is blocked
        int semno; // the semaphore number on which the transaction is queued
        zgt_hlink * head;  // this points to a linked list of the locks currently held by the transaction
        zgt_tx * nextr; // this points to the next transaction hashed to this same bucket

        // methods

        zgt_hlink *others_lock(zgt_hlink *, lonh, long);
        int free_locks();
        int remove_tx(zgt_shmem *);
        long get_tid() {return tid;}
        long set_tid(long t){tid = t; return tid;}
        char get_status() {return status;}
        int set_lock(long, long, char);
        int set_lock_no_wait(long,long, char);
        int upgrade_lock_no_wait(long, long, char);
        int end_tx();
        int cleanup();
        zgt_tx(zgt_shmem *);
        ~zgt_tx(){ };
}
```

Figure 10.2 The Transaction Class.

Zeitgeist's transaction manager is implemented using three classes namely, the *zeitgeist* class, the *zgt_tx* class, and the *zgt_ht* class. The *zeitgeist* class implements transaction operations such as abort_transaction, begin_transaction and commit_transaction while the *zgt_tx* class implements operations related to locks such as lock-release and lock-acquisition. The *zgt_ht* implements the lock hash table. Due to space considerations we only depict the *zgt_tx* class definition which is given in Figure 10.2. The transaction manager's data structures (e.g., lock table, hash table) are maintained in shared memory. The *zgt_init* process (in the background) is responsible for creating and attaching the shared memory segment, for allocating and initializing a specific number of semaphores, and for creating and initializing the data structures. This process must be executed before any application can begin execution. Access to these data structures is regulated using exclusive semaphores. The relationship between the above mentioned three classes is illustrated in Figure 10.3.

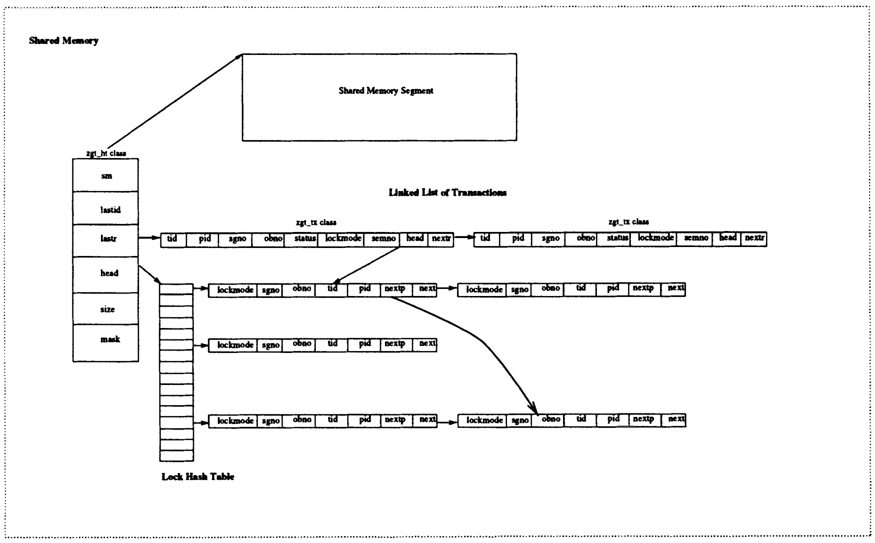


Figure 10.3 Architecture of Zeitgeist's Transaction Manager.

10.3.2 Making Zeitgeist Active at the Systems Level

Sentinel [Anwar et al., 1993, Chakravarthy et al., 1994, Chakravarthy et al., 1995] is an active object-oriented DBMS that seamlessly integrates ECA rules into the object-oriented paradigm. The Sentinel architecture is an extension of the *passive* Zeitgeist system architecture [Texas Instruments, 1993]. The Zeitgeist class hierarchy was modified to include new class definitions which are necessary for supporting active capability. Figure 10.4 depicts the class hierarchy of Sentinel with respect to the Zeitgeist classes. Specifically, the classes introduced are the *Reactive*, *Notifiable*, *Event*, *Rule* and *Event Detector* classes. Note that the classes introduced for making Zeitgeist active (by making it a subclass of the Reactive class as shown in Figure 10.4) is the same for supporting both application- and system-level active capabilities. By making the Zeitgeist class Reactive, all system-defined methods are potential events. Similarly, by making any application object reactive (by making it a subclass of the Reactive class), any method of that class are potential events.

In Sentinel, objects are classified into three categories: passive, reactive and notifiable. **Passive objects** are conventional objects which receive messages, perform some operations and then return results. They do not generate events. An object that needs to be monitored (by informing other objects of its state changes) cannot be passive. **Reactive objects**, on the other hand, are objects that need to be monitored (i.e., on which rules will be defined). A reactive object can declare any, possibly all, of its methods as an *event generator*. All methods declared as event generators constitute a reactive object's *event interface*. Once a method is declared as an event generator, its invocation will generate a primitive event. The primitive event can be generated either *before* or *after* the execution of the method depending on which *event modifier* was specified by the user. The event will be generated before execution and after execution if the user specifies the *begin* and *end* modifier, respectively. In addition, if the user specifies both modifiers then two primitive events will be generated, one before execution and one after execution of the respective method. Lastly, **Notifiable objects** are those objects that are capable of being informed of the events produced by reactive objects. Therefore, notifiable objects become aware of a reactive object's state changes and take appropriate measures (by evaluating conditions and executing actions) in response to those state changes. Notifiable objects *subscribe* to the primitive events generated by reactive objects. After subscription, the reactive objects propagate their generated primitive events to the notifiable objects. Events and rules are examples of notifiable objects. Rules receive events from reactive objects, send them to their local event detector, and take appropriate actions. Event detectors receive events from reactive objects, store them along with their parameters, and use them to detect primitive and complex events. In the following paragraphs we briefly outline the implementation of the *Reactive*, *Notifiable*, *Event* and *Rule* classes. The reader is referred to [Anwar et al., 1993] for a detailed implementation of these classes.

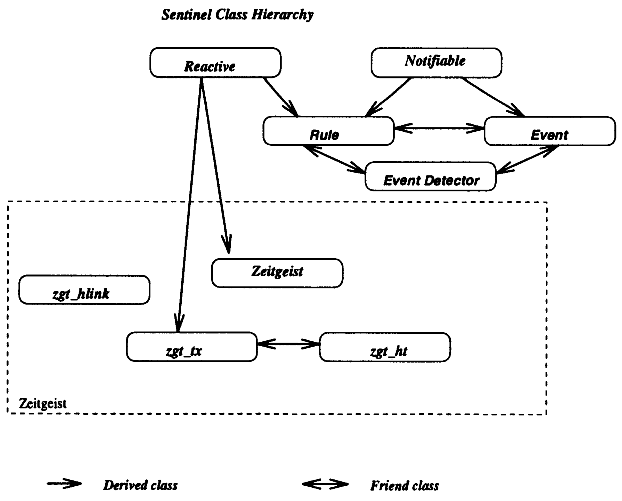


Figure 10.4 System Level Active Functionality.

The Reactive Class: The public interface of the Reactive class consists of methods by which objects acquire reactive capabilities. For an object to be reactive, i.e., have the ability to generate primitive events when methods in its event interface are invoked, it must be an instance of a class derived from the Reactive class.[5] Subclasses of the Reactive class will inherit several methods the most important of which is the *Subscribe* method. This method allows Notifiable objects to subscribe to the primitive events generated by instances of subclasses of the Reactive class. Once this subscription takes place, the notifiable object will be informed of the primitive events generated by the Reactive object. For example, if X is a Reactive object and Y is a Notifiable object, then Y will be informed of the primitive events generated by X after the statement **X.Subscribe(Y)** is executed.

The Notifiable Class: Similarly, the public interface of the Notifiable class consists of methods which allow objects to receive and record primitive events generated by reactive objects. For an object to be notifiable it must be an instance of a class derived from the Notifiable class, i.e., an instance of a subclass of the Notifiable class. The method Record defined in this class documents the parameters computed when an event is raised, namely, the oid of the reactive object generating the event, the event generated, the time-stamp of when the event was generated, and the number and actual values of the parameters sent to the reactive object.

The Event Class Hierarchy: The Event class is the superclass of an event class hierarchy which defines the common structure and behavior shared by all event types. Each event type is a subclass of the Event class. The event types that are supported are primitive as well as complex. The Primitive subclass is for modeling primitive events which are basically method invocations. Creation of a primitive event object requires indicating the *method* which raises the event and *when* the event should be raised, i.e., before or after execution of the method.

The Rule Class : The primary structure defining a rule is the event which triggers the rule, the condition which is evaluated when the rule is triggered, and the action which is executed when the rule is triggered. Therefore, creation of a rule object X is accomplished by executing the statement **Rule X(eventid, Condition, Action)**, where eventid is the oid of the event object representing the event that triggers the rule X, Condition is a function that is to be executed when the event is triggered and Action is a function to be executed if the Condition function returns true.

10.4 REALIZING TRANSACTION MODELS

In accordance with the second alternative for realizing various transaction models using the active database paradigm, we performed the following steps :

- We first stripped the underlying DBMS of its built-in transaction model. This was necessary since the semantics of operations such as lock acquisition, commit, and abort differ from one transaction model to the other, and thus we did not want these operations to have particular semantics. This was accomplished by removing all code from all the methods of both the *zeitgeist* and *zgt_tx* classes, since methods of these classes implemented Zeitgeist's transaction management. Consequently, at this point, if any of the methods of these classes were to be invoked, nothing would happen.

- The second step involved treating the methods of the *zeitgeist* and *zgt_tx* classes as *events*. Thus it was necessary to derive the *zeitgeist* and *zgt_tx classes* from the Reactive class as previously illustrated in Figure 10.4. Once these classes were derived from the Reactive class and methods of the class declared as events, the semantics of methods in these two classes became:

  1. Signal the occurrence of an event when this method is invoked by any transaction. The entity which is notified of the occurrence of the event is the *local event detector* (LED) object.
  2. Marshal the parameters of the event to the LED. For example, if the event which is detected is lock-acquisition, marshal the transaction identifier, the identifier of the object to be locked, and the lock mode requested to the LED object.
  3. Once the event along with its parameters are received by the LED, the LED is responsible for determining which rule should be triggered by the occurrence of this event, and then evaluating the corresponding condition and executing the action if the condition holds. Note that each application has its own local copy of the LED. This allows the application to enable particular rule sets which the LED is responsible for detecting. Consequently, each application can adhere to different transaction semantics by maintaining its own LED and enabling whichever rule sets it requires.

Using the approach sketched above, we have implemented: the traditional transaction model, sagas, and the nested transaction model. There are 18 rules used for the implementation of the traditional transaction model, 22 rules for Sagas, and 36 rules for the nested transaction model. The reader is referred to [Chakravarthy and Anwar, 1995, Anwar, 1996] for the actual ECA rules used for these transaction models.

We learned several important aspects of the use of ECA rules while implementing the various transaction models. The availability and use of nested execution of rules was beneficial for decomposing tasks to avoid replication. This was useful for the nested transactions where the commit and abort operations are extensions/modifications of the conventional commit and abort operations. While implementing Sagas, it became apparent that the semantics of abort of a

Sagas component transaction is different from the abort of a Sagas compensating transaction. The abort of a Sagas component transaction starts executing the compensating transactions in reverse order whereas the abort of a Sagas compensating transaction needs only restart them. This necessitated that we change the abort semantics dynamically (at runtime). We could easily accomplish this in our approach as design of active capability in Sentinel supports dynamic enabling and disabling of rules as well as subscription of events to rules. Prototype implementation also indicated how composite events can be beneficially used for synchronization purposes. For example, synchronizing the commit of a superior with respect to all its children can be accomplished by dynamically creating (which is possible in Sentinel) a Conjunction event, where the commit of each child is a component of the Conjunction event. The semantics of a Conjunction event expects all the component events to occur for the Conjunction event itself to be signaled, disregarding the order of constituent event occurrences. Consequently, once the Conjunction event is signaled, it implies that all the children have committed and thus the parent can continue its execution.

10.5 EXTENSIBILITY

In this section we discuss the extensibility aspects of our approach. There are two distinct aspects of extensibility that need to be addressed: i) extensibility of ECA rules as compared to other approaches (object-oriented and tool-kit) to extensibility and ii) extensibility in modeling newer transaction models. Below, we address each of the above.

We believe that ECA rules at the systems level provide yet another, but more powerful form of extensibility. In contrast to the other two approaches (object-oriented and tool-kit), this approach provides greater control at runtime (with respect to the object-oriented approach) and allows one to redefine semantics dynamically. In a sense, the binding of rules can be controlled by other rules instead of overloading which provides a fixed form of dynamic association.

In contrast to the tool-kit approach, use of rules allows one to support both application-level and system-level modification of behavior in a uniform manner. Further, our approach does not preclude the inline incorporation/compilation of rules to avoid the performance overhead that is associated with rule processing. However, the use of rules allows one to modularize and prototype systems relatively easily.

So far we have used ECA rules defined at the systems level to achieve the semantics of various transaction models. The rule sets defined for the various transaction models focus on the concurrency control aspect of transaction models. Since Zeitgeist uses a lock based method for achieving concurrency control, we also adopted this method in our rules. In particular, we defined events for the operations such as *lock-acquisition*, *lock-release* and *upgrade-lock*. Another reason which prompted our use of a lock-based mechanism for concurrency control, is that it is used in most commercial DBMSs, is well understood and is perhaps the most popular of the concurrency control mechanisms. However, it is important to realize that our approach to realizing transaction models is not limited to a particular concurrency control method. Rather, our approach is extensible enough to be applied to other concurrency control mechanisms, e.g., optimistic concurrency control (OCC).

To show the extensibility of our approach let us assume that OCC using timestamp ordering is preferred over a lock based method. The basic notion behind OCC is to allow transactions to read, compute, and update local copies freely without updating the actual database. Some information is maintained with each data item to ensure serializability of committed transactions. Once a transaction completes it enters a validation phase which consists of checking if the updates maintain the consistency of the database (i.e., the commit of the transaction is serializable). If the answer is affirmative, then the updates are made persistent in the database, otherwise the transaction is aborted.

The three rules of the OCC algorithm [Kung and Robinson, 1981] using read and write sets can be translated into ECA rules when the commit is issued by a transaction. In Zeitgeist only the object-id is kept in the shared memory data structures (along with some other information, but not the value). Local copies of the objects are maintained in the application/client address space. By modifying the tables in the shared memory to keep the timestamp information, it is relatively easy to implement the OCC algorithm based on timestamp order by writing rules on the commit and disabling rules on acquire lock etc. The list of objects accessed by a transaction is already maintained (although there is no distinction between read and write objects) in shared memory.

As previously mentioned, we have concentrated on using ECA rules for the concurrency aspects of transaction models. This, however, does not prohibit its utilization to other aspects of transaction management. Consequently, our approach can also be used for the recovery aspects of transaction management as well as other aspects such as deadlock detection and deadlock resolution. Usage of this paradigm in these other areas of transaction management entails identifying the data structures and operations that need to be detected or trapped as well as defining the semantics of the operations using ECA rules. Therefore, the exact same process used for supporting the concurrency control aspects is utilized to other aspects of transaction management.

10.6 CONCLUSIONS

In this paper, we have taken an extensible approach to support extended transaction models. We have demonstrated a novel application ECA rules at the systems level and concomitant functionality required to support various transaction models. We have shown the implementation details of making a DBMS, such as Zeitgeist active at the systems level. We have analyzed several extended transaction models and derived detailed ECA rules (using low-level data structures) necessary for modeling traditional transactions, Sagas, and nested transactions (with sibling concurrency). We have shown that by not hardwiring the semantics of operations such as commit, abort and acquire-lock, and detecting events (primitive and abstract) at runtime, it is possible to realize different transaction models. Our approach is extensible and can with relative ease support current transaction models as well as newer transaction models as they become available (by reusing existing rules as much as possible). The DBC can add or modify the class interface of the underlying data structures and define additional rules on its operations. In order to demonstrate the versatility of our approach, we have used data structures (on which rules were defined) that are similar to those found in most commercial transaction managers. In particular, our approach assumes no specific underlying architecture or database model and can be applied to any active DBMS.

In this paper, we focused on addressing the concurrency control and functionality issues related to supporting various transaction models. We are currently investigating other related issues, primarily recovery, performance, and optimization of system level ECA rules, and more importantly allowing the concurrent execution of applications adhering to different transaction models.

Notes

1. This paper delimits itself to the discussion to concurrency control aspects of a transaction model; recovery issues are currently being investigated and is not addressed in this paper.
2. Although TSME [Georgakopoulos et al., 1994] also use rules for enforcing transaction dependencies, they have not specified implementation-level details of event detection, condition evaluation, and action execution.
3. Although support for different transaction models, to some extent, can be accomplished in an object-oriented environment by creating a transaction hierarchy and overloading the operations or methods, this approach is specific to the model used rather than the system.
4. We would like to point out that the use of ECA rules by themselves will not make the system completely flexible. However, we do believe that the process of identifying primitive events, details of conditions/actions and writing these rules will make us reexamine the current architecture and the data structures to progress towards a modular systems architecture.
5. Another way a class can become a reactive class is if it is a friend class of another reactive class.

Acknowledgments

This work is partly supported by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory. This work is also supported, in part, by the NSF Grant IRI-9528390.
Part of this work was performed during the author's internship at T. J. Watson Research Center, Hawthorne, NY.

11 INTER- AND INTRA-TRANSACTION PARALLELISM FOR COMBINED OLTP/OLAP WORKLOADS

Christof Hasse and Gerhard Weikum

Abstract:

This paper presents the architecture and run-time mechanisms of an experimental prototype system, PLENTY, that is specifically geared for combined OLTP/OLAP workloads with update transactions and complex queries concurrently executing on the same database. The system is able to parallelize both retrieval and update transactions at the level of precedence-graph scripts with nodes corresponding to SQL-like statements or internal operator trees. Employing this form of intra-transaction parallelism in a multi-user environment reduces the lock duration and thus the potential for data contention, so that OLTP and OLAP applications can be reconciled with good performance on the same shared database. The implementation of the underlying concurrency control and recovery mechanisms is based on multi-level transactions. In addition to presenting the overall architecture and internals of this approach, the paper also discusses heuristic scheduling strategies for combined workloads within the given framework.

11.1 INTRODUCTION

Online transaction processing applications (OLTP applications) are characterized by a potentially large number of concurrently executing, relatively short update transactions with high throughput demands. A transaction typically comprises a few primary-key-driven SQL commands in the simplest case (e.g., TPC-B Debit/Credit [Gray, 1993]) up to twenty or thirty SQL commands (e.g., TPC-C NewOrder [Gray, 1993]), and should have a response time in the order of a few seconds at worst. Online analytical processing applications (OLAP applications), on the other hand, are characterized by complex decision-support queries with a moderate degree of concurrency. OLAP transactions typically include one or more resource-demanding queries with joins and aggregations on very large tables (see, e.g., TPC-D [Raab, 1995]). The TPC-C StockLevel transaction may be viewed as a simple OLAP transaction; much more complex examples arise in data mining applications, e.g., on customers and sales data, or in portfolio management and financial trading applications within banks.

Traditionally, OLTP and OLAP applications cannot be easily reconciled on the same database because of the resulting resource contention and, especially, data contention caused by locking or whatever concurrency control protocol is used. There are several approaches to overcome or work around this problem:

1. OLTP and OLAP applications are run on two different databases, where the OLAP database is a read-only snapshot copy of the OLTP database which is periodically (e.g., on a daily basis) brought up to date. This is a widely used approach that forms the core of most data warehousing architectures [Widom, 1995]. Its inherent disadvantage is that OLAP transactions are run on potentially stale data.

2. To reduce the data contention between OLTP and OLAP transactions, OLAP transactions are executed in a relaxed isolation mode, sacrifizing serializability. This can be done by using, for example, the SQL isolation level "Read Committed" [Berenson et al., 1995], or by transforming an OLAP transaction into a chain of separate transactions with intermediate commit points. The problem here is that OLAP transactions may see inconsistent data (unless specific knowledge on the possible transaction interleavings is available and can be exploited [Shasha et al., 1995]).

3. Since OLAP transactions are mostly read-only transactions, simple multi-version concurrency control protocols [Chan et al., 1982, Mohan et al., 1992b, Brown and Carey, 1992] can eliminate the need for locking in the OLAP transactions, while still guaranteeing a consistent view of the data that reflects the most recent committed state. However, these protocols are not applicable for OLAP transactions that contain a few updates; this case is not exactly typical, but does appear in some applications. More general multi-version concurrency control protocols [Bernstein et al., 1987] that uniformly deal with read-only and update transactions have not (yet) matured to industrial viability.

4. By exploiting semantic properties of the application's database operations, most notabably, the commutativity of certain operations, the conflict probability of transactions can be significantly reduced [O'Neil, 1986, Lynch et al., 1994]. This approach is primarily applicable to OLTP transactions where increment and decrement operations on numerical data (e.g., financial data) are frequent. Alleviating the data contention among OLTP transactions is a significant benefit also for OLAP transactions, as it reduces the probability of transitive waiting and thus the duration of data-contention delays.

5. Finally, parallelizing long OLAP transactions reduces the duration for which locks need to be held in proportion to the achievable speedup. This again reduces the conflict probability and the delays upon conflicts.

Among these approaches, the first two are prevalent in practice. Unfortunately, whereas they are acceptable in many mining-style applications where not quite recent or slightly inconsistent data is tolerable in statistical evaluations, applications such as financial trading require both up-to-date and consistent data and can, therefore, not be served by one of these approaches. The methods 3 through 5, on the other hand, provide theoretically well-founded solutions that are generally applicable without such caveats. Among these three, the multi-version concurrency control method (approach 3) has gained most practical relevance and has been well investigated. However, its limitation to read-only transactions may be a problem for some applications. Therefore, it is important to study the approaches 4 and 5 in more depth, and investigate also to what extent the methods 3, 4, and 5 can be combined. For example, a novel approach to combining multi-version protocols with exploiting commutativity properties of update operations has been recently proposed in [Jagadish et al., 1997]. In this paper, we explore the alternative research avenue of combining semantic concurrency control and intra-transaction parallelism (approaches 4 and 5) for alleviating the data contention among OLTP and OLAP transactions.

The paper presents an architecture for a combined OLTP/OLAP server that has been fully implemented in an experimental prototype system for shared-memory multiprocessors, coined PLENTY (standing for "ParalleL Execution of Nested Transactions on plentY of processors"). The architecture supports intra-transaction parallelism for both read-only and update transactions and can exploit semantic properties like the commutativity of specific operations. The execution engine employs multi-level transactions as a rigorous basis for its high-concurrency transaction manager. While this forms the core of the system's execution *mechanisms* (as far as the paper's subject is concerned), the paper also presents execution *strategies* for the combination of inter- and intra-transaction parallelism in that it discusses heuristics for the CPU scheduling of transactions and subtransactions.

The remainder of the paper is organized as follows. Section 11.2 briefly introduces the necessary background on multi-level transactions. Section 11.3 gives an overview of the architecture of the PLENTY prototype. Sections 11.4 through 11.6 then constitute the paper's algorithmic core: Section 11.4 discusses a precedence graph concept for driving parallelized transactions and its relationships to multi-level transactions, Section 11.5 outlines the algorithms for concurrency control and recovery, which are based on [Weikum and Hasse, 1993], and Section 11.6 introduces the CPU scheduling heuristics. Section 11.7 illustrates the applicability of our approach with a case study of a (simplified) foreign-exchange banking application. We conclude with an outlook on open issues.

11.2 BACKGROUND ON MULTI-LEVEL TRANSACTIONS

Multi-level transactions [Weikum, 1991, Weikum and Schek, 1992] are a variant of nested transactions where the nodes in a transaction tree correspond to executions of operations at particular levels of abstraction in a layered system. The edges in a transactions tree represent the implementation of an operation by a sequence of operations at the next lower level.

The point of multi-level transactions is that the semantics of high-level operations can be exploited in the conflict definition in order to increase concurrency. For example, in the conflict relation shown in Figure 11.1, two *Buy* and *Sell* operations on the same investment account are not in conflict, given that they essentially decrement and increment counters, and can therefore be admitted concurrently. However, executing such high-level operations in parallel requires that a low-level synchronization mechanism takes care of possible low-level conflicts, e.g., on indexes or data pages. Therefore low-level locks are acquired only for the duration of the high-level operation and are released at the end of the operation, that is, at the end of the subtransaction, thereby reducing the low-level lock conflicts.

	Retrieve	Sell	Buy
Retrieve	+	-	-
Sell	-	+	+
Buy	-	+	+

Figure 11.1 Conflict definition for Retrieve, Sell, and Buy operations

Figure 11.2 shows the concurrent execution of two banking transactions T1 and T2. The execution of the read/write operations at the lower page level is not acceptable with respect to T1 and T2. For example, T1 would still hold locks for the write operation on p and q. Therefore, T2 would have to be delayed until the end of T1, so that no parallelism would be feasible. However, at the higher level, one can exploit the fact that the Sell and Buy operations do not conflict. Therefore the execution of the Sell and Buy operations is correct, namely, serializable with respect to T1 and T2. Furthermore, the execution of the read/write operations at the lower level is serializable with respect to the operations at the higher level, and therefore the high-level operations appear as if they were isolated subtransactions. Thus, by exploiting the semantics of the operations at the higher level and by early release of low-level locks, multi-level transactions provide the potential for more concurrency.

Intra-transaction parallelism can be turned into inter-subtransaction parallelism at the lower level [Weikum and Hasse, 1993]. Since the subtransaction

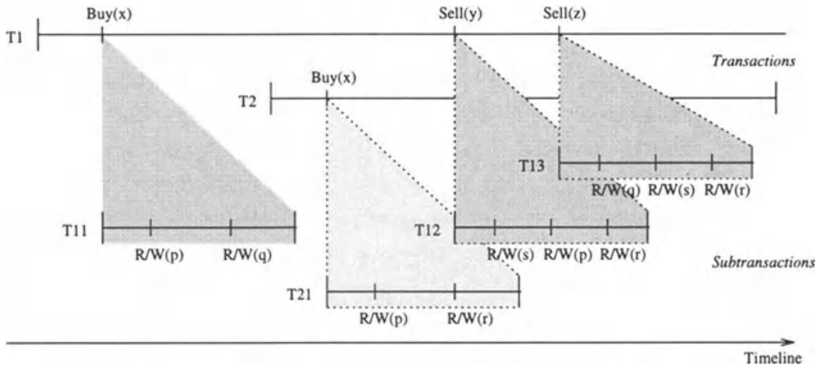


Figure 11.2 Concurrent execution of two transactions

management at the lower level handles subtransactions independently of their parent transaction, a transaction may execute multiple subtransactions concurrently, as illustrated in Figure 11.2 by subtransactions T12 and T13 of T1. So, provided that control flow dependencies at the higher level are observed, the resulting subtransactions of the same transaction can be executed in parallel without having to bother about possible violation of data consistency. However, although the high-level operations of two parallel subtransactions do not conflict, low-level conflicts may arise. For example, when T12 and T13 are executed in parallel, low-level conflicts on s and q would arise so that subtransaction T13 must be delayed until after T12 releases its (low-level) locks. Thus, by simply exploiting the properties of multi-level transactions, intra-transaction parallelism involving update operations of the same transaction can be easily accomodated.

11.3 THE PLENTY ARCHITECTURE

This section gives an overview of the PLENTY prototype system for shared-memory multiprocessor architectures [Hasse, 1995]. Being designed as a performance experiment platform, PLENTY contains a simple relational database engine. Relations can be stored as heap, hash, or B-tree files; the implementation of this storage layer is based on the BSD 4.4 database library. The storage layer has been extended by a page buffer that is managed under an LRU policy and integrated with PLENTY's transaction manager.

Design of PLENTY'S tuple layer is similar to Volcano system [Graefe, 1994]. On top of the storage layer, PLENTY provides an operator-tree interface. Each operator consumes one or more tuple streams and produces one or more result stream. The leaves of an operator tree are scans on stored relations or indices, and the root of a tree delivers a result to the application. Tuple streams can be synchronous such that the consuming operator is responsible for the demand-driven evaluation of the corresponding subtree, or asynchronous such that the producing operator eagerly delivers tuples in a data-driven manner, subject to dataflow control to avoid overflow of intermediate tuple buffers. The operators themselves are expected to reside in an extensible operator library. A few basic operators such as nested-loop join have been implemented; more operators can be added relatively easily. Operators can have parameters in addition to the tuple streams, and they can use a global variable space along with stack-based computations.

Application clients submit a query or update operation by invoking an operator tree, where the operator tree should be regarded as the code that is generated for an SQL operation. In addition, clients can combine multiple operations of this sort into a script, similar to a (very simple form of) stored procedure. This is discussed in more detail in Section 11.4.

Clients can combine either an entire script or an arbitrary set of operator-tree invocations into a transaction by issueing the corresponding BOT (begin of transaction) and EOT (end of transaction) or RBT (rollback transaction) calls. The transaction manager internally implements each transaction as a multi-level transaction where the invoked operators in an operator tree are the higher-level operations and the resulting page accesses form the lower-level operations.

The process model of PLENTY is that of a multi-threaded server. Each invoked operator can spawn a separate thread within a shared address space. This holds for operators in the same tree as well as operators in different trees of the same transaction and also across different transactions. The implementation is based on the POSIX thread library.

The complete PLENTY system contains about 30,000 lines of C code. It is running on shared-memory multiprocessors under Sun Solaris 2.3 and compatible platforms. A particularity is that it can be recompiled to run in a simulation mode, where all thread-related calls are replaced by corresponding calls of the simulation library CSIM [Mesquite, 1995] and all internal functions of PLENTY issue additional calls to use virtual processors and virtual disks and consume the corresponding virtual time. Note, however, that the same code is executed in both simulation and real-execution mode. The main purpose of the simulation mode is to be able to run performance experiments on configurations with resources that exceed those that are available to us.

11.4 GRANULARITY OF PARALLELISM

As discussed in Section 11.3, applications can interact with PLENTY by invoking operator trees. Within these operator trees, disjoint subtrees can be executed in parallel and pipelining may be exploited along a producer-consumer path. In addition, the usual form of data parallelism is feasible in PLENTY, too, by instantiating an operator multiple times and partitioning its incoming tuple streams such that each operator instance processes one of the resulting streams. In PLENTY the resulting "wide" tree is represented explicitly; that is, subtree templates are replicated. Note that there may be more compact ways of representing a data-parallel operator tree, and also note that some operators require more sophisticated dataflow directives for the proper routing of partitioned tuple streams; but these issues are not in the focus of the current paper.

So far, the parallelization is in line with the standard architecture of parallel database systems [DeWitt and Gray, 1992, Graefe, 1994]. Applications with data-intensive individual operations such as join queries benefit largely from this architecture. However, applications that rather consist of many simple operations including primary-key-driven updates do not benefit at all, as none of the simple operations justifies parallelizing an individual operator nor does the simple form of operator trees warrant the use of function parallelism or pipelining. Consider, for example, an application that updates a large number of single tuples with each update based on a specified primary key. Without rewriting the entire application, no parallelism could be exploited.

To this end, PLENTY offers to the application clients the option to combine a set of application functions in a simple form of stored-procedure-like scripts (similar in style to [Reuter and Schwenkreis, 1995] but much more limited). A script in this sense is a *precedence graph*, i.e., an acyclic directed graph, whose nodes correspond to the invocation of application functions. This captures, in a very simple style, two types of control flow of the application: connected nodes are processed sequentially whereas parallel branches in the graph denote parallelism. Note that, although such a script bears some relationship to workflow specifications [Rusinkiewicz and Sheth, 1995, Georgakopoulos et al., 1995, Mohan, 1996, Jablonski and Bussler, 1996], this approach is, of course, way too limited for real workflow applications. Recall that our goal here has been to build an experimental platform for studying the mutual impact of inter- and intra-transaction parallelism; the simple script approach is sufficient for this purpose.

The application functions that correspond to the nodes of a script's graph can be arbitrary C functions. Typically, each such function would contain one call to invoke an operator tree of the database engine. As an example of a script consider the NewOrder transaction of the TPC-C benchmark [Gray, 1993] as depicted in Figure 11.3. The precedence graph contains a BOT source and an EOT sink as transaction brackets. Within these brackets, each SQL operation, or actually its corresponding operator tree, along with the surrounding application code constitutes one node of the graph. Whenever there is a control flow or data flow dependency between two nodes (e.g., output parameters of one function are input parameters of the other), a total order is enforced by connecting the two nodes with an edge. For example, the"Select District" and "Update

District" steps should be sequential as the latter may depend on the outcome of the former (e.g., the district may not exist in the database), whereas the "Select Warehouse" and "Select District" steps are independent of the district processing and thus form a parallel branch.

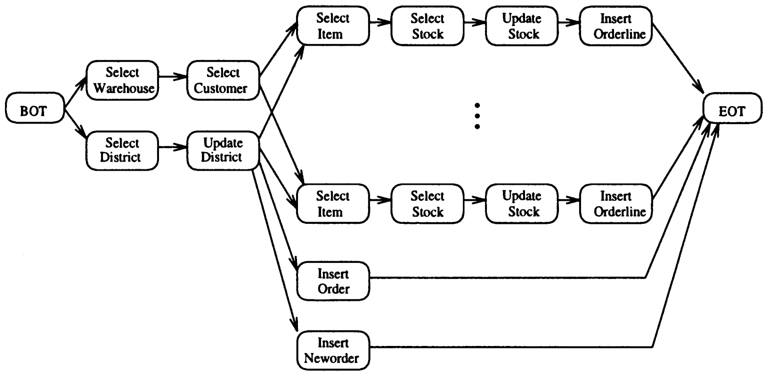


Figure 11.3 Precedence graph representation of a NewOrder transaction

Obviously, the main potential for intra-transaction parallelism in this case stems from the processing of the set of ordered items; this is shown by the parallel branches each starting with a "Select Item" operation. Note that within each such branch, the control flow is sequential. Further note that the number of these parallel branches would actually be variable; the benchmark specifies the number of ordered items as a uniformly distributed random variable between 5 and 15. Scripts can be configured at run-time, after all input parameters of a transaction are known; this is supported by means of a set of simple graph-constructing functions such as "AddNode(...)", "AddPrecedence(...)", and so on.

So the granularity of intra-transaction parallelism in PLENTY can be an application function of a graph-type script, or an operator instance of an operator tree invoked from an application function. In this paper, we will focus on the coarser granularity, the script nodes, as this is the novel aspect of PLENTY. The mapping of these granules to the process architecture of PLENTY is straightforward: each node of the precedence graph spawns one thread. The mapping to subtransactions of a multi-level transaction is less straightforward. The natural default mapping would be to spawn a new subtransaction for each invoked operator tree. Since typically an application function of a script would invoke exactly one operator tree, this default mapping coincides with viewing each node of the precedence graph as one subtransaction. However, in some applications, the enhanced concurrency that is potentially obtained from multi-level transactions may be less important than limiting the overhead of the transaction management. Therefore, it is also allowed to combine a sequential path of nodes into a single subtransaction by explicitly issueing BOS (begin of subtransaction) and EOS (end of subtransaction) calls at the beginning and end of the path. Such a specification is graphically illustrated in Figure 11.4, with subtransactions shown as shaded boxes. The transactional semantics of such a subtransaction is that it forms a unit of isolation and atomicity at the page level of the system. This means that all page locks that are acquired in the course of the execution are held until EOS. As usual, tuple level locks are acquired as well and held until EOT, the completion of the entire transaction. Note that, with the coarser granularity of multi-operation subtransactions, multiple high-level locks are acquired between the BOS and EOS of one subtransaction.

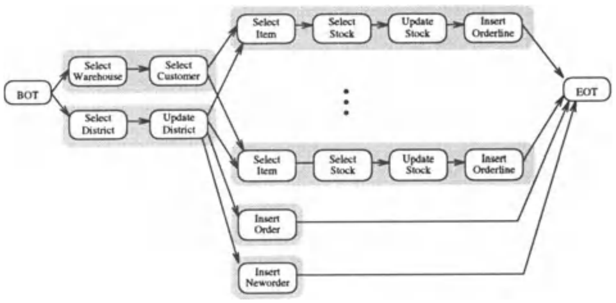


Figure 11.4 Subtransactions of a NewOrder transaction

The flexibility in the mapping of script nodes to subtransactions is limited to nodes in the same sequential path. We disallow grouping nodes from parallel branches (i.e., non-ordered nodes according to the partial order defined by the precedence graph) into one subtransaction. The reason is that any pair of possibly concurrently executing nodes may cause low-level conflicts at the page level when one of them invokes a high-level update operation (e.g., a conflict on index data). Thus, unless the nodes are definitely executed sequentially (as enforced by a precedence edge), we need a low-level concurrency control mechanism. But this is exactly why we use subtransactions. So potentially conflicting concurrent execution units must belong to different subtransactions. Note that the same argument would hold for whatever form of low-level execution units is used, and not only for a multi-level transaction architecture. Further note that similar considerations have recently been brought up in the context of denoting transaction boundaries in workflow specifications [Leymann, 1995] without going into implementation issues. The PLENTY architecture essentially offers a (admittedly not very refined but working) solution to this workflow-related problem as well.

11.5 TRANSACTION MANAGEMENT INTERNALS

This section discusses the implementation of PLENTY's transaction manager, with particular consideration of how the parallelized execution of scripts (see Section 11.4) interacts with the log and recovery management. The transaction manager of PLENTY uses multi-level transactions internally: at the higher level, each invoked operator tree is viewed as an operation of a transaction, and at the lower level each of these operations corresponds to a subtransaction that consists of a set of page reads and writes. The concurrency control for both levels is implemented by the same generic lock manager that is driven by conflict predicates for the operations under consideration. For example, consider the operations Retrieve, Buy, and Sell, shown in Figure 11.1, which have been implemented and included as operators in PLENTY's operator library. The lock manager expects a conflict-testing function for each pair of operators, to decide if there is a conflict or not (based only on the two operators and their actual parameters). Typically, the test is based on the (state-independent) commutativity of the operations; so it is reasonably straightforward for an application to provide these functions. Special care is taken in the deadlock detection to consider also the fact that transactions wait for the completion of subtransactions which may lead to deadlocks that arise from the combination of lock waits at both levels (and could not be recognized by the local view of one level only).

As discussed in prior publications [Weikum, 1991, Weikum and Schek, 1992], employing a multi-level concurrency control protocol requires also a multi-level approach to recovery. The multi-level recovery method that has been implemented in PLENTY is an improved version of the algorithm described in [Weikum and Hasse, 1993]. It is based on undo logging for the high-level operations (i.e., the invoked operator trees) and page-level redo logging for subtransactions, using the following principles:

1. *Transaction atomicity:* Transaction undo, both for rolling back a single transaction and for undoing loser transactions after a crash, makes a backward pass over the high-level log file, following a backward chain for each loser transaction. A log entry (apart from BOT and EOT or RBT log records) describes an inverse or compensating operation for the corresponding forward operation. A log record is generated in a log buffer when the forward operation completes, and is forced to the disk-resident log file when the subtransaction to which the forward operation belongs is made persistent by the low-level log manager.

2. *Subtransaction atomicity:* Subtransactions are guaranteed to be atomic by the low-level recovery, so that the high-level undo pass will always "see" only the effects of complete subtransactions and is thus well-defined. This guarantee is implemented by applying the DB Cache method [Elhardt and Bayer, 1984] to the sets of page writes that are enclosed within

subtransactions. This method keeps transient page before-images in the buffer pool, ensuring that dirty buffer pages are not flushed to disk before the end of a subtransaction, and writes the complete set of after-images to a circular log file in a single, sequential and atomic disk I/O. Upon restarting the system after a crash, the low-level recovery is initiated first; it makes a forward pass over the low-level log file and redoes all completed subtransactions found on the log by re-installing these subtransactions' after-images into the database.

3. *Transaction persistence:* The I/O efficiency of the logging method is improved by deferring the log disk I/O of the low-level log manager until the end of a transaction rather than force-writing after-images upon each end-of-subtransaction. This ensures the persistence of completed transactions. However, since page-level locks are released upon the end of a subtransaction, one has to be careful about possible incompatibilities between the serialization order of subtransactions and the ordering of their after-image-sets on the log file. Adding the proper bookkeeping of dependencies leads to the notion of "persistence spheres". A persistence sphere is associated with a subtransaction or transaction T, and contains all after-images of T itself and, in addition, the after-images of all subtransactions that have a page-level write-write or write-read conflict with T. Persistence spheres are formed dynamically as transactions are executed, log I/Os take place, and dirty buffer pages are written back into the disk-resident database. When a subtransaction needs to be made persistent, the entire persistence sphere is written to disk in a sequential and atomic I/O and then dropped from the bookkeeping. As newly produced after-images supersede previous not yet forced after-images, this method yields a batching effect for the log I/Os and reduces the overall log I/O rate. Details of the method are given in [Weikum and Hasse, 1993].

During the high-level undo pass of a restart, both low-level redo logging and high-level undo logging are again in effect to ensure restart idempotence. In contrast to [Weikum and Hasse, 1993], PLENTY has adopted ARIES-like compensation log records (CLRs) [Mohan et al., 1992a] for tracking the progress of undo steps: a CLR marks the successful completion of an undo step in the high-level log and points to the predecessor of the undone forward operation within the same transaction. Further techniques for reducing the overhead of a multi-level recovery method, such as merging the high-level log and the page-level log into a single log file, are discussed in [Lomet, 1992, Weikum and Hasse, 1993], but are not of specific interest in this paper.

Finally, consider how parallelized scripts are taken care of by this multi-level logging and recovery method. Recall from Section 11.4 that a script corresponds to a precedence graph of application functions and that subtransaction boundaries are constrained such that all operator tree invocations within a subtransaction must lie on a sequential path. Thus the precedence graph directly induces a coarser precedence graph between the subtransactions. Then, undoing a transaction can exploit essentially the same intra-transaction parallelism that is feasible in the transaction's forward execution, and this is achieved by simply reversing the edges of the precedence graph. The reverse precedence edges are included in the high-level log records. When the undo pass follows the backward chain of a transaction, it takes notes on all encountered inverse precedences. By default, undo steps are spawned asynchronously as separate threads, so that the undo pass can proceed with the next (or, actually, chronologically preceding) log record before the undo step is completed. However, when a log record is referenced by one or more inverse precedences, the corresponding undo step is initiated only after the undo steps for all the referencing log records are completed. The reason for this special care is that the corresponding high-level operations on a sequential path within a transaction are not necessarily commutative, whereas one can always assume commutativity for the high-level operations that reside in different parallel branches (otherwise the parallelization should be disallowed in the first place). Note that the multi-level transaction management also supports that such a parallelized transaction rollback proceeds in parallel to the forward processing of other transactions. The correctness of this approach is further investigated from a theoretical angle in [Hasse, 1996].

As an example, Figure 11.5 shows the log records written for the execution of Figure 11.2. In the figure, a system failure is assumed to occur before T1 can finish. The figure also includes the log records that are written during the restart to undo T1. It is assumed that T1's subtransactions T12 and T13 have no precedence constraints between them, but both must follow T11. Thus, the undo log records for these two parallel subtransactions both contain a reverse precedence pointer to the log record of T11.

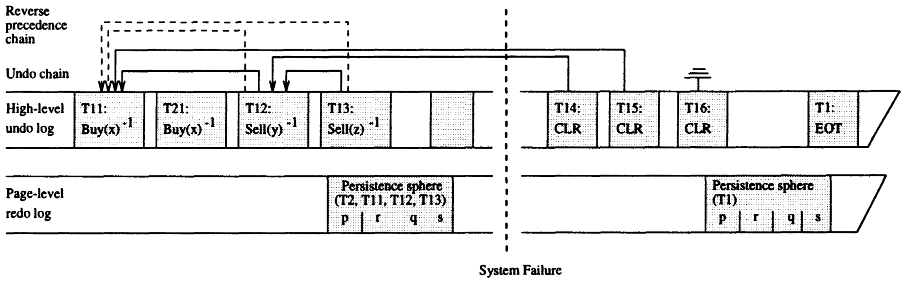


Figure 11.5 Log records written by PLENTY

11.6 SCHEDULING STRATEGIES

So far, the focus of the paper has been on execution *mechanisms* for inter- and intra-transaction parallelism, and efficiency considerations are limited to striving for the highest possible concurrency (by means of semantic multi-level concurrency control) and low overhead (by means of efficient logging and recovery algorithms as well as light-weight thread management). In this section, we consider the major issue in the execution *strategies*, namely, the actual CPU scheduling of transactions and subtransactions. We make the simplifying assumption that no parallelism is exploited within an invoked operator tree (although this would be more than appropriate in a real application), and rather focus on intra-transaction parallelism at the script level, which is the major novel issue of this paper. Thus, the units of the CPU scheduling are transactions and subtransactions as specified at the script level (see Section 11.4). Note that this restricted setting poses already very challenging resource management problems; combining this form of intra-transaction parallelism with intra-operator parallelism and pipelining within operator trees would be an even greater challenge that is beyond this paper's scope (nor has been tackled in the literature, to our knowledge).

In terms of the scheduling theory [Graham et al., 1979, Lawler et al., 1993, Pinedo, 1995], we are dealing with a precedence-constrained multiprocessor scheduling problem, where all subtransactions of all transactions form the pool of dispatchable tasks with precedences described by the union of the corresponding transactions' precedence graphs. This problem is known to be NP-hard; so there is virtually no hope for a scheduling strategy that is both efficient and optimal in the quality of the produced schedules. In fact, however, the problem that we are addressing here is even more complex than the precedence-constrained multiprocessor scheduling problem in a number of ways:

1. The scheduling problem considered here is a hierarchical one with two levels: subtransactions are the actual dispatchable units that are assigned to processors, but the overall performance metrics, throughput or mean response time, are tied to the unit of transactions. Unfortunately, the literature on scheduling contains only little work along these lines. Good heuristic algorithms have been developed for the case when there are no precedence constraints [Turek et al., 1992, Turek et al., 1994] and the case with trees as precedence graphs [Wolf et al., 1995, Chekuri et al., 1995, Garofalakis and Ioannidis, 1996] (motivated by operator trees for database queries), but these cannot be applied to our problem.

2. We address an *online scheduling* problem where the tasks arrive at the system over time and the scheduler needs to make dynamic decisions as tasks arrive. This area is still largely unexplored (see, e.g., [Feldmann et al., 1993] for recent theoretical results).

3. Even online scheduling usually makes the assumption that the task execution time is exactly known at the arrival of the task. This assumption does not hold in real applications. There, the best we can hope for is an estimation of the task's execution time (and other resource demands such as memory requirements), based on task types and probability distributions for the execution time of the various types.

4. The CPU scheduling of transactions and subtransactions interacts in a complex way with the concurrency control. For example, transactions that wait for a lock are not dispatchable at all, and assigning many processors to transactions that are likely to become blocked soon would be an unwise scheduling decision. Unfortunately, the nature of transactional locking is too complex to tie in locks as an additional resource type into standard scheduling theory.

The additional complexity discussed above justifies using a set of pragmatic heuristics for the scheduling of transactions and subtransactions, as elaborated in the following. Problem 3 above, lack of exact information, is circumvented by indeed relying on estimates. For each node in the script of a transaction type, we expect statistical knowledge of the node's execution time so that we can derive an expected execution time for each subtransaction. For the purpose of scheduling decisions, we treat these estimates as if they were exact, given that we cannot make any intelligent decisions at all if no information is available (in that case, a first-come-first-served strategy is the only reasonable choice). However, we take into account dynamic corrections in two ways: first, statistical expectations may change due to long-term workload evolution, and second, derived information like the total execution time of an entire path in the precedence graph is incrementally recomputed at run-time as we gain more information about the actual execution time of running or already completed subtransactions.

Problem 1 of the above list, the hierarchical nature of the scheduling units, is addressed by using a two-tier scheduling strategy based on the following considerations:

- *Primary scheduling:* We are primarily interested in the performance of transactions (as subtransactions are transparent to the applications). Our objective is to minimize the mean transaction response time for a given throughput (i.e., arrival rate) of different transaction types. If each transaction were a sequential program and preemption is disallowed, it is well-known that the mean response time is minimized by a shortest-task-first (STF) strategy [Pinedo, 1995]. However, since a transaction can use multiple processors in our setting, the STF rule needs to be modified into a least-work-first (LWF) strategy [Sevcik, 1994], where the work of a transaction is defined as the sum of the execution time of all its subtransactions regardless of whether these are parallel or sequential. An additional consideration finally is that the remaining work of a transaction changes as the transaction makes progress. This suggests actually using the dynamic counterpart of LWF, namely a *least-remaining-work-first (LRWF)* strategy [Sevcik, 1994]. However, a major drawback of this family of strategies is that it may treat different classes of transactions in an unfair manner, so that long transactions would become susceptible to starvation. Thus, in spite of the above considerations, we have actually adopted the conservative first-come-first-served (FCFS) strategy at the transaction level. Studies of the LRWF strategy, with possible enhancements to prevent starvation, are an interesting subject of future work.

- *Secondary scheduling:* The actual dispatchable units are the subtransactions of active transactions. We consider all subtransactions of the same transaction as a *task group* and now address the scheduling of tasks within a group. The simplest idea would be to apply the LWF rule to subtransactions, too. However, this heuristics would aim at a minimum mean response time of the subtransactions, which is an inadequate metrics at this scheduling level. As the transaction is completed only when its last subtransaction terminates, there is no point in considering the mean response time within a task group. Rather what matters is the duration of the entire "mini-schedule" that is constituted by the task group. This metric is known as the *makespan* of a schedule; minimizing the makespan amounts to maximizing the throughput. For a "vanilla" task system with sequential tasks and dynamic arrivals but without precedence constraints, a well-known, effective heuristics towards minimizing the makespan is the longest-task-first (LTF) strategy [Pinedo, 1995]. Note that each subtransaction is indeed a sequential task, but the existence of precedence constraints requires a more refined strategy. The most important scheduling heuristics for tasks with precedences is the critical-path method (CP), which always gives top priority to the task that heads the longest path in the precedence graph [Pinedo, 1995]. These two heuristics, LTF and CP, can be combined and further generalized into another heuristics that we refer to as the *most-work-first (MWF)* strategy, where the work of a subtransaction is defined as the sum of its own execution time and the execution times of all subtransactions that follow it in the precedence graph. We have adopted this method for the subtransaction scheduling.

PLENTY provides a library of different scheduling algorithms for these two levels of CPU scheduling to support experimentation, but the FCFS/MWF combination outlined above is the heuristics that we advocate for our problem setting. In addition, PLENTY can enforce a specified bound on the maximum number of processors that a transaction may use, to avoid that a single transaction monopolizes the system in that it uses many processors while attain-ing only a moderate speedup. This bound will be referred to as the *DIP limit* where DIP means *degree of intra-transaction parallelism*. Ideally the scheduler would itself select optimal values for these DIP limits; however, this is beyond the scope of this paper (and the state of the art in general, unless one considers special cases). Rather we resort to turning the DIP limit into a tuning knob that can be specified on a per transaction type basis. We disallow preemption, unless a subtransaction becomes blocked due to locking (where a high-level lock wait is attributed to the subtransaction that issued the lock request, although the lock will eventually be held until EOT rather than EOS).

The final point in the above list of problems beyond standard scheduling is the interrelationship of CPU scheduling with data locking (problem 4). Although the presented scheduling strategy does not explicitly consider locking, it behaves favorably also in terms of lock-contention issues. In particular, the MWF strategy for the subtransactions of running transactions contributes to short lock durations by "pushing" transactions that would cause long lock waits if they block other transactions. One may conceive further enhancements to the scheduling strategy along these lines, by dynamically adjusting the priority of transactions and their subtransactions depending on whether they block other transactions or subtransactions. Exploring this research direction is left for future work, however.

The implementation of the described scheduling strategy is based on a *transaction-ready-list* that contains all transactions that have at least one subtransaction that is ready to run, and a separate *subtrans-ready-list* for each transaction that contains all its subtransactions that are ready to run (i.e., whose predecessors in the precedence graph are terminated) and are not blocked by a lock wait. The transaction-ready-list is kept sorted in FCFS order (i.e., the transactions' arrival time), whereas each subtrans-ready-list is kept sorted in descending order by the total execution time of the subtransactions' direct and transitive successors including themselves (i.e., in MWF order). The scheduling component needs to make a decision upon different types of events, in particular, upon the EOS of a subtransaction and the BOT and EOT of a transaction. The scheduling decisions at these points are summarized in pseudo-code form in Figure 11.6.

11.7 AN APPLICATION STUDY

This section illustrates the usefulness of the PLENTY architecture with a case study of a real application. In a joint project with the Ubilab of the Union Bank of Switzerland a combined OLTP/OLAP workload from the area of the foreign exchange has been studied. The following discussion simplifies the actual application but captures major characteristics of the workload. The OLTP transaction under consideration is the so-called *currency swap* which is a major business type for foreign exchange. The OLAP transaction that we focus on

```
upon the arrival of transaction Ti:
  if number-of-idle-processors > 0
  then assign an idle processor to the BOT processing of Ti fi;

upon the EOT of transaction Ti:
  inspect the high-level lock queues and wake up all subtransactions Tjk
    that have become unblocked by the lock releases of Ti
    by placing Tjk in the subtrans-ready-list of Tj
    and placing Tj in the transaction-ready-list;
  while number-of-idle-processors > 0
  and transaction-ready-list is not exhausted do
    let Tj be the next transaction of the list;
    while number of processors in use by subtransactions of Tj < DIP limit of Tj
    and number-of-idle-processors > 0
    and subtrans-ready-list of Tj is not exhausted do
      let Tjk be the next subtransaction of the list;
      assign one processor to Tjk and decrement number-of-idle-processors;
    od
  od

upon the EOS of subtransaction Tim:
  inspect the page-level lock queues and wake up all subtransactions Tjk
    that have become unblocked by the lock releases of Tim
    by placing Tjk in the subtrans-ready-list of Tj;
  check the successors of Tjk as to whether all their predecessors are completed,
    and if so, place those successors into the subtrans-ready-list of Tj;
  while number-of-idle-processors > 0
  and transaction-ready-list is not exhausted do
    let Tj be the next transaction of the list;
    while number of processors in use by subtransactions of Tj < DIP limit of Tj
    and number-of-idle-processors > 0
    and subtrans-ready-list of Tj is not exhausted do
      let Tjk be the next subtransaction of the list;
      assign one processor to Tjk and decrement number-of-idle-processors;
    od
  od
```

Figure 11.6 Pseudo code of the scheduling component

performs a *profit analysis* which assesses the amounts of different currencies that the bank is going to hold in the future.

A swap is a deal where two trading partners exchange equivalent amounts of money in two currencies for a period of time. For example, one bank buys 1 million US Dollars (USD) for 1.5 millions Swiss Franks (SFR). At some point in the future, for example, half a year later, the second bank takes back the amount of 1 million USD for an amount of say 1.4 millions SFR where the difference to the original 1.5 millions SFR corresponds to the different interest rates of the two currencies. The first chart in Figure 11.7 shows several of these cash flows. As the bank performs many of these swap deals, the balance of the currency accounts varies over time. The resulting balance of the USD account is shown in the second chart of Figure 11.7.

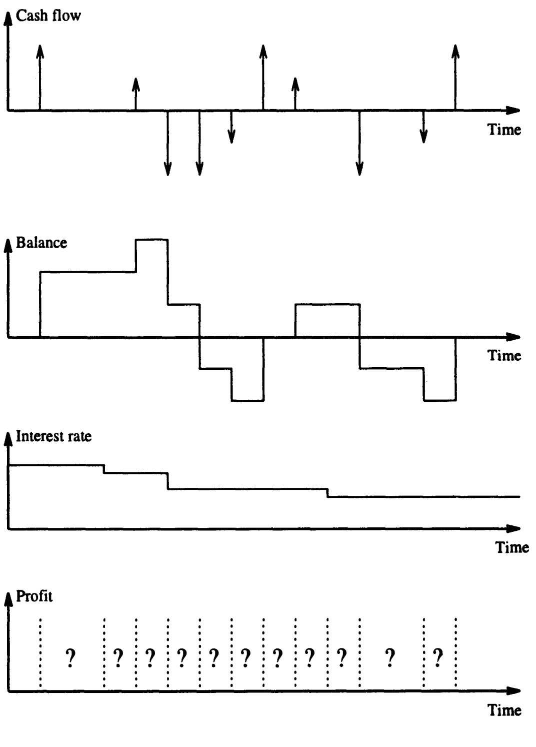


Figure 11.7 Effect of foreign-exchange transactions

Each swap corresponds to two cash flows on the same currency. Multi-level transactions are well suited for this type of application. By exploiting the counter-incrementing and -decrementing semantics of *Buy* and *Sell* operations, concurrent updates of the account balance are possible without blocking of transactions. Thus, almost all lock conflicts between swap transactions can be eliminated.

For decision support of the trading process, traders may want to assess the amount of USD the bank will hold over time. The balance of a currency is assessed by computing the profit according to the expected interest rate for this currency. Since both the account balance and the expected interest rate vary over time, this assessment is actually a computation over a time series. In simplified terms, for each day, the balance is multiplied by the interest rate to compute the daily profit. Then the daily profits are summed up to compute the total profit. This type of profit analysis is schematically depicted in the third and fourth chart of Figure 11.7. Note that the interest rate curve is actually an input parameter of the transaction; traders can assess their currency accounts with different expectations about the future trends in interest rates. The computations over time series are well suited for parallelism since the overall time interval can be split into smaller time intervals and the computations on these smaller intervals can be performed in parallel.

Since the profit analysis accesses a large amount of data, its response time may be criticial in that it serves to provide online trading support. Furthermore, because the retrieve operations on the balance values of the various currencies are even in semantic conflict with the *Buy* and *Sell* operations of the swap transactions, the profit-analysis transactions may lead to unacceptable lock contention. Thus, we have the typical problem of combining OLTP (the swap transactions) and OLAP (the profit analyses). To reconcile the two transaction types, we proceed in two steps. In the first step, intra-transaction parallelism is exploited for the profit-analysis transactions by partitioning the computation based on time intervals. This is of direct benefit for the OLAP transactions' response time, and, in addition, it reduces the duration for which retrieve locks are held. Under very high load, this step may not be sufficient to avoid lock contention. Thus, as a second step, we consider also parallelizing the swap transactions, again based on time interval partitioning for its balance updates, thus reducing the lock duration of Buy and Sell locks as well. The scripts for the two parallelized transaction types are shown in Figures 11.8 and 11.9. Note that each node of the parallel branches is actually a sequence of smaller operations each operating on one tuple of the Balance (and InterestRate) time series, and that subtransaction boundaries can be superimposed flexibly on those operations by grouping say every ten successive operations into one subtransaction.

11.8 CONCLUSION

This paper has presented a carefully designed combination of mechanisms and strategies towards better support of mixed OLTP / OLAP workloads. From an algorithmic point of view, the major contribution is the integration of multi-level transaction management with the parallelization of transaction scripts and the two-tier scheduling algorithm. From a systems point of view, we have

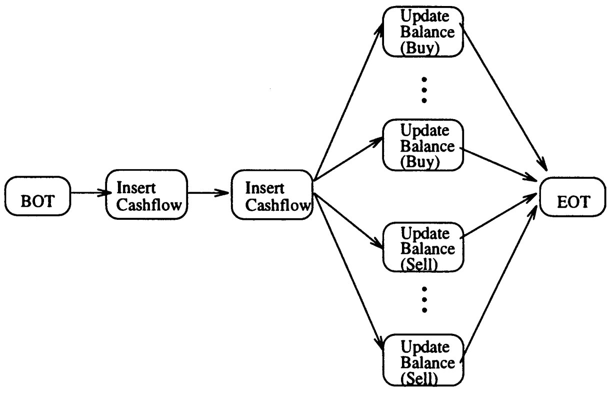


Figure 11.8 Precedence graph for the swap transactions

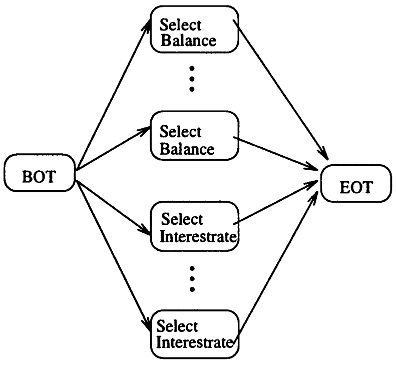


Figure 11.9 Precedence graph for the profit-analysis transactions

shown the feasibility of the approach by integrating all components in the fairly comprehensive PLENTY prototype system.

PLENTY has been aiming particularly at workloads with high data contention. Obviously, it is not the only promising approach towards reducing the adverse performance impact of locking, as we discussed in the introduction of this paper. Combinations of other techniques, especially transient versioning, with parallelized multi-level transactions are certainly worthwhile to explore. An application class where the PLENTY approach appears to be particularly beneficial is text index management, for example, for Web or intranet search engines. Inserting a new document may require hundreds of index updates that can be elegantly embedded into a multi-level transaction, thus guaranteeing a consistent view of search terms to text queries with complex search predicates [Weikum and Schek, 1992, Barbara et al., 1996b, Kamath and Ramamritham, 1996b, Kaufmann and Schek, 1996].

To some extent, PLENTY may also be a good candidate as a low-level platform to support transactional workflows [Rusinkiewicz and Sheth, 1995, Georgakopoulos et al., 1995, Mohan, 1996, Jablonski and Bussler, 1996]. Workflow management requires, of course, a much richer specification and runtime environment, but the specific problem of embedding multiple activities or invoked applications of a workflow into a transactional sphere [Leymann, 1995, Worah and Sheth, 1996] appears to be related to the mapping of the nodes of a PLENTY script onto subtransactions. Further studies along these lines are planned, within the context of the MENTOR project on enterprise-wide workflow management [Wodtke et al., 1996, Weissenfels et al., 1996] (whereas otherwise MENTOR is completely unrelated to PLENTY).

Finally, a problem area that we have merely touched on in the PLENTY project is that of tuning the resource management for complex multi-class workloads [Weikum et al., 1994, Brown et al., 1994, Rahm and Marek, 1995]. For example, we have introduced a limit for a transaction's degree of intra-transaction parallelism as a tuning knob. Ideally, such knobs should be automatically set by the system itself and dynamically adjusted depending on the current load characteristics. The COMFORT project [Weikum et al., 1994], for example, has made some modest contributions towards this ambitious goal of automatic performance tuning, and we plan to continue striving for a better analytical foundation of and effective algorithms for self-tuning information systems.

VIII Real-Time Data Management

IX Mobile Computing

12 TOWARDS DISTRIBUTED REAL-TIME CONCURRENCY AND COORDINATION CONTROL

Paul Jensen, Nandit Soparkar
and Malek Tayara

Abstract:
Concurrency control is an important issue for environments in which shared data and system resources must be managed in real-time (i.e., with implicit or explicit time constraints). The real-time responsiveness and consistency requirements, which often conflict with each other, suggest that traditional transaction processing paradigms need to be modified for target applications. The time cognizant concurrency control techniques developed for centralized systems must be extended and adapted to the distributed environment. We discuss techniques developed for real-time transaction systems for partitioning data and concurrency control to support diverse consistency and responsiveness requirements – often within the same application. Furthermore, we suggest that some of the application-specific requirements may be better met by lower level communication and coordination protocols.

12.1 INTRODUCTION

Transaction processing systems have been highly successful for the concurrent and fault-tolerant access of persistent data. In an effort to accrue similar advantages for time-critical applications and emerging technologies, suitable *concurrency control* (CC) techniques are being considered currently in the research community. There is an increasing need for real-time data management for diverse distributed applications such as groupware, manufacturing automation, air traffic control, telecommunications, etc. (e.g., see [Ellis and Gibbs, 1989, Clauer et al., 1993, Reddy et al., 1993, Lortz and Shin, 1993, Musa, 1995, Zhou et al., 1996]). These environments have shared data and common system resources, and their needs are reflected as *consistency requirements* (CR) and *real-time responsiveness* (RTR). In broad terms, RTR refers to the performance of the system being such as to meet the application requirements (e.g., meeting task deadlines or providing sufficient responsiveness for interactive environments). Similarly, the CR issues are also application-specific, and well-understood in the context of data management (e.g., consistency of values among replicas). Several studies, largely for centralized environments, suggest that the performance, functionality and applicability of data management middleware may be substantially enhanced if temporal considerations are taken into account. In fact, convergent techniques from transaction processing, distributed systems, and real-time computing (e.g., see [Soparkar et al., 1996]), are indicated for managing CR and RTR in the target applications.

The motivation for distributed real-time CC is based on several factors. First, an examination of real-life applications indicate a need for CR and RTR; several existing studies do not examine actual applications. Especially as systems evolve and become more complex, *ad hoc* techniques alone fail to suffice. Second, the target domains exhibit widely varying CR and RTR needs among and within applications. In practice, these needs are met often by a recourse to solutions that are application-specific, and which incur a high cost to maintain and prove inflexible in the long run. Third, concurrent distributed executions indicate that to ensure CR, the atomic execution of semantically coherent sequences of operations, which we refer to as transactions (see [Soparkar et al., 1996]), must be supported. Sometimes such executions are effected implicitly (e.g., to guarantee specific CR) without being referred to as transactions (e.g., see [Jensen et al., 1997]). Fourth, there is a need to provide a programming paradigm that is uniform across different techniques (e.g., including process group protocols; see [Birman, 1993]) to improve the ease of development and performance. Such an environment would also help in the scale-up and evolution of the applications.

Instances of transactions occur in almost all the distributed application domains. For example, in groupware editing of a document, a sequence of changes made to a portion of the document by one user would need to be protected from other users' actions in order to reflect the desired changes correctly. As another example, consider a situation where one user creates a picture, and a different user moves the picture to a specific location on the screen. These two activities would need to be atomic with regard to each other to ensure that an incomplete version of the picture is not moved. In some cases, even human protocols may be used to coordinate the activities, although the provision of the requisite protocols by the system would improve performance. Note that there are implicit RTR needs associated with this interactive application.

There may be situations where a distributed transaction is used to group together the actions of several users. For instance, suppose that a telescope is to be focused by several geographically dispersed users who are responsible for different controls. One user may be responsible for the horizontal movement of the device, the second may handle the vertical movement, and a third may adjust the focal length to maintain a sharp image. To keep the view observed through the device focused, the actions of the three users may be grouped together as a single transaction. Again, there may be additional timing considerations that need to be observed among the users' actions to maintain the focus. Managing distributed executions of this nature is a form of coordination control, and may be stated suitably in the form of CR and RTR requirements on the data (e.g., see [Soparkar et al., 1995b]).

The CR in target distributed applications typically require that autonomous executions from separate sites be coordinated with one another. For this purpose, the two widely used approaches are: the transaction model (e.g., see [Bernstein et al., 1987]) and the process group model (e.g., see [Cheriton and Zwaenepoel, 1985, Birman and Joseph, 1987b]). The transaction model has techniques for ordering concurrent transactions and ensuring their atomicity. Process groups are characterized by modeling the distributed system as a collection of processes that communicate by multicasting messages. Again, techniques are provided for ordering concurrent messages as well as ensuring the atomicity of message delivery. These two different approaches share these common aspects that simplify development of distributed applications. In fact, at the level of protocol implementation, they appear similar (e.g., see [Guerraoui and Schiper, 1994, Jensen et al., 1997]).

It has been observed that neither traditional transaction processing, nor traditional distributed computing techniques, are suited to all applications uniformly. Some applications are better suited to transactions, whereas others are better coordinated by group multicast techniques. Generally, transactions are better suited to cases where a high degree of CR is needed, whereas process group approaches are better in terms of RTR. Most distributed applications could benefit from a good meld of the performance and consistency criteria (e.g., see [Birman, 1994, Cheriton and Skeen, 1993]). We argue that existing techniques from several domains, complemented by new ones, will eventually emerge as a set of appropriate approaches for distributed real-time data management.

There is an inherent difficulty in meeting the RTR and CR simultaneously, and this has been identified in transaction processing as well as distributed real-time systems (e.g., see [Soparkar et al., 1994]). Therefore, there is need to exploit application-specific semantics and to use alternative means to obtain CR guarantees while achieving better RTR. We examine how techniques developed for real-time CC may be applied effectively in distributed environments. We discuss a logical design to segregate the data and CC into distinct levels such that the RTR and CR issues within a level are addressed in a uniform manner, whereas the issues may differ from the other levels. Furthermore, we discuss the use of distributed middleware protocols which may be more efficient, and accrue similar advantages, when compared with transaction-oriented system.

12.2 RESPONSIVENESS AND CONSISTENCY

An examination of different applications illustrates the differing needs of RTR and CR – sometimes within the same application. Simple working descriptions of the RTR and CR for the applications suffice for our discussions. For instance, in groupware environments, RTR may mean a responsiveness which closely approximates a shared workspace for physically co-located users. On the other hand, for a manufacturing environment, RTR could reflect the close temporal coupling of separate machines which need to be coordinated.

In the examples to follow (parts of which appear in [Jensen and Soparkar, 1995]), it is not difficult to state suitable RTR and CR needs associated with the data (as in [Soparkar et al., 1995b]). An example may be to require that distributed replicas of a data item not be mutually inconsistent longer than some period. In turn, this would impose RTR needs on the executions that affect the values of the data items.

12.2.1 Simple Concurrency Control

There are several applications requiring basic real-time control to manage access to shared resources. For instance, in groupware, consider a shared document being edited by several users simultaneously. The RTR suggests that changes being effected by a user should be made visible locally to the user immediately, and also, be propagated to the other users. However, if a remote user simultaneously makes conflicting changes (local to the remote location), then a CR problem may arise in terms of the contents of the document. On the other hand, if an attempt were made to ensure a consistent document across all locations (e.g., by using exclusive locks), then the CC itself may cause inadequate RTR (e.g., due to the time-consuming acquisition of remote locks).

Groupware applications may also exhibit characteristics which permit greater RTR because their CR considerations are less crucial (e.g., as compared to a database environment). Consider data that represents the "presentation" services in groupware (e.g., a shared pointer, or a shared view of a document). The utility of such data lies in providing "instantaneous" interaction among users. However, it may not matter much if, occasionally, a shared pointer or a shared view is not synchronized for some users for a brief period. Clearly, such situations reflect relaxed CR.

12.2.2 More Elaborate Coordination Control

Distributed executions may involve more complicated control. For example, consider automating the task of lifting and cutting a pipe in manufacturing. Two robots may lift the pipe, and other robots may cut the pipe. Separate autonomous controllers for each robot are often needed for reasons of modularity, and these controllers must be coordinated with one another. There may be several constraints among the robots as regards their actions. For instance, the lifting robots may need to act in close synchrony, and also, lifting must be accomplished prior to the cutting (e.g.. see [Shin, 1991]). Such requirements, translated into RTR and CR constraints, could be managed by the use of CC or low-level communication primitives. The latter helps in achieving RTR more easily, and may be able to meet time constraints better. For instance, the use of totally ordered multicasts (e.g., see [Birman, 1993]) invoked from within the application programs, could help in synchronizing the actions efficiently.

More elaborate coordination is exhibited in considering an observation system in scientific domains for mobile objects (adapted from [Soparkar et al., 1994]). The system consists of several tracking stations (i.e., sites), each of which has its own computing and scheduling resources. That is, there are several processing sites that manage object-sensors, cameras, and locally store data pertaining to the readings, positions, etc.

Periodically, the sensors update the data regarding the objects as tracked at each local site, and this data is also sent to specific coordinator sites. The coordinator receives track data from several sites and correlates the data gathered to create the global tracking information. It is necessary to do the correlation since the data obtained at each site may be individually insufficient to identify the objects accurately. The globally correlated data is also disseminated among the sites, and this data affects local decisions at the sites. Finally, global decisions may be taken sporadically for a variety of actions to be executed simultaneously among several sites. For instance, cameras may be activated to take photographs of a particular object from several angles at a particular time, to be then provided to the users.

Assume that at each site, local transactions update the local track data. Also, assume that the collection and correlation of the local track data from the different sites, and the dissemination of the global track data, together constitute one type of distributed transaction. The reading of the local track data and subsequent writing of the global track data at each site constitute the local subtransaction for the distributed transaction.

Suppose that an erroneous local track is recorded at one of the locations – perhaps due to a malfunctioning sensor. This fault may be detected only after the local track data is collected and correlated with (correct) track data from other sites (but before the corresponding global track is committed). Consequently, erroneous global track data may be generated and disseminated to several sites. Such a distributed transaction should be aborted as soon as possible. In standard transaction processing, the execution of a commit protocol ensures that all the subtransactions of the aborted distributed transaction do indeed abort.

The price paid for employing a standard commit protocol may be high. Blocking may cause a situation where none of the sites have recent global track data, and awaiting the coordinator's final decision may unnecessarily cause poor RTR. The fast local commit of a subtransaction would be much more suitable – optimistically assuming that the distributed transactions usually commit. However, uncoordinated local commitment may cause some sites to commit the erroneous global track data they receive, and subsequently to expose the data to other transactions. For instance, a transaction that positions the camera at a site may base its computation on the prematurely committed, and hence inaccurate, global track data. Therefore, there is a need to recover from the effects of the incorrectly committed data by compensatory actions. In our example, the compensatory actions may re-position the camera based on the past history of the execution in an application-specific manner.

12.3 ENABLING TECHNOLOGIES

In the context of distributed real-time data management, a coherent picture is not available, nor is there consensus regarding the accepted paradigms. The primary reasons are the difficult issue of addressing simultaneously the often conflicting demands of CR and RTR.

In contrast, centralized real-time CC, which is a special case of distributed real-time data management, is well-studied. Due to the inherent architecture of distributed systems, we advocate an approach that begins with centralized real-time CC, and thereafter, addresses distributed control. That is, each site in a distributed setting should largely function autonomously with its own local CC, and coordination be effected among the multiple sites for managing the distributed executions. Therefore, the local CC module must handle the local CR as well as the RTR for the local executions, and coordinating agents should manage the interaction among sites. Several studies have advocated this general approach (e.g., see [Son, 1988, Soparkar and Ramamritham, 1996, Son, 1996, Ozsoyoglu and Snodgrass, 1995, Kao and Garcia-Molina, 1992, Graham, 1992]). In consequence, there is need to understand both the centralized as well as the distributed scheduling issues.

12.3.1 Centralized CC for RTR

In an environment with RTR needs for transactions, all executions may not be desirable even if they happen to meet the CR (e.g., by being serializable). In fact, a schedule where all transactions meet their deadlines may be better than another where some deadlines are not met – even if the latter schedule has a shorter overall execution time. It is in trying to meet the RTR that traditional CC must be modified. The general approach adopted in centralized real-time CC is to generate schedules that are optimized for RTR in various ways (e.g., see [Graham, 1992, Ramamritham, 1993, Ozsoyoglu and Snodgrass, 1995, Kao and Garcia-Molina, 1992]). By suitably delaying or aborting input operations, better RTR schedules may be effected, and most of the techniques are described as such.

Re-examining traditional CC indicates that merely improving concurrency is of limited utility (e.g., see [Soparkar et al., 1996]). The traditional approach has the goal of certifying as large a number of schedules as possible that meet the CR (e.g., see [Papadimitriou, 1986]). The expectation is that this would enable the CC to adversely affect the performance minimally, and therefore, every schedule meeting the CR is regarded as being equally desirable. Therefore, the goal of traditional CC is to preserve the CR (i.e., generate logically correct executions) while maintaining a high level of parallelism (i.e., the number of allowable executions). Clearly, traditional CC schedulers do not attempt to provide better RTR explicitly. That is, it is assumed implicitly that the order of the input sequence of operations will provide good RTR, and to the extent possible, the sequence is output unchanged. In fact, allowing a large number of schedules implies that schedules exhibiting *worse* RTR may also be generated – although, it also allows for a wider choice to find better performing schedules.

12.3.2 Characterization Efforts

Surveys (e.g., see [Ramamritham, 1993, Ozsoyoglu and Snodgrass, 1995, Kao and Garcia-Molina, 1992]) indicate efforts have focused on applying real-time scheduling to resolve conflicts that arise among transactions in order to meet the RTR needs. Various combinations of real-time and CC techniques have been considered, and "rules-of-thumb" have been identified and studied empirically.

In order for the CC to meet the RTR, some efforts attempt to relax the CR (e.g., see [Korth et al., 1990a, Soparkar et al., 1995b, Kuo and Mok, 1992]). These approaches rely on the performance benefits of increased concurrency over traditional CR. For instance, [Lin et al., 1992] illustrates specific situations in which non-traditional CR may help meet RTR needs, and where the durability of transaction executions may not be essential. However, such an approach is obviously limited in meeting CR in general settings: indeed, discussions in [Graham, 1992] emphasize the importance of traditional CR even in typical real-time computing environments.

The characterization efforts for distributed cases are few (e.g., see [Ramamritham, 1993, Soparkar et al., 1995a, Soparkar et al., 1994, Purimetla et al., 1995, Ulusoy, 1992]); the difficulties lie in characterizing the CR and RTR in distributed environments, and the lack of solutions for issues such as synchronization and atomic commitment (e.g., see [Soparkar and Ramamritham, 1996]). While it may be argued that centralized real-time CC should be completely understood first, it remains a fact that most of the target applications are inherently distributed.

12.3.3 Performance Studies

Most of the existing research results in real-time CC have been in the context of performance studies (e.g., see [Ramamritham, 1993, Lee and Son, 1995, Ozsoyoglu and Snodgrass, 1995, Kao and Garcia-Molina, 1992]). Typically, different work sets, loads on the system, and fairly innovative techniques for CC are examined by running extensive simulations, and conclusions are drawn from the empirical results. Several locking or time-stamping CC protocols are examined in conjunction with heuristic preemption policies. Often, optimistic approaches to CC have been found to perform favorably in certain situations (e.g., see [Haritsa et al., 1990, Huang et al., 1991]). In general, the data access patterns are assumed unknown *a priori*, although some simulation experiments do take into account the patterns – with the expected improvement in RTR. A few studies have used analytic models, such as [Song and Liu, 1990], in which an application to monitor certain real-time systems situations using lock-based multi-version CC protocols is studied. Similarly, the inclusion of priorities in the handling of different system resources such as the processors, and buffers, for the transactions, is studied analytically in [Carey et al., 1989]. However, real-life applications are often not considered for the workloads, platforms etc., and furthermore, usually only a single RTR factor is addressed.

A useful development for improved RTR is the availability of main-memory resident database systems (e.g., see [Garcia-Molina and Salem, 1992]). These have arisen due to increased memory sizes and smaller data requirements in new application domains. Since the number of disk accesses is reduced, and data may be made durable selectively, transactional accesses to the data become more efficient. In real-time environments, this is particularly useful since the validity of a significant part of the data is time-bound, and therefore, there is little utility in making such data permanently durable.

12.3.4 Real-time and Distributed Systems

Scheduling theory (e.g., see [Lawler et al., 1992]) indicates that even for relatively simple situations, the issue of guaranteeing RTR is computationally expensive. The issue of ensuring CR only exacerbates this problem (e.g., see [Soparkar et al., 1995b]). Coupled with uncertain execution times, multi-tasked environments with preemption, varying RTR needs, and unexpected delays etc., these problems indicate that in pragmatic terms, relying on heuristics for scheduling is unavoidable in all but very simple cases.

In distributed environments, the problems in meeting RTR and CR needs are more difficult. Typically, distributed computing approaches let these issues be handled by the application designers or users, and provide them some tools to do so. Of note are the use of multicast message ordering schemes (e.g., see [Birman, 1993]) which forego transactional CR in many simple instances, and thereby achieve higher RTR. The approach efficiently implements transaction-like primitives in a communication subsystem for common modes of multicasts. However, as CR and RTR demands grow, obtaining less stringent orderings become increasingly difficult, and transactional semantics appear to be necessary (e.g., see [Jensen et al., 1997]). In general, where it is possible to circumvent the inefficiencies that affect RTR in transaction-oriented computing, group communication protocols should be supported for simpler distributed interactions.

12.3.5 Application-specific Approaches

Approaches specific to particular applications are narrowly focused. As such, they are similar to the implement-test-re-implement cycles prevalent in unstructured real-time system development (e.g., see [Stankovic, 1988]). In the interests of short-term costs, simple-minded approaches are used to develop working systems which, unfortunately, often do not scale-up or evolve well. They usually have a poor characterization of the CR needs, and RTR is often "handled" by upgrading to the next higher-speed processing environment. These approaches do not work in more complex environments due to the CR needs.

Instances of application-specific approaches include meeting of RTR on transactions that procure approximate information from a database (e.g., see [Smith and Liu, 1989]). The idea explored has been to improve the required estimates that are gathered depending on the remaining time available. Another approach specific to a few real-time control environments is described in [Kuo and Mok, 1992] which considers liberal CR. Similarly, for specific cases, the need for CC may be entirely avoided as described in [Audsley et al., 1991].

12.4 LOGICAL SYSTEM ARCHITECTURE

We describe a generic distributed real-time CC architecture which could be suitably modified to represent actual cases. Following [Soparkar et al., 1996], a distributed real-time system architecture consists of n sites, $S_1, S_2, \ldots, S_n$, interconnected by a communications network as shown in Figure 12.1. Each site S_i has a largely autonomous local system which must meet the local CR and RTR needs. The local CC, should not distinguish between transactions and subtransactions. The data, which may be replicated, is stored in a local database. The coordination effort is distributed among the sites in the form of n interconnected software agents which are independent with regard to one another. All

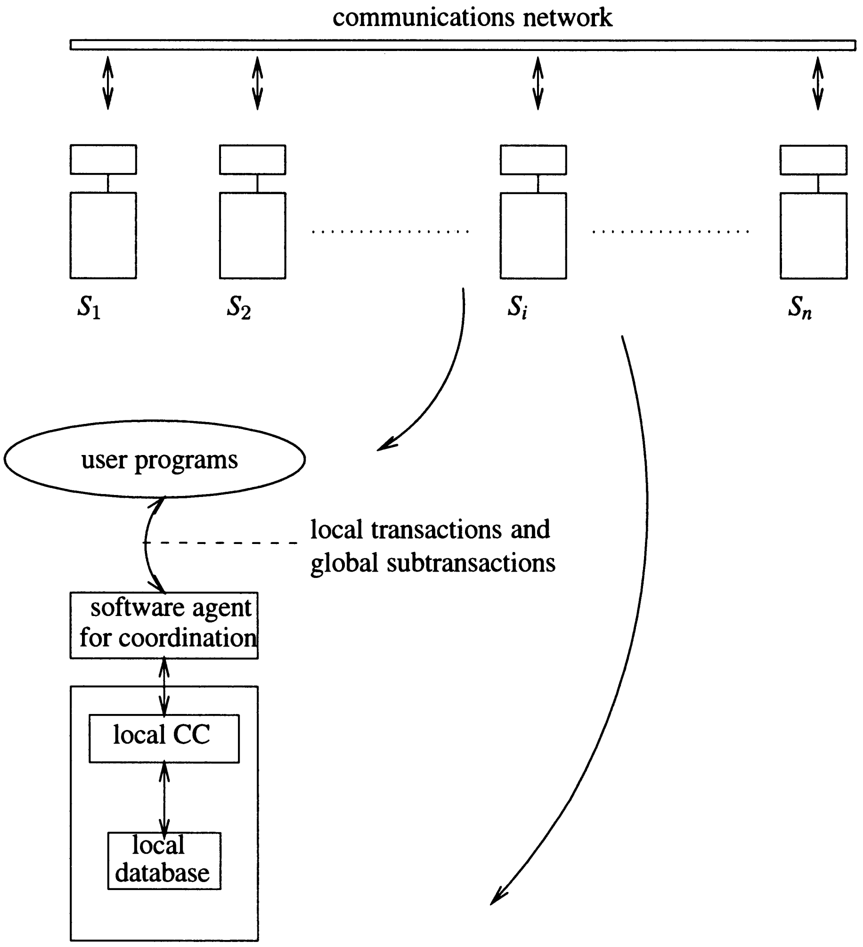


Figure 12.1 Distributed real-time system architecture

interaction among the sites, including the synchronization, is managed by the software agents.

We regard each database as a set of entities, and the data is persistent in that it may have lifetimes greater than the processes that access it. The state of a database is a mapping from entities to their corresponding values. A transaction is a data accessing part of an application program, and may be regarded as a sequence of operations performing a semantically coherent task. In fact, a transaction is the unit for the consistent access of the database as well as the unit for recovery in case of an abort. A consistent access is left unspecified except that it should reflect the application semantics. Similarly, the CR of the concurrent executions is regarded as being application-specific.

12.4.1 Providing RTR at Local Sites

For providing RTR in centralized environments, the basic approaches incorporate real-time scheduling techniques with CC. Such approaches work well for specific RTR and workloads, but not necessarily for general cases. Often, they are difficult to implement since the CC interacts with lower levels of the operating system, and are relatively inflexible. There are several are several surveys (e.g., see [Ramamritham, 1993, Kao and Garcia-Molina, 1992, Ozsoyoglu and Snodgrass, 1995]) that describe the various available techniques and studies.

A simple alternative may be described as follows. First, note that one reason why the traditional CC does not work well for RTR is that it has only two choices with regard to an input operation – either to schedule it, or to delay it. That is because traditional CC schedules one operation at a time. Instead, we suggest scheduling a *set* of input operations at a time – thereby providing the potential to apply scheduling heuristics for RTR to the operations (e.g., [Soparkar et al., 1996]). Second, we propose addressing the RTR *prior to* ensuring the CR (by using CC after the real-time scheduling). This would imply that a change in the CC *per se* is not needed, and yet would allow schedulers to handle different RTR needs with the *same* CC. This flexibility in our approach may outweigh potential disadvantages of not being able to provide RTR to the same degree as that of a combined CR and RTR approach.

12.4.2 Distributed Real-time CC

Executions that access data at a particular site include local transactions as well as subtransactions from distributed transactions. For the granularities of time in the target applications, it is acceptable to assume that the local clocks across the separate sites are well-synchronized. We suggest the use of synchronization protocols across the sites to manage distributed executions. Note that CR (e.g., serializability) may be guaranteed using techniques available in federated database systems (e.g., see [Soparkar et al., 1991, Mehrotra et al., 1992]).

It is also possible to use lower-level communication primitives (e.g., suitable forms of multicast orderings) to synchronize the distributed executions (e.g., see [Birman, 1993, Birman and Joseph, 1987a, Garcia-Molina and Spauster, 1991]). Such approaches may require an effort by the users, or the application program itself, to initiate the needed synchronization and coordination.

12.4.3 Levels with Differing Requirements

As discussed in the examples, there are varying RTR and CR needs – sometimes within the same application. One approach to this problem is to find ways to categorize and isolate the particular needs, and provide distinct mechanisms to meet varying requirements and which may be used within one application, (e.g., see categorized transactions of [Purimetla et al., 1995, Kim and Son, 1995]). Following along these lines, Figure 12.2 depicts a logical architecture for the separation of data and CC into levels in a distributed environment. Sites $S_1, S_2, \ldots, S_n$ represent the separate locations, and at any given site S_i, the levels A_i, B_i, and U_i, represent three disjoint data sets, with different RTR and CR requirements. The data within a given level across the sites (e.g., $A_1, A_2, \ldots, A_n$) may have various CR constraints among them (e.g., in terms of replication, the constraint would be "equality"). It is certainly possible to have several more levels as necessary for a given application environment.

12.4.3.1 Level *A*. This level corresponds to high RTR and low CR. For example, data associated with presentation services in groupware may fall into this level: it may be acceptable for users to occasionally see an action being done – and then undone – due to conflicts with other actions.

As an example, in a situation where several scientists are together studying a geophysical terrain pictured on their computers, a single pointer on the screen may be used to draw the attention of the entire team to particular points of interest. Since the team may be involved in discussions on what they observe, the pointer manipulations would need to be effected in real-time. Furthermore, conflicts arising due to the simultaneous access of the pointer may be resolved in a simple manner (e.g., by restoring the pointer to its original position, by giving one movement preference over others, etc.).

12.4.3.2 Level *B*. This level corresponds to high CR, even if RTR is relatively poor – such as in the case of standard, conservative CC techniques (e.g., from database management systems).

For example, in manufacturing automation, situations may demand that a distributed set of actions be effected only if the set is guaranteed to be committed. Consider the setting of parameter values for the operation of several machine tool units. This may involve intricate and time-consuming changes to be made. It would be costly to repeat that work in case conflicting changes invoked by other tasks prevents the intended changes from being effected. In such environments, it may be worthwhile for the CC to ensure first that any changes reflected would indeed be effected (and not "aborted" due to other concurrent activities). This implies that the required exclusive access to the relevant data and resources must be ensured.

12.4.3.3 Level *U*. This level corresponds to user or application managed CC with varying RTR and CR requirements. Application-specific semantics may be incorporated within the application program code itself.

For example, group communication protocols, voice or video stream synchronization, and interactions among the levels described, may be effected within level U. Also, advanced semantics-based CC, such as compensating transactions (e.g., see [Elmagarmid, 1992, Levy, 1991, Soparkar et al., 1994]),

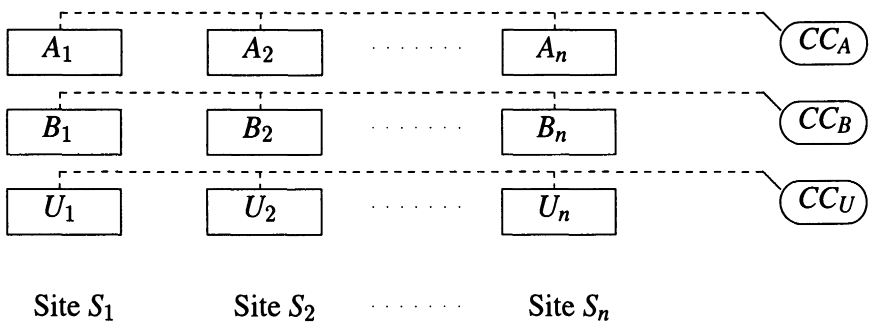


Figure 12.2 Logical system structure with levels.

may be incorporated into this level. Therefore, Level U is expected to manage shared data and resources in an environment that exhibits both RTR and CR in varying degrees. The approaches to CC for this level are likely to be very challenging.

The levels A and B correspond to simple concurrency control, whereas the elaborate distributed coordination examples would belong to level U.

12.5 SYNCHRONIZATION USING APPLICATION SEMANTICS

While it is advisable to use the standard CR where possible in order to accrue their obvious advantage (e.g., see [Gray and Reuter, 1993]), they pose problems for RTR. Ensuring CR in a distributed execution may use synchronization mechanisms such as distributed commitment (e.g., the two-phase commit protocol – see [Bernstein et al., 1987]). If for any reason a site does not obtain the final message for the protocol, the execution in question may be blocked until the necessary message is received – leading to poor RTR. A different problem that may arise is that the local CC may dictate that a subtransaction be aborted in favor of others with higher RTR priorities, and that may be impossible to achieve for similar reasons. Therefore, some of the stringent CR must be relaxed (e.g., see [Singhal, 1988, Stankovic, 1988]), and this may be achieved by taking recourse to application semantics.

As example approaches which may be used in Level U, we describe the concept of relaxed atomicity from [Soparkar et al., 1994], and discuss communication level primitives. The provision of several such tools to the application designers and users would allow their use as and when indicated by the applications. This is similar to the concept of "unbundling" services to be used as required in the context of real-time database technology (e.g., see [Soparkar and Ramamritham, 1996]).

12.5.1 Relaxed Atomicity

A compensating transaction is a recovery transaction that is associated with a specific forward transaction that is committed, and whose effects must be undone. The purpose of compensation is to "undo" a forward transaction semantically without causing cascading aborts. Compensation guarantees that CR is established based on application semantics. The state of the database after compensation takes place may only approximate the state that would have been reached, had the forward transaction never been executed (e.g., see [Korth et al., 1990b, Levy, 1991]).

A distributed transaction may be regarded as a collection of local subtransactions, each of which performs a semantically coherent task at a single site. The subtransactions are selected from a well-defined library of routines at each site. For distributed transactions that can be compensated-for, each forward subtransaction is associated with a predefined compensating subtransaction. Compensating for a distributed transaction need not be coordinated as a global activity (e.g., see [Levy et al., 1991b, Levy, 1991, Levy et al., 1991a]). Consequently, the compensating subtransactions are assumed to have no interdependencies, share no global information, and to not need the use of a commit protocol (i.e., local sites run the compensations autonomously).

An adaptive strategy may be described that assures "semantic" atomicity as a contingency measure. The idea is that when blocking becomes imminent, sites should decide locally to switch from a standard commit protocol to the optimistic version. In the event that the global coordinator decides to abort the entire transaction, compensating executions may be executed at each site where the subtransactions in question were locally committed.

This strategy provides a means to deal with a fast approaching RTR deadline for a subtransaction executing at a particular site. If the site in question has not yet sent a message indicating preparedness to commit to the coordinator, then it may be unilaterally aborted. On the other hand, if that message has already been sent, then the subtransaction may be optimistically committed – the expectation being that the final decision for a distributed transaction is usually to commit.

A simple example of compensatory actions may be described in the context of groupware editing. A user may make certain changes to one part of a document, and may optimistically commit the changes rather than to await confirmation from other users that it is safe to do so. Occasionally, there may occur an incorrectly committed change, and thereupon, a user-specified compensatory action may be taken to restore a state that meets application-specific CR (e.g., see [Prakash and Knister, 1994]).

12.5.2 Communication Level Approaches

Transactional techniques for providing CR in distributed environments have proven useful for their simplicity from a usage perspective. However, relaxed CR in applications suggest communication level approaches, such as use of multicasts. In [Jensen et al., 1997], a framework is developed which applies CC theory to to multicast techniques. The framework provides a better understanding of the manner in which CR for distributed applications may be maintained. This understanding also leads to more efficient multicast techniques which incorporate application semantics. Since it is difficult to state CR for different applications explicitly (as in database transaction systems), a sequential message ordering is regarded as being correct by definition. Thereafter, based on the manner in which events can commute in a distributed history, the correctness of other concurrent, non-sequential histories is exhibited. The commutativity of events is derived from application semantics (in a similar manner to non-conflicting operations in CC theory).

While researchers are analyzing real-time multicast techniques, effective protocols are also being developed. In [Kopetz and Grunsteidl, 1993, Abdelzaher et al., 1996], protocols are described which provide bounded time message transport make them suitable for applications with specific RTR needs. Furthermore, the ordering and atomicity provided for multicast messages make them useful in effecting transaction-like CR.

12.6 CONCLUSIONS

We have considered several concurrency and coordination control issues for distributed real-time data management. In discussing developments that may serve as appropriate approaches, we have provided an architectural framework for concurrency control mechanisms for use in such systems. Our approach partitions the data and concurrency control logically into levels based on the different real-time responsiveness and consistency requirements within applications. Within this general framework, we have indicated how several existing traditional and newly developed techniques may be used to satisfy the desired application requirements. In particular, systems which provide a number of user-managed alternatives to standard concurrency control are in a good position to handle diverse needs. Experience from research with experimental systems suggests that our approach may be profitably used in managing concurrent executions in these environments.

13 TRANSACTION PROCESSING IN BROADCAST DISK ENVIRONMENTS

Jayavel Shanmugasundaram, Arvind Nithrakashyap,
Jitendra Padhye, Rajendran Sivasankaran,
Ming Xiong and Krithi Ramamritham

Abstract: An important limitation in broadcast disk environments is the low bandwidth available for clients to communicate with servers. Whereas advanced applications in such environments do need transaction processing capabilities, given the asymmetric communication bandwidth, we show that serializability is too restrictive in such environments and hence propose a weaker alternative. Specifically, this paper considers the execution of updates under transactional semantics in broadcast disk environments, and develops a weaker correctness criterion. While this is applicable to transaction processing in general, this paper describes mechanisms to achieve this criterion in broadcast disk environments. We show that read-only transactions need not contact the server and can just read consistent data "off the air" without, for example, obtaining locks. Update transactions, however, need to validate their updates at the server.

13.1 INTRODUCTION

Mobile computing systems [Imielinski and Badrinath, 1994] are becoming a reality. The limitations in bandwidth, storage capacity and power of these mobile systems pose significant research challenges to the computer science community. As mobile computing systems continue to evolve, they will be used to run sophisticated applications, which in the past, were used only on "stationary" computing systems. Many of these applications will require transaction processing involving large databases. The performance of the transaction manager is the key to the performance of any large database management system. A large amount of research has gone into the development of efficient transaction management schemes [Bernstein et al., 1987]. Given that mobile computing is an emerging technology, it is not surprising that little research has been done on supporting transaction processing in such environments. In this chapter, we address transaction processing issues in mobile computing environments, particularly those based on a data broadcasting approach exemplified in [Zdonik et al., 1994].

A broadcast disk is an abstraction of a data-dissemination based system for wireless communication environments. The server periodically broadcasts values of certain data items deemed to be of interest to (a subset of) its clients. Hence the clients can view the broadcast medium as a disk from which they read the data they need. Also, the server can simulate multiple disks with different speeds by broadcasting certain data items more frequently than others. An important consideration in such an environment is the limited amount of bandwidth available for clients to communicate with the server. This bandwidth limitation prevents the transactions at the clients from contacting the server for locking data or validating its updates. Hence, providing support for transactions in such an environment is a challenging research issue.

[Herman et al., 1987] discuss transactional support in an asymmetric bandwidth environment. However, they use serializability as the correctness criterion, which we show is very expensive to achieve in such environments. In [Acharya et al., 1996], the authors discuss the tradeoffs between currency of data and performance issues when some of the broadcast data items are updated by processes running on the server. However, no transactional support is present either at the server or at the clients. The updates are made only by processes running on the server, while the processes on clients are assumed to be read-only. However, the following examples show that the clients may also need to update the data:

- Next generation road traffic management systems will make significant use of broadcast databases. These systems will store and broadcast information about traffic conditions, weather forecasts and driver advisories. The information will be used by drivers and possibly even autonomous vehicles to select the optimal route to their destinations. The data will be updated by various sources – special vehicles traveling on various important roads and gathering traffic data in "real time", by satellites and computers monitoring weather conditions and by law enforcement agencies responding to accidents or other emergencies. Since it is essential that the driver (or any querying entity) be presented with a consistent picture of the traffic conditions, transaction semantics will be useful for the query and update operations.

- Broadcast databases may also be used to facilitate operations in large, mostly "robotic" industrial plants. Data gathered from various sensors around the plant about conditions on the factory floor are broadcast to various operators and robots. For example, service personnel, equipped with mobile workstations, can access this data to carry out their work more effectively. As in the traffic management example, note that the service engineer must access sensor data values that are mutually consistent. This requires enforcement of transactional semantics on various update and query operations.

- Consider a server that stores stock trading data for a large stock market. The server continuously broadcasts information about current prices and trading volumes of various financial instruments, current values of various market indexes, and similar data items. Some of the clients (brokers, stock traders, market regulators) use mobile workstations to access this data and perform a wide variety of financial transactions. Since it is important to keep data (e.g. stock trades or liability of a broker) consistent at both server and clients, operations performed by clients can benefit from transactional semantics.

Motivated by these examples, in this chapter we propose a new correctness criterion for transaction processing in broadcast disk environments where the clients can also perform updates on the database. We also describe mechanisms and protocols to ensure correctness according to the new criterion. With the proposed correctness criterion and the mechanisms for transaction processing, read-only transactions running on mobile clients are always able to read consistent values without contacting the server (to acquire locks or to validate their reads), i.e., they will be able to read data "off the air". Two protocols are proposed for update transactions in clients. The first is a hybrid approach where the transactions at mobile clients contact the server for write locks and for validation of their reads, at the time of commit. This combines aspects from optimistic and pessimistic concurrency control protocols. The other protocol is similar to the conventional optimistic concurrency control [Bernstein et al., 1987] protocol where the transactions contact the server for validation of their data access at the time of commit.

The outline of this chapter is as follows. In Section 13.2, we motivate the need for a new correctness criterion in broadcast disk environments with examples. In Section 13.3 we propose some correctness and consistency requirements from the perspective of a user and show their relationship to serializability. In Section 13.4 we discuss the weakening of these requirements. In Section 13.5 we outline the mechanisms and protocols that are required to ensure consistency according to the requirements in Section 13.4. We conclude with an outline of future work in Section in 13.6.

13.2 MOTIVATION FOR WEAKENING SERIALIZABILITY

In this section, it is argued that using serializability as the correctness criterion for transaction processing might be too restrictive in broadcast disk environments. Two justifications are provided for the above claim. First, it is shown that achieving serializability would be very expensive in broadcast disk environments. We then illustrate, using some examples, that traditional (conflict) serializability is not always necessary and that users might be satisfied with a weaker correctness criterion. The correctness criterion alluded to here is in fact weaker than view serializability [Papadimitriou, 1988], as will be shown in later sections.

The main problem in ensuring serializability in broadcast disk environments is the fact that serializability is a global property, i.e., a property involving all the transactions accessing shared data items. Because of this, transactions running at a client either have to communicate local information to the server and/or other clients or the transactions have to ensure that local operations performed do not lead to non-serializable schedules. The first alternative involves expensive communication by the client while the second alternative may lead to unnecessary aborts. An example which illustrates the basic problem is given below.

Example 1. Assume that in broadcast disk environment, clients only know the local transaction execution history and history of updates at the server. Consider two read transactions T_1 and T_3 at two different clients A and B respectively and two transactions T_2 and T_4 which run at the server. Now consider the following execution history:

$$r_1(x);\ w_2(x);\ c_2;\ r_3(x);\ r_3(y);\ w_4(y);\ c_4;\ r_1(y) \qquad (13.1)$$

If transactions running on clients do not inform the server about the operations performed by them, then the server would only be aware of the history,

$$w_2(x);\ c_2;\ w_4(y);\ c_4$$

Client A would be aware of the history,

$$r_1(x);\ w_2(x);\ c_2;\ w_4(y);\ c_4;\ r_1(y)$$

and Client B would be aware of the history,

$$w_2(x);\ c_2;\ r_3(x);\ r_3(y);\ w_4(y);\ c_4$$

If both T_1 and T_3 commit, then the server and both the clients would see serializable histories. However, the global history would not be serializable. Thus, either T_1 or T_3 would have to be aborted. However, since the operations performed by T_1 and T_3 are not communicated, and assuming that there exists

$$\begin{tikzcd} T1 \arrow[r] & T2 \arrow[r] & T3 \arrow[r] & T4 \arrow[lll, bend right] \end{tikzcd}$$

(a)

$$\begin{tikzcd} T2 \arrow[r] & T3 \end{tikzcd}$$

(b)

$$\begin{tikzcd} T4 \arrow[r] & T1 \arrow[r] & T2 \end{tikzcd}$$

(c)

Figure 13.1 Serialization Graphs

no way to inform clients T_1 and T_3 except by expensive message passing, *both* T_1 and T_3 would have to be aborted. This is wasteful by itself since the abortion of either T_1 or T_3 would have ensured a serializable history. Unnecessary aborts would also occur if the execution history is:

$$r_1(x);\ w_2(x);\ c_2;\ w_4(y);\ c_4;\ r_1(y) \qquad (13.2)$$

This is because client A is aware of the above history and would not be able to distinguish it from the previous case. Thus, in this case too, T_1 would have to be aborted. A similar argument can be made for T_3. Essentially, in the absence of communication from read-only transactions to the server, to preserve serializability, the read-only transactions will have to be aborted even in cases like history 13.2 assuming the worst case scenario as in history 13.1 . The above examples illustrate that serializability in broadcast disk environments is either very expensive to achieve or would involve unnecessary aborts.

In the rest of the section, we illustrate via examples how serializability can be weakened while still giving intuitively correct histories. Our correctness notion, although weaker than the traditional serializability, still maintains consistency of the database and of the values read by transactions. Despite the fact that such correct executions are possible even in non-mobile environments, it is especially important to allow such executions in mobile environments because of the high cost of acquiring locks or validating transactions.

Example 2. Consider the following history of transaction execution for transactions T_1, T_2, T_3 and T_4, where T_1 and T_3 are transactions which perform only read operations in the history. Data items x and y are accessed by these transactions. Note that this example is the same as Example 1 with the exception of commit c_1 of T_1.

$$r_1(x);\ w_2(x);\ c_2;\ r_3(x);\ r_3(y);\ c_3;\ w_4(y);\ c_4;\ r_1(y);\ c_1$$

This history is not serializable since there is a cycle in Figure 13.1(a). However, the final state of the database after this execution is exactly the same as that of the serial execution of T_2T_4, the two transactions which perform updates in the history. That is, the final state of the database is consistent.

If the history is considered up to time c_3, the serialization graph for committed transactions is depicted in Figure 13.1(b). At this point, T_3 can be committed without any inconsistency, since T_3 views a consistent database and the database state at this point is also consistent. Further, at the point of c_1, a serialization graph can be formed as in Figure 13.1(c) with T_1 and all the committed update transactions interacting with T_1. The operations of read transaction T_3 are removed from the history when the serialization graph is constructed since T_1 can not see any of the effects of T_3. Therefore, in Figure 13.1(c), T_1 views a serializable history with the history formed by the operations from update transactions and itself. Again, at this point, the state of the database is consistent.

In the above example, although the global history is not serializable, each read transaction reads consistent values and views some serial order that is consistent. Since update transactions are committed at the server, they can be guaranteed to be serializable. On the other hand, read-only transactions are executed at the client. They only view committed data values broadcast by the server and since these data values are consistent, clients can commit without producing any inconsistency in their outputs. The crux of this weakened criterion is that each read transaction can view a different serial execution order of a subset of update transactions and itself. Through the concurrency control of update transactions, all the transactions see their own consistent "view", and the consistency of the database can also be maintained.

13.3 FORMALIZATION OF CONSISTENCY REQUIREMENTS

In the following sections, we come up with a correctness criterion that is appropriate for broadcast disk environments. We do this by first viewing correctness requirements from the perspective of the user in a traditional database system and exploring the correctness criterion implied by these requirements. We then weaken these requirements in the context of broadcast disks. This section expresses the requirements of users in a traditional database system and studies how view serializability matches with them.

For the rest of the chapter, we make the following basic assumptions about the database system.

1. The initial state of the database is consistent.

2. A transaction is the unit of consistency. Each transaction that modifies the database transforms the database from one consistent state to another. Further, the only way to get from one consistent state to another is by executing a transaction as though it were executed in isolation on the initial consistent state.

13.3.1 Requirements

We now informally specify the requirements from the perspective of a user and justify these requirements. Consider a history $\mathcal{H}$. $\mathcal{H}$ is *legal* iff all of the following hold:

1. The final state of the database after the occurrence of all the operations of committed transactions in $\mathcal{H}^{\text{J}}$ (the committed projection of $\mathcal{H}$), in the same order as in $\mathcal{H}$, is consistent.

2. The committed final state of the database is the same as the state produced by the transformations of all and only the committed transactions performed in some serial order on the initial state of the database. Further, all the transactions whose transformations "affect" the final state of the database in this order of execution should read[1] the same values of objects and perform the same transformations as they did in the history $\mathcal{H}$.

3. The values of the objects read (if any) by a committed transaction in $\mathcal{H}$ are consistent and are the same as the values of those objects in a database state produced by the transformations of some subset of committed transactions in $\mathcal{H}$.

4. Every prefix $\mathcal{H}'$ of $\mathcal{H}$ satisfies the above criteria.

Intuitively, for a history to be legal, requirement 1 says that the user wants to see the database in a consistent state after the completion (execution and commitment or abortion) of all transactions in the history. Requirement 2 states that all and only the effects of committed transactions, as reflected in the history, are visible in the database. Requirement 3 states that the values read by committed transactions in the history are consistent committed values. Requirement 4 states that the above properties hold for all the prefixes of the history because the state of the database should be consistent in the event of failures.

13.3.2 Formalization of Requirements

We now formalize the informally defined user requirements for a legal history $\mathcal{H}$ as follows. A history $\mathcal{H}$ is *legal* iff there exists a serial order S of all and only the committed transactions in $\mathcal{H}$ such that:

1. The final state of the database after the occurrence of all the operations of committed transactions in $\mathcal{H}^{\text{J}}$ (the committed projection of $\mathcal{H}$) in the same order as in $\mathcal{H}$, is the same as the state of the database on running each committed transaction in $\mathcal{H}$ in the serial order S.

   Justification. Since the final state of the database must be consistent (requirement 1), the final state of the database should be the state resulting from running some number (≥ 0) of transactions one after the other (assumption 1) in some serial order. Also, by requirement 2, we can infer that the final state of the database must be the same as the state resulting from running committed transactions in $\mathcal{H}$ in some serial order on the initial state of the database.

2. At least one of the following two statements hold:

   - Each transaction in S reads the same values of objects and performs the same transformations on the database as it does in the history $\mathcal{H}$.
   - There exists some suffix of transactions in S such that when these transactions are run in the same order as in S on any consistent state of the database, they read the same values of objects and perform the same transformations on the database as they did in the history $\mathcal{H}$. This suffix is the set of transactions which "affect" the final state of the database.

Justification. By requirement 2, each transaction which affects the final state of the database should read the same values of objects and perform the same transformation on the database as in the history $\mathcal{H}$. If all transactions affect the final state, then we have the first part of the above formal requirement. On the other hand, not all transactions may affect the final state of the database.

Example 3. Consider the following history :

$$r_1[a];\ r_2[a];\ w_1[a];\ w_2[a];\ w_3[a];\ c_3;\ c_2;\ c_1$$

Assume that the above history runs in a database with just one object, a. Since T_3 writes onto a without performing any read operations on objects in the database, the operations performed by the other transactions, T_1 and T_2, do not affect the final state of the database. Irrespective of their operations, the final state of the database is going to reflect only the final write operation by T_3. Thus, the final state of the database is the same as the one produced by running the transactions in the order $T_1T_2T_3$ on the initial state of the database. Note, however, that if there are multiple objects in the database, then, T_1 and T_2 may affect the final state of the database. If they were executed in a state that is different from the state in which they were executed in the history (for example, in the order $T_1T_2T_3$), they may write to an object other than a and hence affect the final state of the database. In this case, the final state of the database will not be the same as the state produced by running the transactions in some serial order.

Since not all transactions may affect the final state of the database, this means that there should exist a serial ordering of transactions S, which satisfies requirement 1, such that a suffix of S consists of exactly the set of transactions which affect the final state of the database ($\{T_3\}$ in the above example). Also, since no other transaction before this suffix affects the final state of the database, the transactions in this suffix when executed in the order specified by S on any consistent state (or equivalently, on any state got by some serial execution of transactions, by assumption 2) should produce the same final state of the database (T_3 would lead to the same state, whatever be the database state it is run on, as long as that state is consistent). This together with requirement 2 implies the second part of the above formal requirement.

3. Each committed transaction in $\mathcal{H}$ reads the same values of objects as in the state after some serial execution of some set of committed transactions in $\mathcal{H}$.

   Justification. Since each transaction reads consistent values of objects (requirement 3), the values of objects read should be the same as the values of objects after the execution of some number of transactions in some serial order (assumption 1). Further, since only the transformations of committed transactions in $\mathcal{H}$ should be seen (requirement 3), the formal requirement follows.

4. The above three requirements also hold for each prefix $\mathcal{H}'$ of $\mathcal{H}$.

   Justification. Follows from requirement 4.

13.3.3 Comparison with View Serializability

In this subsection, we explore the relationship between the correctness criterion presented in the previous subsection and view serializability. In order to formally capture the relationship, we first spell out the knowledge made available to schedulers. Most traditional transaction processing systems assume that schedulers have similar knowledge [Bernstein et al., 1987], but they have been listed here for completeness.

A scheduler which determines legal histories is assumed to know the following:

1. the history of read, write, commit and abort operations of transactions.

2. that a transaction is a deterministic program and its actions depend only on the values it reads from the database.

3. that each transaction terminates when run in isolation on a consistent state.

4. the identity of data objects in the database.

A scheduler does not have any information about the following:

1. the internals of a transaction except what is available in the knowledge of schedulers.

2. correspondence between transaction execution traces and transaction programs. In particular, a scheduler cannot determine whether two transaction execution traces in a history are due to the execution of the same program.

3. the values read from and written to the database.

4. number of values an object in the database can take.

We now characterize the set of histories which satisfy the formal requirements and which are acceptable by some scheduler. The proofs of the claims made are long and hence only the sketches of the proofs are presented.

Theorem 1 *If schedulers know the number of objects in the database and this number is finite, then the set of histories which satisfy the formal requirements and which can be accepted by some scheduler is a strict superset of the set of view serializable histories.*

Proof Sketch 13.1 *It is easy to see that every view serializable history also satisfies the formal requirements given in section 13.3.2. Thus the set of scheduler acceptable histories which satisfy the formal requirements is a superset of the set of view serializable histories. Further, Example 3 gives a history which satisfies the formal requirements but which is not view serializable. The theorem follows from the above statements.*

Theorem 2 *If there are infinite number of objects in the database or if the schedulers do not know the number of objects in the database, then the set of histories which satisfy the formal requirements and which can be accepted by some scheduler is exactly the set of view serializable histories.*

Proof Sketch 13.2 *It is easy to see that the set of scheduler acceptable histories which satisfy the formal requirements is a superset of the set of view serializable histories. Also, given the knowledge available to schedulers, the only way that a scheduler can determine that the final state is consistent after the ocurrence of a history $\mathcal{H}$ is if all committed transactions in $\mathcal{H}$ read the same values of objects as they do in some serial execution. If this were not the case, then the behavior of transactions is unknown and they could potentially write arbitrary values to data objects and thus make the final state inconsistent. Also, the only way that schedulers can determine that transactions in $\mathcal{H}$ read the same values of objects as they do in some serial history S is if each transaction reads values of objects from the same transaction in both histories. But this would mean that $\mathcal{H}$ is view serializable.*

13.4 WEAKENED REQUIREMENTS

In this section, we propose a weaker correctness criterion stemming from weakened user requirements. We first present the motivation for weakening user requirements and then state the requirements informally before expressing them formally. The work presented here differs from that in [Garcia-Molina and Wiederhold, 1982] in two respects. First, we do not assume that the scheduler knows whether transactions are read-only in all possible executions. Also, in our correctness criterion, it is sufficient that read-only transactions are serialized with respect to a *subset* of update transactions without being serialized with respect to *all* the update transactions.

13.4.1 Motivation for Weaker Requirements

In earlier sections it was shown that ensuring serializability in broadcast disk environments may lead to expensive communication or unnecessary aborts. Communication is expensive because of the limited bandwidth available from clients to the server and unnecessary aborts take place because the scheduler has to be conservative in the absence of global information. In this light, it would be worthwhile to come up with a correctness criterion that is weaker than serializability and would avoid the above costs. The requirements stated in the previous section could be weakened to achieve this goal. We illustrate this with an example of a history that is accepted by the weakened requirements but not by the original requirements.

Consider a variation of the history in Example 1:

$$r_1[x];\ w_2[x];\ c_2;\ r_3[x];\ r_3[y];\ w_4[y];\ c_4;\ r_1[y];\ c_1;\ c_3$$

Only T_2 and T_4 perform write operations in this history. The transaction T_1 sees these two transactions in the order T_4T_2 (since it reads the value of an object, y, after T_4 writes to it reads the value of another object, x, before T_2 writes to it). Similarly T_3 sees these two transactions in the order T_2T_4. Thus in any serial order of the committed transactions T_1, T_2, T_3, T_4, at least one of T_1 and T_3 would be executed in a state of the database that is different from the state in which it was executed in the history. Thus the scheduler would not know what operations this transaction would perform. This transaction may also write to an object (even though this transaction is read-only in the history, it could perform an update in another history if it reads different values of objects) that is untouched by other transactions, thus leading to a final state of the database that is different from the final state produced by the above history. Thus the requirements would not be satisfied by this history. However is this history still "correct"?

In the above history, the final state of the database is the one produced by running the two *update* transactions (the transactions that perform write operations in the history), T_2 and T_4. Hence, the final state of the database is consistent. Further, each transaction reads consistent values of objects because:

- T_1 reads values of objects that are the same as those in a state produced by running T_4 on the initial state of the database

- T_3 reads values of objects that are the same as those in a state produced by running T_2 on the initial state of the database
- T_2 and T_4 do not perform any read operations and hence, vacuously read consistent values of objects

For schedulers to accept such histories, weaker requirements need to be specified.

13.4.2 Weakened Requirements

The weakened requirements differ from the requirements outlined in section 13.3.1 in only the second part. Instead of requiring the final state of the database to contain the effect of *all* transactions as in the original requirements, only the effects of *update* transactions in the history are required to be present in the final database state. More precisely, the second part of the requirements is modified as follows in the weakened correctness requirements:

- The committed final state of the database is the same as the state produced by the transformations of all and only committed *update* transactions performed in some order on the initial state of the database. Further, all the transactions whose transformations "affect" the final state of the database in this order of execution should read the same values of objects and perform the same transformations as they did in the history $\mathcal{H}$.

Transactions that perform only read operations in a history may perform updates when run in a database state that is different from the one in which it is run in the history. An implication of the weakening of requirements is that these transactions may see serialization orders that are different from the serial order seen by the update transactions. Thus, update operations that may be done by these transactions, when run in the global serial order of update transactions, are ignored. However, this may still be acceptable to users since the transactions read consistent values of objects and the final state of the database is consistent.

13.4.3 Formalizing Weakened Requirements

A history $\mathcal{H}$ satisfies the weakened requirements iff there exists a serial order S of all and only the committed transactions in $\mathcal{H}$ such that:

1. The final state of the database after the occurrence of all the operations in committed transactions in $\mathcal{H}^{\rfloor}$ (the committed projection of $\mathcal{H}$), in the same order as in $\mathcal{H}$ is the same as the state of the database on running each committed *update* transaction in $\mathcal{H}$ in the order S.
2. At least one of the following two statements hold:

- Each transaction in S reads the same values of objects and performs the same transformation on the database as it does in the history $\mathcal{H}$.
- There exists some suffix of transactions in S such that when these transactions are run in the same order as in S on any consistent state of the database, they read the same values of objects and perform the same transformation on the database as they did in the history $\mathcal{H}$. This suffix is the set of transactions which "affect" the final state of the database.

3. Each committed transaction in $\mathcal{H}$ reads the same values of objects as in the state after some serial execution of some set of committed transactions in $\mathcal{H}$.

4. The above two requirements also hold for each prefix $\mathcal{H}'$ of $\mathcal{H}$.

The final state of the database should be the same as that obtained by running all the committed update transactions in some serial order. Thus all update transactions are serialized with respect to each other. Transactions that do not make updates may see serialization orders that are different from the order seen by update transactions. Each of these transactions is serialized *after* a *subset* of the update transactions, and hence, reads consistent values. This subset would include all transactions that it "directly or indirectly reads from". The lack of a global serialization order involving transactions which perform only read operations in the history can be effectively used in broadcast disk environments, as is shown in the next section.

13.5 MECHANISMS TO GUARANTEE CORRECTNESS

In this section, we propose mechanisms and protocols that are required to ensure correctness according to the criteria described in the previous sections. Specifically, we consider the weakened correctness criteria described in Section 13.4. The mechanisms and protocols to enforce this criterion are influenced mainly by the assumption that contacting the server in a broadcast disk environment is not cost-effective as only limited bandwidth is available for client-to-server communication. A key feature of our protocol is that it eliminates the need for client-to-server communication for read-only transactions, while ensuring correctness according to requirements described in Section 13.4.

The rest of this section is organized as follows. We begin by describing the concept of a broadcast disk from an implementation point of view. We then describe the protocol followed by clients and the server to ensure correctness even when read transactions do not contact the server at all. The protocol involves steps to be taken by a client to read or write database items and extra processing required at client and server while committing a transaction. We then prove that adherence to this protocol results in transaction histories that are acceptable according to the requirements described in Section 13.4.

13.5.1 Broadcast Disks

Broadcast disk is a form of data-dissemination based system appropriate for wireless communication environments. The sever periodically broadcasts values of all the data items in the database. The clients view the broadcast medium as a disk and can read the value of various data items by listening to the broadcast. For writing an item in the database, the clients contact the server with appropriate information. The period of broadcast of different data items can be different (i.e. "hot" data items may be broadcast more frequently). This is analogous to having several disks with different rotation speeds. In this paper, however, we only consider a single disk – i.e., all the items are broadcast with the same period. At the beginning of each broadcast cycle the server may broadcast several items of control information such as indices. Indices help clients save power by determining the exact times when the receiving units need to be switched on. The format and techniques of actual broadcasts have been described in detail in [Acharya et al., 1996]. Our protocol requires additional control information to be broadcast in order to ensure correctness.

13.5.2 Protocol

To ensure correctness when read-only transactions do not contact the server, both clients and server have to perform certain additional functions. We first describe the functionality provided by the server.

13.5.2.1 Server Functionality. The server is responsible for three main tasks:

- Broadcast the values of all data items in the database in a periodic manner. We assume that all data items are broadcast with the same period. We term this period as a broadcast cycle.
- Provide locking and validation services to update transactions. The write operations in a broadcast disk database are mostly similar to write operations in conventional client-server databases. The concurrency control approach can either be hybrid or purely optimistic. Under the hybrid approach, the server grants write locks on each item a client wishes to write. Write locks granted by our broadcast database server only prevent other transaction from writing that data item. It does not prevent the server from broadcasting the previously committed value of that data item (thus allowing read operations on that data item to succeed). Under the optimistic approach, the server validates each transaction at commit time to ensure serializability. Thus, in both cases the server ensures that write-write conflicts are avoided.
- The server broadcasts two items of control information at the beginning of each broadcast cycle. One is an *update list* which is a list of all updates

made to the database. The list consists of entries that are three-tuples of the form $< T, ob, c >$, indicating that transaction T, updated the value of data object *ob* and committed between the beginning of broadcast cycles $c-1$ and c. The other is the *serialization graph*, which is a dependency graph of all the update transaction validated at the server. Note that to construct a dependency graph, the update transactions have to inform the server of not only the items they have written, but also about the items they had read and the cycles in which they read the items.

In short, the server is responsible for the actual broadcast, maintaining serializability of the update transactions and providing sufficient information to read-only transactions such that they can make decisions about the consistency of the data items they have read.

13.5.2.2 Client Functionality. The clients can execute two types of transactions – read-only and update. We describe the protocol followed by clients to read and write items in the database, and the extra steps required during the commit of an update transaction.

- *Read Operation:* When a transaction needs to read a data item, it first reads the update list and the serialization graph broadcast at the beginning of the broadcast cycle. Using the update list, the transaction can add its own dependency edges in the serialization graph, for the read/write operations it has carried out so far and the read operation it intends to perform next. If the resulting serialization graph is acyclic, the transaction reads the value of the data item during the ongoing broadcast cycle. Otherwise, the transaction aborts.

- *Write Operation:* When a transaction wishes to write a data item in the database, it can take two different approaches, depending on whether the optimistic or hybrid concurrency control mechanism is in use.

  - *Optimistic Concurrency Control:* Under the optimistic approach, the update transaction first makes a local copy of the data item and updates that copy. This copy is sent to the server at commit time.
  - *Hybrid Concurrency Control:* If hybrid concurrency control is in use, then the client sends a request for write lock to the server. The server grants the request if the item is not already locked. The server continues to broadcast the previously committed value of the data item. The transaction then sends the new value of the data item to the server. This value appears on the broadcast after the transaction has committed.

- *Commit Operation:* The commit time processing is different for read-only and update transactions.

- *Read-Only Transactions:* If a transaction does not write any data items during its execution, its commit operation is trivial: do nothing. This is because each read operation is validated before it is performed, and if the transaction has not aborted during its course, there is little it needs to do at commit time. There is no need to contact the database server. Notice that this protocol has an interesting implication: the server does not need know anything about the read-only transactions. This augers well for scalability.

- *Update Transactions:* If a transaction is an update transaction (i.e. it has written onto some data item in the database), it must convey all the information necessary for the server to update the entries in the update list and the serialization graph. Consequently, depending upon whether optimistic or hybrid concurrency control scheme is in use, following actions are taken at commit time:

    * *Hybrid Concurrency Control:* As the hybrid approach requires that the client acquire locks on every data item it wishes to write, the server is always aware of all the data items updated by the transaction that is about to commit. Hence, at commit time, the client only needs to send to the database a list of all the data items it has read and the broadcast cycle number in which each item was read. The server checks its serialization graph to ensure that the dependencies induced by the read operations, do not result in a cycle in the graph. If a cycle is detected, then the transaction is aborted. Otherwise, the serialization graph and update list are updated to reflect the operations of this transaction.

    * *Optimistic Concurrency Control:* Under optimistic concurrency control, the client is not required to obtain locks on data items they wish to write. Instead, all the updates are validated at the server at commit time. At commit time, the transaction sends to server a record of all the data items it has updated and their values. In addition, it also sends a list of all the data items it has read and the broadcast cycle number in which each item was read. The server checks to see if these reads and writes introduce any cycles in the serialization graphs. If they do not, the transaction is allowed to commit and the serialization graph and update list are updated to reflect the operations of this transaction. Otherwise, the transaction is aborted.

13.5.3 Proof of Correctness

In this section we informally prove that if the protocol described in the previous section is followed, it guarantees correctness according to the correctness criteria described in Section 13.4.3.

- *Proof of weakened formal requirement 1.* This follows from the fact that the server ensures the serializability of all update transactions.
- *Proof of weakened formal requirement 2.* The first statement of this requirement always holds since the server ensures that all update transactions are serialized in the traditional sense.
- *Proof of weakened formal requirement 3.* The server broadcasts updated values of data items if and only if the updating transaction commits successfully. Thus, in any broadcast cycle, the values of the data items reflect only the changes introduced by transactions that committed before the beginning of this broadcast cycle. Before each read operation, the client verifies it against the current update list and the serialization graph to ensure that the read operation does not introduce any cycles in the serialization graph. Hence it follows that each transaction reads values of objects that are the same as the values in a state produced by running some of the committed transactions that committed before the beginning of this broadcast cycle in some serial order.
- *Proof of weakened formal requirement 4.* This follows by noting that the protocol is followed by each transaction.

13.6 CONCLUSIONS AND FUTURE WORK

In this chapter we have presented a new correctness criterion appropriate for environments like broadcast disks where client-to-server communication bandwidth is limited and asymmetric. We have also presented mechanisms and protocols by which this correctness criterion can be enforced in broadcast disk environment. We plan to extend this work in the following directions:

- Study the implications of weakening scheduler restrictions on the set of acceptable histories.
- Investigate extensions to the mechanisms which better exploit the potential of the correctness criterion.
- Explore ways to minimize the control information exchanged between the clients and the server.
- Evaluate and compare the performance of various mechanisms through simulation.

- Examine the possibility of using multi-version concurrency control methods.

Notes

1. This should not be confused with the reads-from relation between transactions in a history.

References

[Abdelzaher et al., 1996] Abdelzaher, T., Shaikh, A., Jahanian, F., and Shin, K. (1996). RTCAST: Lightweight Multicast for Real-Time Process Groups. In *IEEE Real-Time Technology and Applications Symposium*, Boston, MA, USA.

[Acharya et al., 1996] Acharya, S., Franklin, M., and Zdonik, S. (1996). Disseminating Updates on Broadcast Disks. In *Proceedings of the International Conference on Very Large Data Bases.*

[ActionTechnologies, 1993] ActionTechnologies (1993). What is ActionWorkflow?- A primer. ActionTechnologies, Inc., Alameda, California.

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

[Alonso et al., 1996a] Alonso, G., Agrawal, D., and El Abbadi, A. (1996a). Process Synchronization in Workflow Management Systems. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing, New Orleans, LA.*

[Alonso et al., 1996b] Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Günthör, R., and Mohan, C. (1996b). Advanced transaction models in workflow contexts. In *Proceedings of the 12th International Conference on Data Engineering, Feb. 26 - March 1*, New Orleans, Louisiana, USA.

[Alonso et al., 1995a] Alonso, G., Agrawal, D., El Abbadi, A., Mohan, C., Günthör, R., and Kamath, M. (1995a). Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proceedings of the IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organizations*, Trondheim, Norway.

[Alonso and El Abbadi, 1994] Alonso, G. and El Abbadi, A. (1994). Cooperative Modeling in Applied Geographic Research. *International Journal of Intelligent and Cooperative Information Systems*, 3(1):83–102.

[Alonso et al., 1996c] Alonso, G., Günthör, R., Kamath, M., Agrawal, D., El Abbadi, A., and Mohan, C. (1996c). Exotica/FMDC: A workflow management system for mobile and disconnected clients. *International Journal of Distributed and Parallel Databases*, 4(3):229–247.

[Alonso et al., 1995b] Alonso, G., Kamath, M., Agrawal, D., El Abbadi, A., Günthör, R., and Mohan, C. (1995b). Failure handling in large scale workflow management systems. Technical Report RJ 9913, IBM Almaden Research Center.

[Alonso and Schek, 1996a] Alonso, G. and Schek, H.-J. (1996a). Database technology in workflow environments. *INFORMATIK-INFORMATIQUE (Journal of the Swiss Computer Science Society).*

[Alonso and Schek, 1996b] Alonso, G. and Schek, H. J. (1996b). Research Issues in Large Workflow Management Systems. In [Sheth, 1996]. Available from `http://LSDIS.cs.uga.edu/activities/NSF-workflow`.

[Alonso et al., 1994] Alonso, G., Vingralek, R., Agrawal, D., Breitbart, Y., Abbadi, A. E., Schek, H., and Weikum, G. (1994). A Unified Approach to Concurrency Conrol and Transaction Recovery. In *Proceedings of the 4th International Conference on Extending Database Technology.*

[Ammann et al., 1996] Ammann, P., Jajodia, S., and Ray, I. (1996). Ensuring atomicity of multilevel transactions. In *Proceedings 1996 IEEE Computer Society Symposium on Security and Privacy*, pages 74–84, Oakland, CA.

[Ammann et al., 1997] Ammann, P., Jajodia, S., and Ray, I. (1997). Applying formal methods to semantic-based decomposition of transactions. To appear in ACM Transactions on Database Systems.

[Ansari et al., 1992] Ansari, M., Ness, L., Rusinkiewicz, M., and Sheth, A. (1992). Using Flexible Transactions to Support Multi-system Telecommunication Applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 65–76, Vancouver, Canada.

[Anwar, 1996] Anwar, E. (1996). *A New Perspective on Rule Support for Object-Oriented Databases*. PhD thesis, CISE Department, University of Florida, Gainesville, FL.

[Anwar et al., 1993] Anwar, E., Maugis, L., and Chakravarthy, S. (1993). A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings of the International Conference on Management of Data, Washington D.C.*

[Arnold and Gosling, 1996] Arnold, K. and Gosling, J. (1996). *The Java^{TM} Programming Language*. The Java^{TM} Series. Addison-Wesley.

[Atkinson et al., 1996] Atkinson, M. P., Daynès, L., Jordan, M. J., Printezis, T., and Spence, S. (1996). An Orthogonally Persistent Java^{TM}. *SIGMOD RECORD*, 25(4).

[Atkinson and Jordan, 1996] Atkinson, M. P. and Jordan, M., editors (1996). *First International Workshop on Persistence and Java*, Drymen, Scotland. Sunlabs Technical Report.

[Atkinson and Morrison, 1995] Atkinson, M. P. and Morrison, R. (1995). Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3).

[Atluri et al., 1994] Atluri, V., Bertino, E., and Jajodia, S. (1994). Degrees of isolation, concurrency control protocols and commit protocols. In Biskup, J. et al., editors, *Database Security, VIII: Status and Prospects*, pages 259–274. North-Holland, Amsterdam.

[Attie et al., 1992] Attie, P., Singh, M., Rusinkiewicz, M., and Sheth, A. (1992). Specifying and Enforcing Intertask Dependencies. Technical Report MCC Report: Carnot-245-92, Microelectronics and Computer Technology Corporation.

[Attie et al., 1993] Attie, P., Singh, M., Sheth, A., and Rusinkiewicz, M. (1993). Specifying and Enforcing Intertask Dependencies. In *Proceedings of the International Conference on Very Large Data Bases*, pages 134–145, Dublin, Ireland.

[Audsley et al., 1991] Audsley, N., Burns, A., Richardson, M., and Wellings, A. (1991). A Database Model for Hard Real-Time Systems. Technical report, University of York, Real-time Systems Group.

[Badrinath and Ramamritham, 1991] Badrinath, B. R. and Ramamritham, K. (1991). Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 16.

[Barbara et al., 1996a] Barbara, D., Mehrotra, S., and Rusinkiewicz, M. (1996a). INCAs: Managing Dynamic Workflows in Distributed Environments. *Journal of Database Management, Special Issue on Multidatabases*, 7(1):5–15.

[Barbara et al., 1996b] Barbara, D., Mehrotra, S., and Vallabhaneni, T. (1996b). The Gold Text Indexing Engine. In *Proceedings of the IEEE International Conference on Data Engineering, New Orleans.*

[Barga and Pu, 1995] Barga, R. and Pu, C. (1995). A Practical and Modular Method to Implement Extended Transaction Models. In *Proceedings of the International Conference on Very Large Data Bases*, pages 206–217, Zurich, Switzerland.

[Barga, 1997] Barga, R. S. (1997). *A Reflective Framework for Implementing Extended Transactions.* PhD thesis, Oregon Graduate Institute of Science & Technology.

[Barga and Pu, 1996] Barga, R. S. and Pu, C. (1996). Reflecting on a Legacy Transaction Processing Monitor. In *Proceedings of the ACM Reflections '96 Conference*, Palo Alto, CA.

[Barga et al., 1994] Barga, R. S., Pu, C., and Hseush, W. W. (1994). A Practical Method for Realizing Semantics-based Concurrency Control. Technical Report OGI-CSE-94-032, Oregon Graduate Institute of Science & Technology, Department of Computer Science and Engineering.

[Barghouti and Kaiser, 1991] Barghouti, N. S. and Kaiser, G. E. (1991). Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317.

[Ben-Shaul and Kaiser, 1995] Ben-Shaul, I. and Kaiser, G. E. (1995). *A Paradigm for Decentralized Process Modeling*. Kluwer Academic Publishers, Boston, USA.

[Berenson et al., 1995] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose*.

[Bernstein and Goodman, 1984] Bernstein, P. and Goodman, N. (1984). An algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Transactions on Database Systems*, 9(4):596–615.

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

[Bernstein, 1996] Bernstein, P. A. (1996). Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98.

[Bernstein et al., 1987] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA.

[Bertino et al., 1997] Bertino, E., Jajodia, S., Mancini, L., and Ray, I. (1997). Advanced Transaction Processing in Multilevel Secure File Stores. *IEEE Transactions on Knowledge and Data Engineering*. To appear.

[Bhargava, 1987] Bhargava, B. K., editor (1987). *Concurrency Control and Reliability in Distributed Systems*. Van Nostrand Reinhold Company, New York.

[Biliris et al., 1994] Biliris, A., Dar, S., Gehani, N., Jagadish, H., and Ramamritham, K. (1994). ASSET: A System for Supporting Extended Transactions. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 44–54, Minneapolis, MN.

[Birman, 1993] Birman, K. (1993). The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12).

[Birman, 1994] Birman, K. (1994). A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *ACM Operating System Review*, 28(1).

[Birman and Joseph, 1987a] Birman, K. and Joseph, T. (1987a). Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Austin, TX, USA.

[Birman and Joseph, 1987b] Birman, K. and Joseph, T. (1987b). Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1).

[Birman and Renesse, 1994] Birman, K. P. and Renesse, R. V. (1994). *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press.

[Bodorik and Riordon, 1988] Bodorik, P. and Riordon, J. (1988). Distributed Query Processing Optimization Objectives. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 320–329.

[Bonner et al., 1996] Bonner, A., Shruf, A., and Rozen, S. (1996). LabFlow-1: A Database Benchmark for High Throughput Workflow Management. In *Proceedings of the 5th. Intnl. Conference on Extending Database Technology*, pages 25–29, Avignon, France.

[Bracchi and Pernici, 1985] Bracchi, G. and Pernici, B. (1985). The Design Requirements of Office Systems. *ACM Transactions on Office Information Systems*, 2(2):151–170.

[Breitbart et al., 1993] Breitbart, Y., Deacon, A., Schek, H., Sheth, A., and Weikum, G. (1993). Merging Application-centric and Data-centric Approaches to Support Transaction- oriented Multi-system Workflows. *SIGMOD Record*, 22(3):23–30.

[Brown and Carey, 1992] Brown, K. P. and Carey, M. J. (1992). On Mixing Queries and Transactions via Multiversion Locking. In *Proceedings of the IEEE International Conference on Data Engineering, Phoenix.*

[Brown et al., 1994] Brown, K. P., Mehta, M., Carey, M., and Livny, M. (1994). Towards Automated Performance Tuning for Complex Workloads. In *Proceedings of the International Conference on Very Large Data Bases.*

[Bukhres et al., 1993] Bukhres, O., Elmagarmid, A., and Kuhn, E. (1993). Implementation of the Flex Transaction Model. *Bulletin of the IEEE Technical Committee on Data Engineering*, 12(2):28–32.

[Bukhres and Elmagarmid, 1996] Bukhres, O. A. and Elmagarmid, A. K., editors (1996). *Object-Oriented Multidatabase Systems.* Prentice Hall, Englewood Cliffs, New Jersey.

[Cabrera et al., 1993] Cabrera, L.-F., McPherson, J. A., Schwarz, P. M., and Wyllie, J. C. (1993). Implementing Atomicity in Two Systems: Techniques, Tradeoffs and Experience. *IEEE Trans. on Software Engineering*, 19(10):950 –961.

[Carey et al., 1989] Carey, M., Jauhari, R., and Livny, M. (1989). Priority in DBMS Resource Scheduling. In *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, Netherlands.

[Ceri and Pelagatti, 1984] Ceri, S. and Pelagatti, G. (1984). *Distributed Databses: Principles and Systems*. McGraw Hill.

[Chakravarthy and Anwar, 1995] Chakravarthy, S. and Anwar, E. (1995). Exploiting Active Database Paradigm for Supporting Flexible Transaction Models. Technical Report UF-CIS TR-95-026, CISE Department, University of Florida, E470-CSE, Gainesville, FL.

[Chakravarthy et al., 1994] Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S. K. (1994). Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings of the International Conference on Very Large Data Bases*.

[Chakravarthy et al., 1995] Chakravarthy, S., Krishnaprasad, V., Tamizuddin, Z., and Badani, R. (1995). ECA Rule Integration into an OODBMS: Architecture and Implementation. In *Proceedings of the IEEE International Conference on Data Engineering*.

[Chan et al., 1982] Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D. (1982). The Implementation of an Integrated Concurrency Control and Recovery Scheme. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[Chekuri et al., 1995] Chekuri, C., Hasan, W., and Motwani, R. (1995). Scheduling Problems in Parallel Query Optimization. In *Proceedings of the ACM Symposium on Principles of Database Systems, San Jose*.

[Chen et al., 1993] Chen, J., Bukhres, O. A., and Elmagarmid, A. K. (1993). IPL: A Multidatabase Transaction Specification Language. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA.

[Chen and Dayal, 1996] Chen, Q. and Dayal, U. (1996). A Transactional Nested Process Management System. In *Proceedings of 12th. IEEE International Conference on Data Engineering*, pages 566–573, New Orleans, LA.

[Cheriton and Skeen, 1993] Cheriton, D. and Skeen, D. (1993). Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, USA.

[Cheriton and Zwaenepoel, 1985] Cheriton, D. and Zwaenepoel, W. (1985). Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2).

[Chrysanthis and Ramamritham, 1991] Chrysanthis, P. and Ramamritham, K. (1991). A formalism for extended transaction models. In *Proceedings of the International Conference on Very Large Data Bases.*

[Chrysanthis and Ramamritham, 1990] Chrysanthis, P. K. and Ramamritham, K. (1990). ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203.

[Chrysanthis and Ramamritham, 1992] Chrysanthis, P. K. and Ramamritham, K. (1992). ACTA: The SAGA continues. In Elmagarmid, A., editor, *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers.

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

[Clauer et al., 1993] Clauer, C. et al. (1993). UARC: A Prototype Upper Atmospheric Research Collaboratory. *EOS Transactions, American Geophysical Union.*

[Coalition, 1994] Coalition, T. W. M. (1994). Glossary – A Workflow Management Coalition Specification. Technical report, The Workflow Management Coalition, Brussels, Belgium. URL: `http://www.aiai.ed.ac.uk/WfMC/`.

[Cristian, 1991] Cristian, F. (1991). Understanding fault tolerant distributed systems. *Communications of the ACM*, 34(2):57–78.

[Curtis et al., 1992] Curtis, B., Kellner, M. I., and Over, J. (1992). Process Modelling. *Communications of the ACM*, 35(9).

[Das, 1997] Das, S. (1997). ORBWORK: The CORBA-based Distributed Engine for the METEOR$_2$ Workflow Management System. Master's thesis, University of Georgia, Athens, GA. In preparation. URL: `http://LSDIS.cs.uga.edu/`.

[Dayal et al., 1990] Dayal, U., Hsu, M., and Ladin, R. (1990). Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, Atlantic City.

[Dayal et al., 1991] Dayal, U., Hsu, M., and Ladin, R. (1991). A Transactional Model for Long-running Activities. In *Proceedings of the International Conference on Very Large Data Bases*, pages 113–122, Barcelona, Spain.

[Daynès, 1995] Daynès, L. (1995). *Conception et réalisation de mécanismes flexibles de verrouillage adaptés aux SGBDO client-serveur.* PhD thesis, Université Pierre et Marie Curie (Paris VI – Jussieu).

[Daynès et al., 1995] Daynès, L., Gruber, O., and Valduriez, P. (1995). Locking in OODBMS clients supporting Nested Transactions. In *Proceedings of the*

11th International Conference on Data Engineering, pages 316–323, Taipei, Taiwan.

[DeWitt and Gray, 1992] DeWitt, D. J. and Gray, J. N. (1992). Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98.

[dos Santos and Theroude, 1996] dos Santos, C. S. and Theroude, E. (1996). Persistent Java. In [Atkinson and Jordan, 1996]. Sunlabs Technical Report.

[Duke and Duke, 1990] Duke, D. and Duke, R. (1990). Towards a semantics for Object Z. In Bjorner, D., Hoare, C. A. R., and Langmaack, H., editors, *VDM'90: VDM and Z*, volume 428 of *Lecture Notes in Computer Science*, pages 242–262. Springer-Verlag.

[Eder and Liebhart, 1995] Eder, J. and Liebhart, W. (1995). The Workflow Activity Model WAMO. In *Proceedings of the 3rd. Int. Conference on Cooperative Information Systems*, Vienna, Austria.

[Eder and Liebhart, 1996] Eder, J. and Liebhart, W. (1996). Workflow Recovery. In *Proceedings of the 1st. IFCIS Conference on Cooperative Information Systems*, Brussels, Belgium.

[Elhardt and Bayer, 1984] Elhardt, K. and Bayer, R. (1984). A Database Cache for High Performance and Fast Restart in Database Systems. *ACM Transactions on Database Systems*, 9(4):503–525.

[Ellis and Gibbs, 1989] Ellis, C. and Gibbs, S. (1989). Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, OR, USA.

[Ellis et al., 1991] Ellis, C. A., Gibbs, S. J., and Rein, G. L. (1991). Groupware, some issues and experiences. *Communications of the ACM*, 34(1):39–58.

[Elmagarmid, 1992] Elmagarmid, A. K., editor (1992). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, Inc., San Mateo, CA.

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

[Emmrich, 1996] Emmrich, M. (1996). Object framework for business applications. In *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT96), March 25-29*, Avignon, France.

[Encina, 1993] Encina (1993). *Encina Toolkit Server Core Programmer's Reference*. Transarc Corporation, Pittsburgh, PA. 15219.

[Eppinger et al., 1991] Eppinger, J. L., Mummert, L. B., and Spector, A. Z. (1991). *Camelot and Avalon : A Distributed Transaction Facility*. Morgan Kaufmann, San Mateo, CA.

[Farrag and Özsu, 1989] Farrag, A. A. and Özsu, M. T. (1989). Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525.

[Feldmann et al., 1993] Feldmann, A., Kao, M. Y., Sgall, J., and Teng, S. H. (1993). Optimal Online Scheduling of Parallel Jobs with Dependencies. In *Proceedings of the ACM Symposium on Theory of Computing.*

[Fernandez and Zdonik, 1989] Fernandez, M. F. and Zdonik, S. (1989). Transaction Groups: A Model for Controlling Cooperative Transactions. In *Persistent Object Stores (Proceedings of the Third Int. Workshop on Persistent Object Systems)*, Workshops in Computing, pages 341–350, Newcastle, New South Wales, Australia. Springer-Verlag in collaboration with the British Computer Society.

[Fischer, 1995] Fischer, L. (1995). *The Workflow Paradigm - The Impact of Information Technology on Business Process Reengineering, 2nd. Edition.* Future Strategies, Inc., Alameda, CA.

[Frye, 1994] Frye, C. (1994). Move to Workflow Provokes Business Process Scrutiny. *Software Magazine*, pages 77–89.

[Garcia-Molina, 1983] Garcia-Molina, H. (1983). Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213.

[Garcia-Molina et al., 1991] Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. (1991). Modeling Long-Running Activities as Nested Sagas. *Bulletin of the Technical Committe on Data Engineering, IEEE*, 14(1): 18–22.

[Garcia-Molina and Salem, 1987] Garcia-Molina, H. and Salem, K. (1987). SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259.

[Garcia-Molina and Salem, 1992] Garcia-Molina, H. and Salem, K. (1992). Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516.

[Garcia-Molina and Spauster, 1991] Garcia-Molina, H. and Spauster, A. (1991). Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems*, 9(3).

[Garcia-Molina and Wiederhold, 1982] Garcia-Molina, H. and Wiederhold, G. (1982). Read-Only Transactions in a Distributed Database. *ACM Transactions on Database Systems*, 7(1):209–234.

[Garofalakis and Ioannidis, 1996] Garofalakis, M. N. and Ioannidis, Y. E. (1996). Multi-dimensional Resource Scheduling for Parallel Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Montreal.*

[Garthwaite and Nettles, 1996] Garthwaite, A. and Nettles, S. (1996). Transaction for Java. In [Atkinson and Jordan, 1996]. Sunlabs Technical Report.

[Gawlick, 1994] Gawlick, D. (1994). High Performance TP-Monitors – Do We Still Need to Develop Them? *Bulletin of the Technical Committee on Data Engineering, IEEE*, 17(1):16–21.

[Georgakopoulos, 1994] Georgakopoulos, D. (1994). Workflow Management Concepts, Commercial Products, and Infrastructure for Supporting Reliable Workflow Application Processing. Technical Report TR-0284-12-94-165, GTE Laboratories Inc., Waltham, MA.

[Georgakopoulos et al., 1994] Georgakopoulos, D., Hornick, M., Krychniak, P., and Manola, F. (1994). Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proceedings of the 10th. IEEE International. Conference on Data Engineering*, pages 462–473, Houston, TX.

[Georgakopoulos et al., 1996] Georgakopoulos, D., Hornick, M., and Manola, F. (1996). Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation. *IEEE Transactions on Knowledge and Data Engineering. April.*

[Georgakopoulos et al., 1995] Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–154.

[Georgakopoulos and Hornick, 1994] Georgakopoulos, D. and Hornick, M. F. (1994). A Framework for Enforceable Specification of Extended Transaction Models and Transactional Workflows. *Intnl. Journal of Intelligent and Cooperative Information Systems*, 3(3):599–617.

[Gifford, 1979] Gifford, D. (1979). Weighted Voting for Replicated Data. *Proceedings 7th Symposium on Operating System Principles*, pages 150–162.

[Gottemukkala and Lehman, 1992] Gottemukkala, V. and Lehman, T. J. (1992). Locking and Latching in a Memory-Resident Database System. In *Proceedings of the International Conference on Very Large Data Bases*, pages 533–544.

[Graefe, 1994] Graefe, G. (1994). Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6:120–135.

[Graham, 1992] Graham, M. (1992). Issues in Real-Time Data Management. *Real-Time Systems Journal*, 4(3). Special Issue on Real-Time Databases.

[Graham et al., 1979] Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. H. G. R. (1979). Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326.

[Gray, 1981] Gray, J. (1981). The Transaction Concept: Virtues and Limitations. In *Proceedings of the International Conference on Very Large Data Bases*, Cannes, France.

[Gray, 1993] Gray, J. (1993). *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann.

[Gray and Reuter, 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA.

[Guerraoui and Schiper, 1994] Guerraoui, R. and Schiper, A. (1994). The Transaction Model vs The Virtual Synchrony Model: Bridging the gap. In *Theory and Practice in Distributed Systems, Lecture Notes in Computer Science*, volume 938, New York, NY, USA. Springer Verlag.

[Hachem et al., 1993] Hachem, N. I., Qiu, K., Gennert, M., and Ward, M. (1993). Managing Derived Data in the Gaea Scientific DBMS. In *Proceedings of the International Conference on Very Large Data Bases*, Dublin, Ireland.

[Hagen, 1996] Hagen, C. (1996). Kombination von aktiven Mechanismen und Transaktionen im TRAMs-Projekt. In *8th Workshop "Grundlagen von Datenbanken", Friedrichsbrunn, Deutschland.*

[Hammer and Champy, 1993] Hammer, M. and Champy, J. (1993). *Reengineering the Corporation: A Manifesto for Business Revolution*. HarperBusiness, New York.

[Härder and Rothermel, 1993] Härder, T. and Rothermel, K. (1993). Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1):39–74.

[Haritsa et al., 1990] Haritsa, J., Carey, M., and Livny, M. (1990). Dynamic Real-time Optimistic Concurrency Control. In *Proceedings of the Real-Time Systems Symposium*, Lake Buena Vista, FL, USA.

[Hasse, 1995] Hasse, C. (1995). *Inter- and Intra-transaction Parallelism in Database Systems*. PhD thesis, Department of Computer Science, ETH Zurich. (In German).

[Hasse, 1996] Hasse, H. (1996). *A Unified Theory for the Correctness of Parallel and Failure-Resilient Executions of Database Transactions*. PhD thesis, Department of Computer Science, ETH Zurich. (In German).

[Heineman and Kaiser, 1997] Heineman, G. T. and Kaiser, G. E. (1997). The CORD Approach to Extensible Concurrency Control. In *Proceedings of the IEEE International Conference on Data Engineering*.

[Helal et al., 1996a] Helal, A., Heddaya, A., and Bhargava, B. (1996a). *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers.

[Helal et al., 1996b] Helal, A., Kim, Y., Elmagarmid, A., and Heddaya, A. (1996b). Transaction Optimization. In Bertino, E. et al., editors, *Proceedings of the Workshop on Advanced Transaction Models and Architectures, Goa, India.*

[Herlihy, 1987] Herlihy, M. (1987). Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, 36(4): 443–448.

[Herlihy and Weihl, 1991] Herlihy, M. P. and Weihl, W. E. (1991). Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61.

[Herman et al., 1987] Herman, G., Gopal, G., Lee, K. C., and Weinrib, A. (1987). The Datacycle Architecture for Very High Throughput Database Systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data, New York.*

[Hollinsworth, 1996] Hollinsworth, D. (1996). The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition. Accessible via: `http://www.aiai.ed.ac.uk/WfMC/`.

[Hsu, 1993] Hsu, M., editor (1993). *Special Issue on Workflow and Extended Transaction Systems*, volume 16. IEEE Computer Society, Washington, DC.

[Hsu, 1995] Hsu, M., editor (1995). *Special Issue on Workflow Systems*, volume 18. IEEE Computer Society, Washington, DC.

[Huang et al., 1991] Huang, J., Stankovic, J., Ramamritham, K., and Towsley, D. (1991). Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *Proceedings of the International Conference on Very Large Data Bases*, Barcelona, Spain.

[IBM, 1995] IBM (1995). Flowmark - managing your workflow, version 2.1. Document No. SH19-8243-00.

[Imielinski and Badrinath, 1994] Imielinski, T. and Badrinath, B. R. (1994). Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28.

[Informix, 1993] Informix (1993). *Informix-Online/Secure Security Features User's Guide.* Informix Software Inc., Menlo Park, CA.

[Ioannidis, 1996] Ioannidis, Y. (1996). Query optimization. *ACM Computing Surveys*, 28(1).

[Jablonski and Bussler, 1996] Jablonski, S. and Bussler, C. (1996). *Workflow Management - Modeling Concepts, Architecture, and Implementation.* Thomson Computer Press.

[Jagadish et al., 1997] Jagadish, H. V., Mumick, I. S., and Rabinovich, M. (1997). Scalable Versioning in Distributed Databases with Commuting Updates. In *Proceedings of the IEEE Conference on Data Engineering.*

[Jajodia and McCollum, 1993] Jajodia, S. and McCollum, C. (1993). Using two-phase commit for crash recovery in federated multilevel secure database management systems. In Landwehr, C. E. et al., editors, *Dependable Computing and Fault Tolerant Systems, Vol. 8*, pages 365–381. Springer-Verlag, New York.

[Jajodia et al., 1994] Jajodia, S., McCollum, C. D., and Blaustein, B. T. (1994). Integrating concurrency control and commit algorithms in distributed multilevel secure databases. In Keefe, T. F. and Landwehr, C. E., editors, *Database Security, VII: Status and Prospects*, pages 109–121. North-Holland, Amsterdam.

[Jarke and Koch, 1984] Jarke, M. and Koch, J. (1984). Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152.

[JavaSoft, 1996] JavaSoft (1996). Java^{TM} Core Reflection – API and Specification.

[Jensen and Soparkar, 1995] Jensen, P. and Soparkar, N. (1995). Real-Time Concurrency Control in Groupware. Technical Report CSE-TR-265-95, EECS department, The University of Michigan, Ann Arbor. Invited for ESDA'96 conference publication.

[Jensen et al., 1997] Jensen, P., Soparkar, N., and Mathur, A. (1997). Characterizing Multicast Orderings using Concurrency Control Theory. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, MD, USA.

[Jin et al., 1993] Jin, W., Ness, L., Rusinkiewicz, M., and Sheth, A. (1993). Concurrency Control and Recovery of Multidatabase Work Flows in Telecommunication Applications. In *Proceedings of ACM SIGMOD Conference.*

[Joosten et al., 1994] Joosten, S., Aussems, G., Duitshof, M., Huffmeijer, R., and Mulder, E. (1994). *WA-12: An Empirical Study about the Practice of Workflow Management.* University of Twente, Enschede, The Netherlands. Research Monograph.

[Jordan, 1996] Jordan, M. J. (1996). Early Experiences with PJava. In Atkinson, M. P. and Jordan, M., editors, *First International Workshop on Persistence and Java*, Drymen, Scotland. Sunlabs Technical Report.

[Kaiser and Pu, 1992] Kaiser, G. E. and Pu, C. (1992). Dynamic Restructuring of Transactions. In [Elmagarmid, 1992], chapter 8, pages 266–295.

[Kamath et al., 1996] Kamath, M., Alonso, G., Günthör, R., and Mohan, C. (1996). Providing High Availability in Very Large Workflow Management Systems. In *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT'96)*, Avignon, France. Also available as IBM Research Report RJ9967, IBM Almaden Research Center, July 1995.

[Kamath and Ramamritham, 1996a] Kamath, M. and Ramamritham, K. (1996a). Bridging the gap between Transaction Management and Workflow Management. In [Sheth, 1996]. Available from `http://LSDIS.cs.uga.edu/activities/NSF-workflow`.

[Kamath and Ramamritham, 1996b] Kamath, M. and Ramamritham, K. (1996b). Efficient Transaction Support for Dynamic Information Retrieval System. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, Zurich.*

[Kao and Garcia-Molina, 1992] Kao, B. and Garcia-Molina, H. (1992). An Overview of Real-Time Database Systems. In *Proceedings of NATO Advanced Study Institute on Real-Time Computing*, St. Maarten, Netherlands Antilles.

[Katz and et al., 1993] Katz, R. H. and et al. (1993). Design of a Large Object Server Supporting Earth System Science Researchers. In *AAAS Workshop on Adavances in Data Management for the Scientist and Engineer, Boston, Massachussetts, USA*, pages 77–83.

[Kaufmann and Schek, 1996] Kaufmann, H. and Schek, H. J. (1996). Extending TP-Monitors for Intra-Transaction Parallelism. In *Proceedings of the International Conference on Parallel and Distributed Information Systems, Miami Beach.*

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

[Kiczales et al., 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol.* MIT Press.

[Kim and Son, 1995] Kim, Y.-K. and Son, S. (1995). Predictability and Consistency in Real-Time Database Systems. In *Advances in Real-time Systems.* Prentice Hall.

[Kopetz and Grunsteidl, 1993] Kopetz, H. and Grunsteidl, G. (1993). A Protocol for Fault-tolerant Real-time Systems. *IEEE Computer*, 27(1).

[Korth and Silberschatz, 1991] Korth, H. and Silberschatz, A. (1991). *Database System Concepts.* McGraw Hill.

[Korth et al., 1990a] Korth, H., Soparkar, N., and Silberschatz, A. (1990a). Triggered Real-Time Databases with Consistency Constraints. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia. Also included in *Readings in Advances in Real-Time Systems*, IEEE Computer Society Press, 1993.

[Korth, 1995] Korth, H. F. (1995). The Double Life of the Transaction Abstraction: Fundamental Principle and Evolving System Concept. In *Proceed-*

ings of the International Conference on Very Large Data Bases, pages 2–6, Zurich, Switzerland.

[Korth et al., 1990b] Korth, H. F., Levy, E., and Silberschatz, A. (1990b). A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia.

[Kreifelts et al., 1991] Kreifelts, T., Hinrichs, E., Klein, K. H., Seuffert, P., and Woetzel, G. (1991). Experiences with the DOMINO Office Procedure System. In *Proceedings ECSCW '91*, pages 117–130. Amsterdam.

[Krishnakumar and Sheth, 1995] Krishnakumar, N. and Sheth, A. (1995). Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations. *Distributed and Parallel Databases*, 3(2):155–186.

[Krychniak et al., 1996] Krychniak, P., Rusinkiewicz, M., Chichocki, A., Sheth, A., and Thomas, G. (1996). Bounding the Effects of Compensation under Relaxed Multi-Level Serializability. *Distributed and Parallel Database Systems*, 4(4):355–374.

[Kumar, 1996] Kumar, V., editor (1996). *Performance of Concurrency Control Mechanism in Centralized Database Systems*. Prentice Hall Inc.

[Kung and Robinson, 1981] Kung, H. T. and Robinson, J. T. (1981). On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226.

[Kuo, 1996] Kuo, D. (1996). Model and Verification of a Data Manager Based on ARIES. *ACM Trans. on Database Systems*, pages 427–479.

[Kuo and Mok, 1992] Kuo, T.-W. and Mok, A. (1992). Concurrency Control for Real-Time Database Management. In *Proceedings of the Real-Time Systems Symposium*, Phoenix, AZ, USA.

[Lawler et al., 1992] Lawler, E., Lenstra, J., Kan, A., and Shmoys, D. (1992). Sequencing and Scheduling: Algorithms and Complexity. In *Handbooks in Operations Research and Management Science*, volume 4. North Holland Publishing Company.

[Lawler et al., 1993] Lawler, E. L., Lenstra, J. K., Kan, A. H. G. R., and Shmoys, D. (1993). Sequencing and Scheduling: Algorithms and Complexity. In *Handbooks in Operations Research and Management Science, Vol 4: Logistics of Production and Inventory*. North-Holland.

[Lee and Son, 1995] Lee, J. and Son, S. (1995). Performance of Concurrency Control Algorithms for Real-Time Database Systems. In Kumar, V., editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice Hall.

[Levy, 1991] Levy, E. (1991). Semantics-Based Recovery in Transaction Management Systems. Ph.D. dissertation. Department of Computer Sciences, University of Texas at Austin.

[Levy et al., 1991a] Levy, E., Korth, H., and Silberschatz, A. (1991a). A Theory of Relaxed Atomicity. In *Proceedings of the ACM Symposium on Principles of Distributed Computing.*

[Levy et al., 1991b] Levy, E., Korth, H., and Silberschatz, A. (1991b). An Optimistic Commit Protocol for Distributed Transaction Management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, USA.

[Leymann, 1995] Leymann, F. (1995). Supporting business transactions via partial backward recovery in workflow management systems. In *GI-Fachtagung Datenbanken in Büro Technik und Wissenschaft - BTW'95*, Dresden, Germany. Springer Verlag.

[Leymann et al., 1996] Leymann, F., Schek, H. J., and Vossen, G. (1996). Transactional workflows. Dagstuhl Seminar 9629.

[Lin et al., 1992] Lin, K.-J., Jahanian, F., Jhingran, A., and Locke, C. (1992). A Model of Hard Real-Time Transaction Systems. Technical Report RC No. 17515, IBM T.J. Watson Research Center.

[Lindholm and Yellin, 1996] Lindholm, T. and Yellin, F. (1996). *The Java^{TM} Virtual Machine Specification.* The Java^{TM} Series. Addison-Wesley.

[Liskov, 1988] Liskov, B. (1988). Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312.

[Lomet, 1992] Lomet, D. (1992). MLR: A Recovery Method for Multi-level Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego.*

[Lortz and Shin, 1993] Lortz, V. and Shin, K. (1993). MDARTS: A multiprocessor database architecture for real-time systems. Technical Report CSE-TR-155-93, EECS Department, The University of Michigan, Ann Arbor.

[Lynch et al., 1994] Lynch, N., Merritt, M., Weihl, W., and Fekete, A. (1994). *Atomic Transactions.* Morgan Kaufmann.

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

[Maffeis, 1996] Maffeis, S. (1996). PIRANHA: A Hunter of Crashed CORBA Objects. Technical report, Olsend & Associates, Zurich.

[Medina-Mora and Cartron, 1996] Medina-Mora, R. and Cartron, K. W. (1996). ActionWorkflow in Use: Clark County Department of Business License. In

Proceedings of the 12th. Intnl. Conference on Data Engineering, New Orleans, LA.

[Medina-Mora et al., 1993] Medina-Mora, R., Wong, H. K., and Flores, P. (1993). ActionWorkflow as the Enterprise Integration Technology. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 16(2).

[Mehrotra et al., 1992] Mehrotra, S., Rastogi, R., Breitbart, Y., Korth, H., and Silberschatz, A. (1992). The Concurrency Control Problem in Multidatabases: Characteristics and Solutions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA, USA.

[Meidanis et al., 1996] Meidanis, J., Vossen, G., and Weske, M. (1996). Using Workflow Management in DNA Sequencing. In *Proceedings of the 1st International Conference on Cooperative Information Systems (CoopIS96), Brussels, Belgium.*

[Mesquite, 1995] Mesquite (1995). *CSIM 17 User's Guide*. Mesquite Software Inc., Austin, Texas.

[Miller et al., 1996] Miller, J. A., Sheth, A. P., Kochut, K. J., and Wang, X. (1996). CORBA-based Run-Time Architectures for Workflow Management Systems. *Journal of Database Management, Special Issue on Multidatases*, 7(1):16–27.

[Mohan, 1992] Mohan, C. (1992). Interaction between query optimization and concurrency control. *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 26–35.

[Mohan, 1993] Mohan, C. (1993). A Survey of DBMS Research Issues in Supporting Very Large Tables. In *Proceedings of Data Organization and Algorithms*, pages 279–300, Chicago, Il.

[Mohan, 1994] Mohan, C. (1994). A Survey and Critique of Advanced Transaction Models. In *ACM SIGMOD International Conference on Management of Data, Minneapolis.* Tutorial Notes.

[Mohan, 1996] Mohan, C. (1996). Tutorial: State of the art in workflow management system research and products. 5th International Conference on Extending Database Technology, Avignon, March 1996 and at ACM SIGMOD International Conference on Management of Data, Montreal, June.

[Mohan et al., 1995] Mohan, C., Alonso, G., Günthör, R., and Kamath, M. (1995). Exotica: A Research Perspective on Workflow Management Systems. *In [Hsu, 1995]*, 18(1):19–26.

[Mohan and Dievendorff, 1994] Mohan, C. and Dievendorff, R. (1994). Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queuing. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 17(1):22–28.

[Mohan et al., 1992a] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992a). ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162.

[Mohan et al., 1986] Mohan, C., Lindsay, B., and Obermarck, R. (1986). Transaction Management in R* Distributed Database Management System. *ACM Transaction on Database Systems*, 11(4):378–396.

[Mohan et al., 1992b] Mohan, C., Pirahesh, H., and Lorie, R. (1992b). Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-only Transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego.*

[Molesky and Ramamritham, 1995] Molesky, L. D. and Ramamritham, K. (19-95). Recovery protocols for shared memory database systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, San Jose, Calif.

[Moss, 1982] Moss, J. (1982). Nested Transactions and Reliable Distributed Computing. In *Proceedings of the 2nd. Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39, Pittsburgh, PA. IEEE CS Press.

[Moss, 1987] Moss, J. (1987). Nested transactions: An introduction. In [Bhargava, 1987].

[Moss, 1981] Moss, J. E. B. (1981). *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, EECS Department, M.I.T.

[Moss, 1985] Moss, J. E. B. (1985). *Nested Transactions. An Approach to Reliable Distributed Computing*. Information Systems Series. The MIT Press, Cambridge, Massachussetts.

[Musa, 1995] Musa, S. (1995). Display Technology & Manufacturing, Annual Report, July 1994–June 1995. Technical report, Center for DT&M, College of Engineering, University of Michigan, Ann Arbor.

[Nodine and Zdonik, 1990] Nodine, M. and Zdonik, S. (1990). Cooperative transaction hierarchies: a transaction model to support design applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 83–94.

[Obermack, 1994] Obermack, R. (1994). Special Issue on TP Monitors and Distributed Transaction Management. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 17(1).

[OMG, 1995a] OMG (1995a). CORBAservices: Common Object Services Specification. Technical report, Object Management Group.

[OMG, 1995b] OMG (1995b). The Common Object Request Broker: Architecture and Specification, revision 2.0. Technical report, Object Management Group.

[O'Neil, 1986] O'Neil, P. (1986). The Escrow Transactional Method. *ACM Transaction on Database Systems*, 11(4):405–430.

[Owicki and Gries, 1976] Owicki, S. and Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285.

[Ozsoyoglu and Snodgrass, 1995] Ozsoyoglu, G. and Snodgrass, R. (1995). Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4).

[Ozsu and Valduiez, 1991] Ozsu, T. and Valduiez, P. (1991). *Prinicples of Distributed Database Systems*. Prentice-Hall.

[Palaniswami, 1997] Palaniswami, D. (1997). WebWork: The Web-based Distributed Engine for the METEOR$_2$ Workflow Management System. Master's thesis, University of Georgia, Athens, GA. In preparation.

[Palaniswami et al., 1996] Palaniswami, D., Lynch, J., Shevchenko, I., Mattie, A., and Reed-Fourquet, L. (1996). Web-based Multi-Paradigm Workflow Automation for Efficient Healthcare Delivery. In [Sheth, 1996]. Available from `http://LSDIS.cs.uga.edu/activities/NSF-workflow`.

[Papadimitriou, 1986] Papadimitriou, C. (1986). *The Theory of Database Concurrency Control*. Computer Science Press.

[Papadimitriou, 1988] Papadimitriou, C. H. (1988). *The Theory of Database Concurrency Control*. Computer Science Press.

[Pedregal Martin and Ramamritham, 1997] Pedregal Martin, C. and Ramamritham, K. (1997). Delegation: Efficiently Rewriting History. In *Proceedings of IEEE 13th International Conference on Data Engineering*, Birmingham, UK.

[Perry et al., 1996] Perry, D., Porter, A., Votta, L., and Wade, M. (1996). Evaluating Workflow and Process Automation in Wide-Area Software Development. In [Sheth, 1996]. Available from `http://LSDIS.cs.uga.edu/activities/NSF-workflow`.

[Pinedo, 1995] Pinedo, M. (1995). *Scheduling - Theory, Algorithms, and Systems*. Prentice Hall.

[Prakash and Knister, 1994] Prakash, A. and Knister, M. (1994). A Framework for Undoing Actions in Collaborative Systems. *ACM Transactions on Computer-Human Interactions*, 1(4).

[Pu et al., 1988] Pu, C., Kaiser, G. E., and Hutchinson, N. (1988). Split-Transactions for Open-Ended Activities. In *Proceedings of the International Conference on Very Large Data Bases*, pages 27–36, Los Angeles.

[Purimetla et al., 1995] Purimetla, B., Sivasankaran, R., Ramamritham, K., and Stankovic, J. (1995). Real-Time Databases: Issues and Applications. In Son, S., editor, *Advances in Real-time Systems*. Prentice Hall.

[Raab, 1995] Raab, F. (1995). *TPC Benchmark D*. Transaction Processing Performance Council.

[Rahm and Marek, 1995] Rahm, E. and Marek, R. (1995). Dynamic Multi-Resource Load Balancing in Parallel Database Systems. In *Proceedings of the International Conference on Very Large Data Bases*, Zurich, Switzerland.

[Ramamritham, 1993] Ramamritham, K. (1993). Real-Time Databases. *International Journal on Parallel and Distributed Databases*, 1(2).

[Ramamritham and Chrysanthis, 1992] Ramamritham, K. and Chrysanthis, P. K. (1992). In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In Gupta, A., editor, *Distributed Object Management*, pages 212–230. Morgan Kaufmann Publishers.

[Ramamritham and Chrysanthis, 1996] Ramamritham, K. and Chrysanthis, P. K. (1996). A Taxonomy of Correctness Criteria in Database Applications. *The VLDB Journal*, 5:85–97.

[Ramamrithan and Pu, 1995] Ramamrithan, K. and Pu, C. (1995). A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6).

[Rastogi et al., 1995] Rastogi, R., Korth, H. F., and Silberschatz, A. (1995). Exploiting transaction semantics in multidatabase systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 101–109, Vancouver, Canada.

[Ray et al., 1996] Ray, I., Bertino, E., Jajodia, S., and Mancini, L. V. (1996). An Advanced Commit Protocol for MLS Distributed Database Systems. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 119–128.

[Reddy et al., 1993] Reddy, E. et al. (1993). Computer Support for Concurrent Engineering. *IEEE Computer*, 26(1).

[Reinwald and Mohan, 1996] Reinwald, B. and Mohan, C. (1996). Structured Workflow Management with Lotus Notes Release 4. In *Proceedings of 41st. IEEE Computer Society Intnl. Conference*, pages 451–457, Santa Clara, CA.

[Reuter, 1989] Reuter, A. (1989). ConTracts: A Means for Extending Control Beyond Transaction Boundaries. In *Proceedings of the 3rd. International Workshop on High Performance Transaction Systems*, Asilomar.

[Reuter and Schwenkreis, 1995] Reuter, A. and Schwenkreis, F. (1995). ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems. *IEEE Data Engineering Bulletin*, 18(1).

[Rusinkiewicz and Sheth, 1995] Rusinkiewicz, M. and Sheth, A. (1995). Specification and Execution of Transactional Workflows. In Kim, W., editor, *Modern Database Systems: The Object Model, Interoperability and Beyond.* ACM Press, New York, NY.

[Saastamoinen, 1995] Saastamoinen, H. (1995). *On the Handling of Exceptions in Information Systems*. PhD thesis, University of Jyvaskyla.

[Samaras et al., 1995] Samaras, G., Britton, K., Citron, A., and Mohan, C. (1995). Two-phase optimizations in a commercial distributed environment. *International Journal of Distributed and Parallel Databases*, 3(4):325–360.

[Schaad et al., 1995] Schaad, W., Schek, H.-J., and Weikum, G. (1995). Implementation and Performance of Multi-level Transaction Management in a Multidatabase Environment. In *Proceedings of the 5. Int. Workshop on Research Issues on Data Engineering, Distributed Object Management, Taipei, Taiwan.*

[Schuster et al., 1994] Schuster, H., Jablonski, S., Kirsche, T., and Bussler, C. (1994). A Client/Server Architecture for Distributed Workflow Management Systems. In *Proceedings of the 3rd. International. Conference on Parallel and Distributed Information Systems*, pages 253–256, Austin, TX. IEEE CS Press.

[Schwenkreis and Reuter, 1996] Schwenkreis, F. and Reuter, A. (1996). Synchronizing Long-Lived Computations. In [Kumar, 1996], chapter 12.

[Sevcik, 1994] Sevcik, K. C. (1994). Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. *Performance Evaluation*, 19:107–140.

[Shasha et al., 1995] Shasha, D., Llirbat, F., Simon, E., and Valduriez, P. (1995). Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems*, 20(3):325–363.

[Sheth, 1995] Sheth, A. (1995). Tutorial Notes on Workflow Automation: Application, Technology and Research. Technical report, University of Georgia. presented at ACM SIGMOD, San Jose, CA, URL: `http://LSDIS.cs.uga.edu/publications`.

[Sheth, 1996] Sheth, A., editor (1996). *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, Athens, GA. University of Georgia. Available from `http://LSDIS.cs.uga.edu/activities/NSF-workflow`.

[Sheth et al., 1996a] Sheth, A., Georgakopoulos, D., Joosten, S., Rusinkiewicz, M., Scacchi, W., Wileden, J., and Wolf, A. (1996a). Report from the NSF

Workshop on Workflow and Process Automation in Information Systems. Technical report, University of Georgia, UGA-CS-TR-96-003. URL: `http://LSDIS.cs.uga.edu/activities/NSF-workflow`.

[Sheth and Joosten, 1996] Sheth, A. and Joosten, S. (1996). Workshop on Workflow Management: Research, Technology, Products, Applications and Experiences.

[Sheth et al., 1996b] Sheth, A., Kochut, K. J., Miller, J., Worah, D., Das, S., Lin, C., Palaniswami, D., Lynch, J., and Shevchenko, I. (1996b). Supporting State-Wide Immunization Tracking using Multi-Paradigm Workflow Technology. In *Proceedings of the International Conference on Very Large Data Bases*, Bombay, India.

[Sheth and Rusinkiewicz, 1993] Sheth, A. and Rusinkiewicz, M. (1993). On Transactional Workflows. *In [Hsu, 1993]*.

[Shin, 1991] Shin, K. (1991). HARTS: A distributed real-time architecture. *IEEE Computer*, 24(5).

[Shrivastava et al., 1991] Shrivastava, S. K., Dixon, G. N., and Parrington, G. D. (1991). An overview of Arjuna: A programming system for reliable distributed computing. *IEEE Software*, 8(1):66–73.

[Shrivastava and Wheater, 1990] Shrivastava, S. K. and Wheater, S. M. (1990). Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-coloured Actions. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 203–210, Paris, France.

[Silberschatz et al., 1996] Silberschatz, A., Stonebraker, M., and Ullman, J. (1996). Database Research: Achievements and opportunities into the 21st century. Report of the NSF Workshop on the Future of Database Systems Research. *ACM SIGMOD Record*, 25(1):52–63.

[Silver, 1995] Silver, B. R. (1995). *The Guide to Workflow Software: A Visual Comparison of Today's Leading Products*. BIS Strategic Decisions.

[Singhal, 1988] Singhal, M. (1988). Issues and Approaches to Design of Real-time Database Systems. *ACM SIGMOD Record*, 17(1).

[Smith and Liu, 1989] Smith, K. and Liu, J. (1989). Monotonically Improving Approximate Answers to Relational Algebra Queries. In *Proceedings of the International Computer Software Applications Conference*, Orlando, FL, USA.

[Smith, 1993] Smith, T. (1993). The Future of Workflow Software. *INFORM*, pages 50–51.

[Son, 1988] Son, S., editor (1988). *ACM SIGMOD Record: Special Issue on Real-Time Databases*. ACM Press.

[Son, 1996] Son, S. H. (1996). *Proceedings of the 1st International Workshop on Real-Time Databases.*

[Song and Liu, 1990] Song, X. and Liu, J. (1990). Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency. Technical report, University of Illinois at Urbana-Champaign.

[Soparkar et al., 1991] Soparkar, N., Korth, H., and Silberschatz, A. (1991). Failure-resilient Transaction Management in Multidatabases. *IEEE Computer*, 24(12).

[Soparkar et al., 1995a] Soparkar, N., Korth, H., and Silberschatz, A. (1995a). Autonomous Transaction Managers in Responsive Computing. In *Responsive Computer Systems: Toward Integration of Fault-Tolerance and Real-Time*. Kluwer Academic Publishers.

[Soparkar et al., 1995b] Soparkar, N., Korth, H., and Silberschatz, A. (1995b). Databases with Deadline and Contingency Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 7(4).

[Soparkar et al., 1996] Soparkar, N., Korth, H., and Silberschatz, A. (1996). *Time-Constrained Transaction Management: Real Time Constraints in Database Transaction Systems.* Kluwer Academic Publishers.

[Soparkar et al., 1994] Soparkar, N., Levy, E., Korth, H., and Silberschatz, A. (1994). Adaptive Commitment for Real-Time Distributed Transactions. In *Proceedings of the 3rd International Conference on Information and Knowledge Management.*

[Soparkar and Ramamritham, 1996] Soparkar, N. and Ramamritham, K. (1996). DART'96. In *Proceedings of the International Workshop on Databases: Active and Real-Time (Concepts meet Practice).*

[Spivey, 1992] Spivey, J. M. (1992). *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, Englewood Cliffs, NJ.

[Stamos and Cristian, 1993] Stamos, J. W. and Cristian, F. (1993). Coordinator Log Transaction Execution Protocol. *Distributed and Parallel Databases*, 1:383–408.

[Stankovic, 1988] Stankovic, J. (1988). Misconceptions About Real-Time Computing. *IEEE Computer*, 21(10).

[Sullivan and Notkin, 1992] Sullivan, K. and Notkin, D. (1992). Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268.

[Swenson et al., 1994] Swenson, K. D., Maxwell, R. J., Matsumoto, T., Saghari, B., and Irwin, K. (1994). A business process environment supporting collaborative planning. *Journal of Collaborative Computing*, 1(1).

[Tang and Veijalainen, 1995] Tang, J. and Veijalainen, J. (1995). Transaction-oriented Work-flow Concepts in Inter-organizational Environments. In *Proceedings of the 4th. International. Conference on Information and Knowledge Management*, Baltimore, MD.

[Technologies, 1995] Technologies, A. (1995). Metro Tour. Technical report, Action Technologies, Inc. URL: `http://www.actiontech.com/`.

[Texas Instruments, 1993] Texas Instruments (1993). *Open OODB Toolkit Release 0.2 (Alpha) Document*. Texas Instruments, Dallas.

[Thé, 1994] Thé, L. (1994). Getting Into the Workflow. *Datamation*, October.

[Thomas, 1979] Thomas, R. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209.

[Transarc, 1995] Transarc (1995). Writing encina applications. Transarc Corporation, ENC-D5012-00.

[Tsichritzis, 1982] Tsichritzis, D. (1982). Form Management. *Communications of the ACM*, 25(7):453–478.

[Turek et al., 1994] Turek, J., Ludwig, W., Wolf, J. L., Fleischer, L., Tiwari, P., Glasgow, J., Schwiegelshohn, U., and Yu, P. S. (1994). Scheduling Parallelizable Tasks to Minimize Average Response Time. In *Proceedings of the Symposium on Parallel Algorithms and Architectures, Cape May, New Jersey*.

[Turek et al., 1992] Turek, J., Wolf, J. L., Pattipati, K. R., and Yu, P. S. (1992). Scheduling Parallelizable Tasks: Putting it All on the Shelf. In *Proceedings of the ACM SIGMETRICS Conference*.

[Ulusoy, 1992] Ulusoy, O. (1992). Scheduling Real-Time Database Transactions. PhD Thesis. Department of Computer Science, University of Illinois at Urbana-Champaign.

[Unland and Schlageter, 1992] Unland, R. and Schlageter, G. (1992). A Transaction Manager Development Facility for Non Standard Database Systems. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*, chapter 11. Morgan Kaufmann.

[Vivier et al., 1996] Vivier, B., Haimowitz, I., and Luciano, J. (1996). Workflow Requirements for Electronic Commerce in a Distributed Health Care Enterprise. In [Sheth, 1996]. Available from `http://LSDIS.cs.uga.edu/activities/NSF-workflow`.

[Waechter and Reuter, 1992] Waechter, H. and Reuter, A. (1992). The ConTract Model. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, San Mateo.

[Wang, 1995] Wang, X. (1995). Implementation and Performance Evaluation of CORBA-Based Centralized Workflow Schedulers. Master's thesis, University of Georgia.

[Weihl, 1984] Weihl, W. E. (1984). *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA.

[Weihl, 1988a] Weihl, W. E. (1988a). 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. IEEE Computer Society.

[Weihl, 1988b] Weihl, W. E. (1988b). Commutativity-based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488–1505.

[Weikum, 1991] Weikum, G. (1991). Principles and Realization Strategies of Multi-Level Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180.

[Weikum, 1993] Weikum, G. (1993). Extending Transaction Management to Capture More Consistency With Better Performance. In *Proceedings of the 9th. French Database Conference*, pages 27–30, Toulouse.

[Weikum and Hasse, 1993] Weikum, G. and Hasse, C. (1993). Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism. *The VLDB Journal*, 2(4):407–453.

[Weikum et al., 1994] Weikum, G., Hasse, C., Moenkeberg, A., and Zabback, P. (1994). The COMFORT Automatic Tuning Project. *Information Systems*, 19(5).

[Weikum and Schek, 1992] Weikum, G. and Schek, H. (1992). Concepts and applications of multilevel transactions and open-nested transactions. In [Elmagarmid, 1992], chapter 13.

[Weissenfels et al., 1996] Weissenfels, J., Wodtke, D., Weikum, G., and Kotz-Dittrich, A. (1996). The Mentor Architecture for Enterprise-wide Workflow Management. In *Proceedings of the NSF International Workshop on Workflow and Process Automation in Information Systems, Athens, Georgia.*

[WFMC, 1994] WFMC (1994). Glossary, A Workflow Management Coalition Specification. Technical report, The Workflow Management Coalition. Workflow Management Coalition accessible via: `http://www.aiai.ed.ac.uk/WfMC/`.

[Widom, 1995] Widom, J. (1995). Special Issue on Materialized Views and Data Warehousing. *IEEE Data Engineering Bulletin*, 18(2).

[Wodtke et al., 1996] Wodtke, D., Weissenfels, J., Weikum, G., and Kotz-Dittrich, A. (1996). The Mentor Project: Steps Towards Enterprise-Wide Workflow

Management. In *Proceedings of 12th. IEEE International Conference on Data Engineering*, pages 556–565, New Orleans, LA.

[Wolf et al., 1995] Wolf, J. L., Turek, J., Chen, M. S., and Yu, P. S. (1995). A Hierarchical Approach to Parallel Multiquery Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):578–590.

[Worah, 1997] Worah, D. (1997). Error Handling and Recovery in the METEOR$_2$ Workflow Management System. Master's thesis, University of Georgia, Athens, GA. In preparation. URL: `http://LSDIS.cs.uga.edu/`.

[Worah and Sheth, 1996] Worah, D. and Sheth, A. (1996). What Do Advanced Transaction Models Have to Offer for Workflows? In *Proceedings of the International Workshop on Advanced Transaction Models and Architectures, Goa, India.*

[Wu and Schwiderski, 1996] Wu, Z. and Schwiderski, S. (1996). Design of Reflective Java. Technical Report APM.1818.00.05, APM Limited, Poseidon house, Castle Park, Cambridge, CB3 0RD, U.K. ANSA Work Programme.

[Zdonik et al., 1994] Zdonik, S., Alonso, R., Franklin, M., and Acharya, S. (1994). Are Disks in the Air Just Pie in the Sky. In *Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA.*

[Zhang et al., 1994a] Zhang, A., Nodine, M., Bhargava, B., and Bukhres, O. (1994a). Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proceedings 1994 SIGMOD International Conference on Management of Data*, pages 67–78.

[Zhang et al., 1994b] Zhang, A., Nodine, M., Bhargava, B., and Bukhres, O. (1994b). Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 67–78.

[Zhou et al., 1996] Zhou, L., Shin, K., Rundensteiner, E., and Soparkar, N. (1996). Probabilistic Real-Time Data Access with Interval Constraints. In *Proceedings of the 1st International Workshop on Real-Time Databases: Issues and Applications.* Book chapter in Real-Time Databases Systems: Issues and Applications. To appear in 1997.

[Zisman, 1978] Zisman, M. (1978). Office automation: Evolution or revolution. *Sloan Management Review*, 19(3):1–16.

Contributing Authors

Dr. Gustavo Alonso is a senior research associate at the Swiss Federal Institute of Technology in Zürich (ETH). He received his M.S. (1992) and Ph.D. (1994) degrees in computer science from University of California at Santa Barbara and an engineering degree in telecommunications (1989) from Madrid Technical University (UPM). Current address: Institute of Information Systems, ETH Zentrum, CH-8092, Zürich, Switzerland. Dr. Alonso can be reached by e-mail at `alonso@inf.ethz.ch`.

Dr. Paul Ammann is an Associate Professor of Information and Software Systems Engineering at George Mason University in Fairfax, Virginia. He received the AB in Computer Science summa cum laude from Dartmouth College, and the MS and PhD in Computer Science from the University of Virginia. He has published over 35 refereed technical papers. His research interests center around why computing systems fail and what can be done about it. Current areas of interest are formal methods, software for critical systems, software testing, and computer security. The URL for his web page is `http://www.isse.gmu.edu/faculty/pammann` and his e-mail address is `pammann@gmu.edu`.

Dr. Eman Anwar received a BSc. degree in computer science from Kuwait University, Kuwait in 1989. She received both a M.E. and Ph.D degree from the University of Florida in 1992 and 1996, respectively. Her current research interests include active databases, distributed object-oriented databases and transaction processing. Eman Anwar joined Transarc Corporation as a Member of Technical Staff in August 1996. She is currently working on improving the performance of recoverable multi-threaded applications on shared and shared nothing multi-processor machines. She has published several papers in refereed journals and conferences proceedings in the area of active databases and transaction processing. Eman Anwar can be contacted by e-mail at eman@transarc.com.

Dr. Malcolm Atkinson is currently a professor at the University of Glasgow and leader of the PJava team. He proposed the concept of orthogonal persistence in 1978, was jointly responsible for the first orthogonally persistent language, PS-algol, and was the initiator of the international workshop series on Persistent Object Systems. He has worked with O2 Technology and is working at Sun Labs in the summer of 1997. Professor Atkinson can be reached at Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.

Roger Barga is currently a Ph.D. candidate at the Oregon Graduate Institute. His research interests include transaction processing, concurrency control mechanisms, storage and management of scientific data, and new techniques to construct extensible and flexible systems software. His current address is Department of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR 97291, USA. He can be reached by e-mail at barga@cse.ogi.edu.

Elisa Bertino is professor of computer science in the Department of Computer Science of the University of Milan where she heads the Database Systems Group. She has also been professor in the Department of Computer and Information Science of the University of Genova, Italy. Her main research interests include object-oriented databases, deductive databases, multimedia databases, interoperability of heterogeneous systems, database security. She is on the editorial board of the following scientific journals: IEEE Transaction on Knowledge and Data Engineering, International Journal of Theory and Practice of Object Systems, Journal of Computer Security, VLDB Journal, and International Journal of Parallel and Distributed Databases. Professor Bertino's current address is Dipartimento di Scienze dell'Informazione, Università di Milano, Via Comelico 39/41, 20135 Milano, Italy. Her e-mail address is bertino@dsi.unimi.it.

Dr. Sharma Chakravarthy received the B.E. degree in Electrical Engineering from the Indian Institute of Science, Bangalore, India in 1973. He received M.S. and Ph.D degrees from the University of Maryland in College park in 1981 and 1985, respectively. Currently, he is Associate Professor in the Computer and Information Science and Engineering department at the University of Florida, Gainesville. His current research interests are: active and real-time databases, data mining, and workflow management. He is listed in Who's Who

Among South Asian Americans and Who's Who Among America's Teachers. Professor Chakravarthy can be reached at: CISE Department, E470 CSE Building, University of Florida, Gainesville, FL 32611-6125, and by e-mail at sharma@cise.ufl.edu.

Dr. Laurent Daynès received his Ph.D. degree in Computer Science from the University of Paris 6 in 1995. From 1991 to 1995, he has worked at INRIA in Patrick Valduriez's research group. There, he participated in the group's research on transaction management for persistent programming languages. He is a research fellow at the University of Glasgow since January 1996, where he is the main designer and implementor of the Persistent Java system.

Dr. Ahmed Elmagarmid is a professor of Computer Science at Purdue University. He founded the workshop series on Research Issues in Data Engineering, was a founding member of the International Endowment on Cooperative Information Systems. He is the editor-in-chief of Distributed and Parallel Databases. He has published close to 100 papers and 4 books. Dr. Elmagarmid has been an IEEE Distinguished Lecturer, an NSF Presidential Young Investigator and he has been named a Distinguished Alumni by the college of Engineering at Ohio State University College. He is the principal investigator of the InterBase Project and was responsible for its transfer to many research labs. InterBase has influenced the Carnot project at MCC and has been extensively used by BNR and GTE. He was a Chief Architect of the XA/21 DBMS produced for SES by Harris Corporation.

Christof Hasse received the M.Sc. degree (Dipl.-Inform.) from the University of Darmstadt, Germany, in 1989 and the Ph.D. degree (Dr.-Ing.) from the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland, in 1995. Dr. Hasse is a database specialist in the Business Information Technology department of the Union Bank of Switzerland where he is working on the worldwide real-time trading and risk management system. He is responsible for the logical and physical database design, database performance analysis, database tuning, and application design.

Dr. Abdelsalam Heddaya obtained his B.Sc. degree in Computer Engineering and Automatic Control from Alexandria University, Egypt in 1980 and his S.M. and Ph.D. degrees in Computer Science from Harvard University in 1983 and 1988, respectively. Since then, he has been on the Computer Science faculty at Boston University. Dr. Heddaya's research in distributed systems focuses on highly available and reliable systems, and on realizing parallel speedups for computations running on distributed systems. He has led project Mermera, which produced a model and system for specifying, reasoning about, and programming mixed-coherence distributed shared memory. Among his current research projects is WebWave, a model and protocol for large scale caching of published documents on the Internet. He has worked on congestion control, bulk-synchronous parallelism, and software cache coherence for multiprocessors.

Dr. Abdelsalam (Sumi) Helal received the B.Sc. and M.Sc. degrees in Computer Science and Automatic Control from Alexandria University, Alexandria, Egypt, and the M.S. and Ph.D. degrees in Computer Sciences from Purdue University, West Lafayette, Indiana. Before joining MCC to work on the Collaboration Management Infrastructure project (CMI), he was an Assistant Professor at the University of Texas at Arlington, and later, a Visiting Professor of Computer Sciences at Purdue University. His research interests include large-scale systems, fault-tolerance, OLTP, mobile data management, heterogeneous processing, standards and interoperability, and performance modeling. His current research deals with developing scalable data and transaction management protocols for large-scale distributed system, and for wireless and nomadic environments. Dr. Helal is a member of the Executive Committee of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments (TCOS). He is also the Editor-in-Chief of the TCOS quarterly Bulletin. Dr. Helal can be reached at helal@mcc.com.

Dr. Sushil Jajodia is Director of Center for Secure Information Systems and Professor of Information and Software Systems Engineering at the George Mason University, Fairfax, Virginia. His research interests include information security, temporal databases, and replicated databases. He received the 1996 Kristian Beckman award from IFIP TC 11 for his contributions to the discipline of Information Security. Dr. Jajodia has served in different capacities for various journals and conferences. He is the founding co-editor-in-chief of the Journal of Computer Security. He is on the editorial boards of IEEE Concurrency and International Journal of Cooperative Information Systems and a contributing editor of the Computer & Communication Security Reviews. He has been named a Golden Core member for his service to the IEEE Computer Society. He is a past chairman of the IEEE Computer Society Technical Committee on Data Engineering and the Magazine Advisory Committee. He is a senior member of the IEEE and a member of IEEE Computer Society and Association for Computing Machinery. Dr. Jajodia's home page on the web is http://www.isse.gmu.edu/~csis/faculty/jajodia.html and he can be reached by e-mail at jajodia@gmu.edu.

Paul Jensen is a PhD candidate at the University of Michigan, Ann Arbor. His research interests include transaction management and process group communications. Paul Jensen holds an IBM Graduate Fellowship, and has worked at Chrysler Technology Center and IBM Toronto Laboratory. He can be reached by e-mail at `pjensen@eecs.umich.edu`.

Yoo-Sung Kim is an assistant professor at INHA University, Korea. He is a member of ACM and IEEE. He received his Ph.D. in computer science from Korea Advanced Institute of Science and Technology(KAIST) in 1992. His current research interests include multi-database, mobile database and workflows. Dr. Kim can be reached by regular mail at Department of Computer Science & Engineering INHA University INCHON 402-751, Korea, and by e-mail to `yskim@dragon.inha.ac.kr`.

Dr. Luigi Mancini received the Laurea degree in Computer Science from the University of Pisa, Italy, in 1983, and the Ph.D. degree in Computer Science from the University of Newcastle upon Tyne, Great Britain, in 1989. Since 1996 he has been an Associate Professor of the Dipartimento di Scienze dell'Informazione of the University "La Sapienza" of Rome. His research interests include distributed algorithms and systems, transaction processing systems, and computer and information security. Professor Mancini can be reached by e-mail at `mancini@dsi.uniroma1.it` and by regular mail at Dipartimento di Scienze dell'Informazione, Università La Sapienza di Roma, Via Salaria, 113, 00198 Roma - Italy.

Cris Pedregal Martin is a doctoral candidate at the University of Massachusetts, Amherst, USA. He obtained his Master's degree from the University of Massachusetts, Amherst in 1993 and his Licenciado en Informática degree from ESLAI, Argentina. His current interests include recovery of database transaction systems.

Dr. C. Mohan recipient of the 1996 ACM SIGMOD Innovations Award, is the founder and leader of the Exotica workflow project at IBM Almaden. He is the primary inventor of the ARIES family of locking and recovery algorithms, and the Presumed Abort commit protocol. Dr. Mohan is an editor of the VLDB Journal and Distributed and Parallel Databases - An International Journal. He has received one IBM Corporate Award and seven IBM Outstanding Innovation Awards. He is an inventor on twenty eight issued and two pending patents. Dr. Mohan received his Ph.D. from the University of Texas at Austin. His current address is K55/B1, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA and can be reached by e-mail at `mohan@almaden.ibm.com`.

Arvind H Nithrakashyap is a graduate student at the University of Massachusetts, Amherst. He obtained his B.Tech degree in Computer Science and Engineering from the Indian Institute of Technology, Madras, India in 1995. His research interests include transaction processing and active databases and he can be reached by e-mail at nithraka@cs.umass.edu.

Dr. Marian Nodine is the project coordinator for the Infosleuth project at MCC. She received her Ph.D. from Brown University in 1993, and her S.B. and S.M. from MIT in 1981. Prior to joining MCC, Marian worked as a post-doctoral research associate and an adjunct assistant professor at Brown University. Her primary areas of interest include object-oriented query optimization and advanced database transaction models. She also worked at BBN in data communication and internet monitoring and management. Dr. Nodine has published over 15 papers in journals, conferences and books and is a member of ACM.

Jitendra Padhye received his B.E. degree from Victoria Jubilee Technical Institute, Bombay, India, and his M.S. degree from Vanderbilt University, Nashville, Tennessee. He is currently a PhD candidate in the department of computer science at the University of Massachusetts, Amherst. His research interests include multimedia systems, database systems, performance evaluation and computer networks. He can be reached at jitu@cs.umass.edu.

Dr. Calton Pu was born in Taiwan, but grew up in Brazil. He is currently Professor at the Oregon Graduate Institute. His research interests are in operating systems (Synthesis and Synthetix projects), extended transactions (Split/Join and Epsilon Serializability), and large distributed system interoperability (DIOM and Continual Queries). His current address is Department of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR 97291, USA. Professor Pu can be reached by e-mail at calton@cse.ogi.edu.

Dr. Krithi Ramamritham is a professor at the University of Massachusetts, Amherst, USA. Professor Ramamritham's interests span the areas of real-time systems, transaction processing in databases and real-time databases. He is an editor of the IEEE Transactions on Distributed and Parallel Systems, Real-Time Systems Journal, International Journal of Applied Software Technology, and the Distributed Systems Engineering Journal. He has co-authored two tutorial texts on hard real-time systems and a text on advances in database transaction processing.

Indrajit Ray received his B.E. degree from Bengal Engineering College, University of Calcutta, India in 1988 and his M.E. from Jadavpur University, Calcutta, India in 1991, both in Computer Science. At present he is a doctoral candidate at George Mason University, Fairfax, VA. He expects to receive his Ph.D. in Summer 1997. Indrajit's research interests include transaction processing, concurrency control, information systems security and data and knowledge based systems. He can be reached by e-mail at `iray@isse.gmu.edu`.

Indrakshi Ray graduated from Bengal Engineering College, University of Calcutta, India with a B.E. degree in Computer Science in 1988 and from Jadavpur University, Calcutta, India in 1991 with an M.E. degree also in Computer Science. Currently she is a doctoral candidate at George Mason University, Fairfax, VA, with an expected graduation datae of Summer 1997. Her research interests include database management systems, transaction processing, software requirement specification and verification and formal methods. She can be reached by e-mail at `indrakshi@isse.gmu.edu`.

Dr. Andreas Reuter is a professor for computer science at the University of Stuttgart. In 1988 he established the Institute of Parallel and Distributed High-Performance Systems. His research interests are: Transaction processing, database technology, parallel programming, and performmance analysis. His address is Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany. Professor Reuter can be reached by e-mail at `reuter@informatik.uni-stuttgart.de`.

Kerstin Schneider received her M.Sc. degree (Dipl.-Inform.) in computer science from the University of Kaiserslautern in 1994. Since then she is working as a research assistant at the IPVR at Stuttgart University. Her research interests are reliable workflow systems. Her address is Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany. She can be reached by e-mail at `kerstin.schneider@informatik.uni-stuttgart.de`.

Friedemann Schwenkreis received a diploma in computer science from the University of Stuttgart. He was a scientific staff member of the IPVR, University of Stuttgart. Since 1997 he is working for IBM Deutschland Entwicklung GmbH as a scientific consultant. His address is Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany. She can be reached by e-mail at schwenkreis@informatik.uni-stuttgart.de.

Jayavel Shanmugasundaram received his B.E. degree in Computer Science and Engineering from the Regional Engineering College, Tiruchirappalli, India in 1995. He is currently a graduate student in the Department of Computer Science at the University of Massachusetts, Amherst. His research interests include transaction processing, data mining and object management for advanced database applications. He can be reached at shan@cs.umass.edu.

Dr. Amit Sheth directs the Large Scale Distributed Information Systems Lab (LSDIS, http://lsdis.cs.uga.edu) and is an Associate Professor of Computer Science at the University of Georgia, Athens, GA. His research interests include multiparadigm transactional workflow (project METEOR), management of heterogeneous digital data and semantic issues in global information systems (projects InfoHarness & InfoQuilt), and interoperable information systems involving intelligent integration of collaboration, collaboration and information management technologies. Professor Sheth has approximately 100 publications to his credit, given over 50 invited and colloquia talks and 15 tutorials and professional courses, and lead four international conferences and a workshop as a General/Program (Co-)Chair in the area of information system cooperation/interoperability, workflow management, and parallel and distributed information systems, has served on over thirty five program and organization committees, is on the editorial board of five journals, and has served twice as an ACM Lecturer.

Rajendran M Sivasankaran is a Ph.D. student in the Department of Computer Science at the University of Massachusetts, Amherst. He received his bachelors degree from the Birla Institute of Technology and Science, India. His research interest include real-time databases, active databases, query processing and high performance databases and he can be reached at sivasank@cs.umass.edu.

Dr. Nandit Soparkar is in the faculty at the University of Michigan, Ann Arbor. He obtained a Ph.D. from the University of Texas at Austin, and has worked at the AT&T Bell Laboratories at Murray Hill. Nandit Soparkar has authored several research papers and books, and has organized research activities in his areas of research interest which include time-constrained transaction management, data-mining technology, and logic-enhanced hardware memory. He can be reached by e-mail at soparkar@eecs.umich.edu.

Malek Tayara is a Ph.D. student at the University of Michigan, Ann Arbor. He has research interests in distributed databases and real-time communications. His current work is on distributed coordination for reconfigurable automated manufacturing systems. Malek Tayara has work experience at Intel and I-Cube.

Dr. Patrick Valduriez received his Ph.D. degree and Doctorat d'Etat in Computer Science from the University of Paris in 1981 and 1985, respectively. He is currently a Director of Research at INRIA, the national research center for computer science in France. There he heads a group of 20 researchers working on advanced database technology including distributed, parallel, active, and object-oriented database systems. Since 1995, he is also heading the R&D joint venture Dyade between Bull and Inria to foster technology transfer in the areas of Internet/Intranet. He is an associate editor of several journals including ACM Transactions on Database Systems, the VLDB Journal and Distributed and Parallel Databases. In 1994, he was elected a trustee of the VLDB endowment. He can be reached by e-mail at `Patrick.Valduriez@inria.fr`.

Marisa Viveros is an advisory software engineer at IBM T.J. Watson Research Center in Hawthorne, New York. She received her M.S. in Computer Science from California State University, and her B.S. in Electrical Engineering from the University of Concepcion, Chile. Her research areas include parallel databases, decision support, data mining and transaction processing. Prior to IBM, Ms. Viveros worked as an independent consultant, and at AT&T/Teradata.

Dr. Gerhard Weikum received the M.Sc. degree (Dipl.-Inform.) and the Ph.D. degree (Dr.-Ing.) both in computer science from the University of Darmstadt, Germany, in 1982 and 1986, respectively. Dr. Weikum is a Full Professor in the Department of Computer Science of the University of the Saarland at Saarbruecken, Germany, where he is leading a research group on database systems. Dr. Weikum's research interests include parallel and distributed information systems, transaction processing and workflow management, and database optimization and performance evaluation. Dr. Weikum serves on the editorial boards of ACM Transactions on Database Systems, The VLDB Journal, and the Distributed and Parallel Databases Journal. Professor Weikum can be reached by e-mail at `weikum@cs.uni-sb.de`

Devashish Worah is a senior component-software engineer in the Professional Services group at I-Kinetics, Inc., Burlington, MA. He is involved with the analysis, design, and implementation of large-scale, CORBA and WWW-based distributed information systems, and in defining a methodology for transitioning legacy systems to objec t-based distributed environments. He completed his B.S. in Computer Science and Mathematics at Georgia College and State University, Milledgeville, GA and is working towards completing his Master's dissertation at the University of Georgia. His current research interests include workflow technology, distributed object computing, reliability in distributed systems, and legacy integration.

Ming Xiong is a Ph.D. candidate in the Department of Computer Science at the University of Massachusetts, Amherst. He received his B.S. degree in Computer Science and Engineering from Xi'an Jiaotong University, Xi'an, China in 1990 and his M.S. degree in Computer Science from the University of Massachusetts at Amherst in 1996. His research interests include databases and real-time systems. He is a student member of ACM and IEEE. Ming Xiong can be reached at `xiong@cs.umass.edu`.

Index