MARCH 1991, VOLUME 14, NO. 1

a quarterly bulletin of the
IEEE Computer Society
technical committee on

Data Engineering

CONTENTS

SPECIAL ISSUE ON UNCONVENTIONAL TRANSACTION MANAGEMENT

MARCH 1991, VOLUME 14, NO. 1

Data Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Data Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Data Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Data Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Message from the TC Chair

A professional group exists only to serve its members. The service is primarily by members for other members. A chair is not a sage who divines the needs of members, nor a slave to do all the work. A chair can be collection point for needs, problems and dreams.

As your chair I ask you to communicate with me as part of what you do for yourself through the technical committee. Send me mail or email, fax or phone, bend my ear at conferences, or drop by as you tour beautiful Minnesota.

What can the TC do for you? How do you want to serve the TC? Are we doing things well? Are we doing the right things? Are we reaching the right audience? Who else should join and be an active member? What questions should we be asking?

John Carlis Computer Science Dept. University of Minnesota 200 Union St SE Minneapolis, MN 55455 (612)-625-6092 (612)-625-0572 [fax] carlis@umn-cs.cs.umn.edu

Letter from the Issue Editor

This special issue is devoted to describing *some* of the activities underway on the topic of transaction models in applications that go beyond traditional banking scenarios. Most of these new transaction models stem from a practical need to relax one or more of the ACIDity (Atomicity, Consistency, Isolation and Durability) properties of a transaction. These papers are, for the most part, written in the context of a specific system. The systems covered in these papers vary, from the heterogeneous database, computer publishing, computer aided software development, computer aided design, active databases etc. Many of these papers address long transactions such as those found in CASE and CAD systems. The variety of systems are as interesting as the differences in approaches. I hope that one of the conclusions to be drawn from this particular special issue is that transactions should be perceived as an enabling technology that can be used to build advanced applications that are flexible. This flexibility manifests itself in the user's ability to select the degree to which he wants certain properties enforced.

The papers are presented in alphabetical order of the first author's name. The first paper by Dayal, Hsu and Ladin presents an extended transaction model for long lived transactions and active databases. The paper by Ellis differs from most other papers in that it only briefly discusses the collaborative nature of groupware. Instead, it spends more time in discussing the approach to concurrency control used in groupware. The paper by Garcia-Molina, Gawlick, Klein, Kleissner and Salem describes nested sagas. Nested sagas allow for composition of long-running activities into sagas, thereby, gaining the ability to abort or commit activities independently. The paper by Kaiser and Perry addresses transaction models for cooperative environments. The authors' experience in software development environments has set the tone for the extensions they discussed. The paper by Lee, Mansfield and Sheth addresses requirements for transaction processing in a multimedia telecommunications environment. The paper uses a feedback mechanism to make the model interactive. This provides the ability to build applications as a set of cooperative tasks. The paper by Leu presents a transaction model for multidatabase systems. By allowing more than one acceptable execution path for a single transaction and typed subtransactions, this model relaxes both the atomicity and isolation properties in a way that facilitates transaction processing in multidatabase systems. The paper by Muth, Rakow, Klas and Neuhold is written for a fairly unusual and relevant application. The paper addresses the requirements of a transaction model for publication environments. More specifically, it presents an open distributed publication environment for multimedia products. The paper by Reuter and Wachter on the contract model, addresses the limitations of classical models and deals with various mechanisms for managing activities based on the contract model. This model uses a script to describe the activities in a contract. The paper on polytransactions by Rusinkiewicz and Sheth describes a multidatabase environment in which data is interrelated in various ways. After briefly describing interdependent data, the paper introduces the notion of polytransactions to deal with them. The paper by Unland and Schlageter is probably more related to the management of transactions than any of the other papers. The adaptable tool kit approach is described and then various strategies for performing concurrency control and recovery are presented based on this approach. The paper on s-transactions by Veijalainen and Eliassen discusses how the s-transaction model is used to preserve both local and global consistency in a highly autonomous multidatabase environment. Finally, the paper by Weikum and Schek gives an overview of the multi-level transaction model and its generalization for open nested transactions. They also discuss the various requirements of ACIDity and provide various potential applications.

I want to thank all the authors for putting up with my nit picking. I know some of them have gotten pushed to the limit at times. I believe that we are all better for this effort of writing and rewriting. I would also like to thank Mr. Yungho Leu for helping out at various stages of this process. Since the papers had to be limited to 5 pages for this special issue, we have decided to publish extended versions of a subset of these papers, and a few other papers that did not make it into this issue, in a book to be tentatively published by Morgan Kauffmann later in 1991.

Before closing this letter, I would like to introduce to the readers of the Bulletin the newly established Indiana Center for Database Systems. The center is a state wide effort by Purdue University, Indiana University and various other institutions in the state. The primary objectives of this center are the establishment of research, technology transfer and outreach programs to benefit the database industry. The Director of the center is Judith Copler and the Executive Directors are myself and Edward Robertson of Indiana University. The center was established through a generous grant from the Indiana Corporation for Science and Technology.

Ahmed K. Elmagarmid
Associate Professor and Executive Director
The Indiana Center for Database Systems
Department of Computer Sciences
Purdue University

A Generalized Transaction Model for Long-Running Activities and Active Databases

Umeshwar Dayal[*]
Meichun Hsu[†]
Rivka Ladin[‡]

Digital Equipment Corporation

1 Introduction

In the conventional transaction model supported by existing database management systems, a *transaction* is the atomic unit of work. A transaction is guaranteed to satisfy the concurrency atomicity, failure atomicity, and permanence properties.

This model is limited for many applications. First, it presupposes strictly sequential transactions. The nested transaction model [Lisk85, Moss81] overcomes this limitation by allowing a transaction to spawn *subtransactions* that execute concurrently. In the nested transaction model, the subtransactions are *immediate* in that they can be scheduled for execution as soon as they are spawned.

Sometimes, however, it is necessary to *defer* the execution of some actions to the end of a transaction. For example, the deferred actions might check integrity constraints, propagate updates performed during the transaction to replicas of the updated objects, propagate updates to derived data (e.g., to materialized views), or execute "automation rules" for postprocessing [HC88]. The execution semantics of deferred actions are not described by the nested transaction model.

A more serious limitation is that both the conventional transaction model and the nested transaction model presuppose short, isolated transactions. Sometimes, it is necessary to break off some actions of a transaction and to execute these actions in one or more separate, *decoupled* transactions. For example, in an inventory control application, transactions may update the quantity on hand of some item in the inventory database (to reflect the consumption of the item). If the quantity of hand falls below a threshold, then the item has to be reordered. However, there is no need to execute the reordering action as part of the original transaction that consumed the item. Decoupling some actions permits transactions to finish more quickly, thereby releasing system resources earlier, and improving transaction response times. A decoupled transaction can execute concurrently with the transaction from which it was spawned. Often, we want the decoupled transaction to be *causally dependent*: it must be serialized after the transaction from which it was spawned, and it can commit only if the latter commits. Sometimes, however, *causally independent decoupled* transactions are desirable. For example, suppose we want to write a record in the security log whenever a user accesses some data object; we want the security log record to be written irrespective of whether the original transaction that accessed the object commits or aborts. To do this, we write the security log in a causally independent transaction.

In [HLM88, Chak89, DHL90], we introduced a generalized transaction model, and accompanying language primitives, that supports different kinds of *nested transactions*: concurrent subtransactions, deferred subtransactions, and decoupled (causally dependent and independent) transactions. We were motivated by real application needs, rather than some theoretical notion of completeness. In particular, we were motivated by the need to describe the semantics of active databases and long running activities. (In contrast, the work on generalized

[*]Address: Digital Equipment Corp., One Kendall Square – Building 700, Cambridge, MA 02139; dayal@crl.dec.com.
[†]Address: Digital Equipment Corporation, Mountain View, CA 94040; hsu@ocean.dec.com
[‡]Address: Digital Equipment Corp., One Kendall Square – Building 700, Cambridge, MA 02139; rivka@crl.dec.com.

transaction frameworks, such as ACTA [CR90], is motivated by the desire to describe and compare all existing transaction models.)

In this short paper, we first give in Section 2 a brief overview of our generalized transaction model. Then in Section 3, we illustrate its use as the execution model for active databases. In Section 4, we show how to use the model to express control flow in long-running activities. We augment the model to support the recovery semantics (and other desired features) of activities. For details, the reader is referred to [HLM88, DHL90, DHL9x]. For details of the algorithms for implementing the model, see [Chak89, DHL9x].

2 The Transaction Model

Our model gives the programmer fine control over the scope in which a transaction is executed. To achieve this, we extend the nested transaction model described in [Moss81, Lisk85]. A *nested transaction* is a transaction that is started from inside another transaction (the *parent transaction*). Transactions can be nested to arbitrary levels, forming a tree with a *top transaction* at the root. We first describe briefly the basic nested transaction model. Then, we present our extensions, which are twofold: the first allows tasks to be explicitly deferred to the end of the transaction; and the second permits the decoupling of tasks to be performed in a separate transaction.

2.1 The Nested Transaction Model

A transaction may contain any number of nested transactions or *subtransactions*, some of which may be required to perform sequentially, some concurrently. We use standard tree terminology in referring to relationships between transactions, for example, *parent*, *child*, *ancestor* and *descendant.* A subtransaction may be aborted without causing its parent transaction to abort.

Concurrency within a transaction is obtained by allowing the parent to start concurrent subtransactions. While a child is running, its parent is suspended. However, sibling subtransactions may execute concurrently. Because siblings are serializable at each level of the transaction tree, there is no problem with concurrent siblings interfering with one another. Sequential siblings are ordered according to when they run. This structure can't be observed from the outside; i.e., the overall transaction still satisfies the atomicity properties.

The commit of a subtransaction is always relative to its parent. If a subtransaction commits and its parent aborts, the effects of the subtransaction will be undone. When a subtransaction T and all its ancestors up to, but not including, the top transaction commit, we say that T has *committed to the top.* When T's top transaction then commits we say that T has *committed through the top.* The top transaction commits only after all of its subtransactions have terminated.

A top transaction and its descendants can be modelled by means of a tree structure called a *transaction tree.* The root of the tree is labelled by the top transaction; the interior nodes are labelled by the descendant subtransactions. For convenience, we assume that there exists a distinguished system transaction, T_{sys}; every top transaction is a child of T_{sys}.

To constrain the execution order of concurrent siblings, priorities can be assigned to transactions. The system guarantees that the serialization order of concurrent siblings is consistent with their priority order.

2.2 Deferred and Decoupled Transactions

In addition to the nesting of subtransactions described above, we allow three more types of nesting. First, we allow the creation of *deferred subtransactions* whose execution is explicitly delayed until the end of the user's top transaction T and before any deferred subtransaction is executed, a point we shall refer to as the *cycle-0 end.* When T reaches its cycle-0 end, a deferred subtransaction is started, and runs as a proper subtransaction of T. If more than one deferred subtransaction is created before T reaches its cycle-0 end, then all these subtransactions are started as concurrent subtransactions in *cycle 1* at cycle-0 end. If the processing of subtransactions in cycle 1 causes more deferred transactions to be created, the latter are started when all subtransactions in cycle 1 have finished, and are started as concurrent subtransactions of T in *cycle 2.* The cycles of execution of T continue until the last cycle finishes in which no more deferred subtransactions are created. Like a regular subtransaction, the commit of a deferred subtransaction is conditional on its parent (the transaction that created it) committing through the top.

Second, we allow a separate top transactions to be started from inside another transaction. Such a "nested top transaction" is called a *decoupled transaction.* A decoupled top transaction will be represented by its own tree. We identify two kinds of decoupled transactions based on whether they are causally dependent on their parent or not. Let T be the top transaction and let T' be a *decoupled transaction* created either by T or by one of its descendants. Then T' is *causally dependent* on transaction T (we say that T' is a *CDTop transaction*) iff T' is serialized after T and the commit of T' is conditional on the committing of its parent through the top. Note however, that aborting of T' has no effect on its parent. It is important to note that CDtop transactions whose natural parents have committed must be scheduled for execution. Therefore, CDtop transactions that are interrupted by a system failure should be automatically restarted as part of system recovery.

The execution of a *causally independent decoupled transaction* T' has no special privileges relative to its parent T; the commit of T' is not relative to its parent, but rather independent. Note that we don't constrain the serialization order of T' relative to its parent.

In the standard nested transaction model, a subtransaction cannot control the fate of the top level transaction. We extend these failure semantics to allow a subtransaction to request that its top transaction and all its decendants (excluding the causally independent decoupled top transactions) be aborted.

To constrain possible execution orders of concurrent CDtop transactions, we support a *pipelining* mechanism. We say that a decoupled transaction T' created by transaction T satisfies the pipelining property if for all transactions Ti that are serialized before (after) T, any decoupled transaction Ti' created by Ti is serialized before (respectively, after) T'. Thus, suppose a decoupled transaction is used to display a moving target's position on a screen every time the position is updated. If many update transactions occur in a short period, several decoupled display actions may be queued. For the display to reflect the correct sequence of updates, the display actions must be pipelined.

3 Execution Model for Active Databases

In this section, we illustrate the use of the transaction model to describe the execution semantics of active databases. An *active database* contains both data and rules [Daya88b, DBM88]. A rule is an event-action pair. The event may be a database operation, a temporal event, an external signal, or combinations of these; the action is any program.

A transaction may trigger the execution of a rule's action by causing its event to occur. In most rule models (e.g., [Ston86, Syba87, KDM88, WF90]) the triggered actions execute within the triggering transaction, either immediately (when the triggering event occurs) or they may be deferred (to the end of the transaction). This prolongs the original transaction, especially if rule executions are allowed to cascade, causing locks to be held for a long time and thereby limiting database concurrency. Also, most rule models support only sequential execution of rules.

In our model, a rule includes the specification of a *coupling mode* — immediate, deferred, causally dependent decoupled, or causally independent decoupled. The coupling mode specifies the transaction scope within which the action is executed relative to the triggering transaction (i.e., the transaction that caused the event to occur). When the event is detected, the system creates an appropriate (nested) transaction to execute the action part. If several rules have the same triggering event and the same coupling mode, they are executed concurrently. Priorities, and the cycling and pipelining mechanisms, may be used to restrict concurrent execution. Also, the execution of one rule may raise events that cause other rules to be triggered.

In the inventory control example, we can write a rule whose event is the update of the quantity on hand of an item, and whose action invokes the reorder procedure if the quantity on hand has dropped below the threshold. The desired semantics are obtained by executing the action part of the rule in a decoupled transaction.

In addition to monitoring events and starting nested transactions, the system may also need to recover events that were signaled by committed transactions that spawned uncommitted nested CDtop and top transactions. Therefore, a transaction commits only after its database updates, *and* the events signaled by it, are stably captured. With these signals recovered, the system can restart the interrupted decoupled transactions and ensure the completion of the execution. We distinguish between *recoverable* and *irrecoverable* events; all events triggered by database updates are recoverable events. Temporal events, on the other hand, may be recoverable or not. Upon recovery, events signalled by commited transactions and for which the necessary action was taken before the failure, are recovered independent on whether they are recoverable events or not. Events signalled by commited transactions, and whose processing has not completed before the failure, are signalled only if they are recoverable events.

4 Organizing Long-Running Activities

A *long running activity* involves multiple steps of processing (which may be serviced by different servers, perhaps on different nodes of a distributed system) and that typically are of long duration. For example, a purchase order may be issued from an inventory clerk, then passed to a manager who approves it, and then passed to an accountant who makes proper accounting entries. Executing a long-running activity as a single transaction is not strictly necessary in most cases, and can significantly delay the execution of short transactions. For example, if purchase order processing is run as a single transaction, locks on the inventory records and the budget records may be held for a long time, severely limiting database concurrency. When these steps involve several distributed servers, commit processing is also expensive, and the transaction can run only when all servers are available simultaneously.

One approach to handling a long-running activity, therefore, is to have each step run as a transaction; thus, the long-running activity corresponds to multiple transactions. In conventional transaction processing systems, the control flow among the steps is embedded in application programs (e.g. [McGe78]). However, there is no system support for handling failures or exceptions across the steps of the long-running activity.

Several extended transaction models to support long-running activities have been proposed [GS87, KR88, Reut89, Garc90, ELMA90]. Each step is executed as a transaction. Control flow among steps is declaratively specified. These models also provide an automatic *compensation* capability that offers failure atomicity for the user request.

To govern the execution of multiple application steps that are related, we augment our transaction model with a control stucture called an *activity*.

A top transaction is created from within an activity. Activities can be further nested. Thus, children of an activity may be activities or top transactions or a combination of these. An activity also has three states: active, committed and aborted. The relationship between an activity and its children is similar to that between a parent transaction and its children. A parent activity is committed only after all its children have terminated. However, there are a couple of differences between activities and transactions. If a parent activity is aborted, then all its *active* children are aborted; committed children are not affected. Furthermore, sibling activities are *not* serializable; their effects on the database may be interleaved.

It is desirable to allow a user to query the status of an activity, or to stop or alter the progress of an activity. For this purpose, a program that creates an activity is given a *handle* for the activity. After an activity is created, the program may *query* the status of the activity by presenting the activity handle to the system.

The program may also ask the system to *abort* the activity. Aborting an activity is defined as follows. All children activities are aborted; all active top transactions are aborted. Committed top transactions are *not* aborted, but may be compensated for (see [DHL9x] for more information).

The control flow among the steps of an activity is expressed in the application program, or is implicit in rules as described in [DHL90]. Thus, a step S1 that must start another step S2 can do so by starting a nested transaction to execute S2, or it can signal an event that triggers a rule whose action part executes S2. The use of rules allows the control flow to be dynamically modified based on the database state or the history of events that have occurred. Exception handlers (and compensation actions) can be associated with each activity or step as desired; these are invoked auomatically by the system using a fixed policy as in the saga model; or, they can be dynamically invoked by rules.

References

[Chak89] Chakravarthy, S., et al., "HiPAC: A Research Project in Active Time-Constrained Database Management. Final Technical Report." Xerox Advanced Information Technology, Cambridge, Mass., July 1989.

[CR90] Chrysanthis, P.K., K. Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior." *Proc. ACM SIGMOD Conf.*, May 1990.

[Daya88b] Dayal, U., "Active Database Systems." *Proc. 3rd International Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

[DBM88] Dayal, U., A. Buchmann, D. McCarthy, "Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System", *Proc. 2nd International Workshop on Object-Oriented Database Systems*, West Germany, September 1988.

[DHL90] Dayal, U., M. Hsu, R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions." *Proc. ACM SIGMOD Conf.*, May 1990.

[DHL9x] Dayal, U., M. Hsu, R. Ladin, "A Transactional Model for Activities." (in progress).

[ELMA90] Elmagarmid, A.K., Y. Leu, W. Litwin, M. Rusinkiewicz, "A Mulidatabase Transaction Model for InterBase." *Proc. VLDB*, August 1990.

[Garc90] Garcia-Molina, H., et al., "Coordinating Multi-Transaction Activities." Report UMIACS-TR-90-24, CS-TR-2412, Computer Science Technical Report Series, University of Maryland, College Park, MD.

[GS87] Garcia-Molina, H. and K. Salem, "Sagas," *Proc. ACM SIGMOD Conf.*, May 1987.

[HC88] Hsu, M. and T.E. Cheatham, "Rule Execution in CPLEX", *Proc. 2nd International Workshop on Object Oriented Database Systems*, West Germany, September 1988.

[HLM88] Hsu, M., R. Ladin, and D. McCarthy, "An Execution Model for Active Database Management System," *Proc. 3rd International Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

[KDM88] Kotz, A.M., K.R. Dittrich, and J.A. Mueller, "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism," *Proc. Conf. on Extending Data Base Technology*, Venice, 1988.

[KR88] Klein, J. and A. Reuter, "Migrating Transactions," *Future Trends in Distributed Computer Systems in the '90s*, Hong Kong, 1988.

[Lisk85] B. H. Liskov. "The Argus Language and System." *Distributed Systems: Methods and Tools for Specification.* pp. 343-430. Springer-Verlag, Berlin 1985.

[McGe77] McGee, W.C., "The Information Management System IMS/VS Part V: Transaction Processing Facilities," *IBM Sys. Journal*, Vol. 16, No 2., 1977, pp. 148-169.

[Moss81] J. Moss. "Nested Transactions: An Approach To Reliable Distributed Computing." MIT Laboratory for Computer Science, MIT/LCS/TR-260 1981.

[Reut89] Reuter, A., "Contracts: A Means for Extending Control Beyond Transaction Boundaries," Presentation at 3rd Workshop on High Performance Transaction Systems, Pacific Grove, CA, September 1989

[Ston86] Stonebraker, M. et. al. "A Rule Manager For Relational Database Systems." *The POSTGRES Papers.* Univ. of California, Berkeley, Electronics Research Lab, Memo No. UCB/ERL M86/85, 1986.

[Syba87] Sybase, Inc. *Transact-SQL User's Guide.* 1987.

[WF90] Widom, J., and S.J. Finkelstein, "Set-Oriented Production Rules in Relational Database Systems." *ACM SIGMOD Conf.*, May 1990.

Consistency within Concurrent Groupware Systems*

by

Clarence A. Ellis
MCC Software Technology Program
Austin, Texas, USA

Abstract

Groupware systems are computer based systems which supports two or more users working in a tightly coupled fashion on a common task. This paper briefly introducess a family of concurrency control algorithms for groupware systems. These algorithms maintain consistency without locking, and without rollback, within a dynamic non-serializable environment. A consistency theorem for this family of algorithms has been devised, and its proof is sketched in a companion document.

1. Introduction

Groupware aims to assist groups in communicating, in collaborating, and in coordinating their activities. Groupware can be defined as computer based systems that support two or more users engaged in a common task or goal, and that provide an interface to a shared environment. The groupware group at MCC's software technology program has been researching groupware and computer supported cooperative work for the past five years. A number of prototypes have been produced, measurement and modeling of those prototypes has occurred, and lessons learned have lead to various theories and models of the resultant processes and systems.

These systems can be categorized as real-time groupware versus non-real-time groupware. Examples of real-time groupware are multi-player video games, real time group editors, video conferencing systems, and group decision support (electronic meeting room) systems. Examples of non-real-time groupware are office coordination systems, intelligent electronic mail, and software engineering project managers. See [Elli90b] for further motivation, examples, issues, and references. Groupware systems differ dramatically from general database management systems and other multi-user systems because they are built to explicitly allow users to know and easily keep track of the presence of others.

* *Many of the notions mentioned in this extended abstract are further elaborated in the proceedings of the ACM SIGMOD'89 International Conference on Management of Data.*

Real-time groupware is characterized by the following:

- highly interactive - response times must be short.
- real-time - notification times must be comparable to response times.
- distributed - in general, one cannot assume that participants are all connected to the same machine or even to the same local area network.
- volatile - participants are free to come and go during a session.
- ad hoc - generally the participants are not following a pre-planned script, it is not possible to tell a priori what information will be accessed.
- focused - during a real-time work session there is a high degree of synergistic shared data access, and an unusually high probability of unwanted access conflicts.
- external channels - often participants are connected by one or more external (to the computer system) channels such as an audio or video link. We used speaker phones within our offices for many of our groupware sessions.

Examples of advanced real-time groupware include GROVE (Group Outline Viewing Editor) [Elli88], and rIBIS (real-time hypermedia system) [Rein90]. These were fully distributed systems implemented in the Software Technology Program at MCC. They were specifically designed for real-time use by groups of people performing simultaneous editing.

2. Concurrency Control Problem

Concurrency control is needed within real-time groupware to help resolve information access conflicts between participants, and to allow them to perform tightly coupled group activities. For example, with a group editor, clearly there is a conflict if one participant deletes a sentence while a second inserts a word into the sentence. In the usage observations of GROVE, we have noticed that there is a mode of operation in which a tightly coupled group will do a complex sequence of edit operations in a concurrent fashion, getting the task performed in a much more efficient manner. Many CASE tools (computer aided software engineering) discourage rather than enhance closely coupled teamwork. There is a need for mechanisms which go beyond today's typical technology. The various approaches to providing concurrency control, such as explicit locking or transaction processing, that have been developed for database applications do not appear to be suitable in groupware contexts. Interactive concurrency control techniques are much more useful in this context. This section identifies some of the issues related to concurrency control in groupware, and overviews our approach.

2.1 Issues

WYSIWIS. Although there has been little experience in the evaluation of interfaces to groupware [Grud88, Elli89] it appears that some form of a WYSIWIS (what you see is what I see) interface [Stef87] is very useful to maintain group focus. If each user sees a slightly different or out-of-date version then the session's cohesiveness is soon lost. WYSIWIS interfaces have two implications on concurrency control. First response times are important - the time taken to access data, modify data, or notify users of changes must be as short as possible. Secondly, if the concurrency control scheme entails the use of modes where actions of one user are not immediately seen by the others, then the effect of these modes on the group's dynamics must be carefully analyzed and only allowed if they are not disruptive.

Wide-area distribution. One of the main benefits of groupware is that it allows people to work together, in real-time, even though separated by great physical distances. Consequently these systems may be geographically distributed. With current communications technology, transmission times and rates for wide-area networks are significantly worse than those found in their local area counterparts; the possible impact on response time must be taken into account.

Replication. Because the response time demands of groupware are so high, the data state is usually replicated for each participant. This allows many potentially expensive operations to be done locally. For instance, consider an editing session where one participant is in Los Angeles and the other in New York. Typically each participant would be working in a windowing environment. If the objects being edited and the data state are not replicated then even simple scrolling operations require communication between the two sites. The resulting degradation in response time may be catastrophic.

Robustness. Traditionally robustness refers to recovery from unusual circumstances, typically these are component failures - the crash of a site or a communications link. While these are also concerns within groupware, there is also a second form of robustness these system must achieve, in particular, robustness to user actions. For example, the addition of a new user to the set of users issuing transactions on a database is not normally considered a major problem. However, with groupware, the addition of a participant may result in what amounts to a reconfiguration of the system. Clearly the concurrency control algorithm must adapt to such reconfigurations and in general recover from "unexpected" user actions (abruptly leaving the session, going away for coffee, etc.)

2.2 Our Approach

At MCC, we have explored notions of *soft locks [Elli87]*, and *interactive concurrency control [Yeh89]*. In our recent groupware systems, we have employed the dOPT algorithm which, when combined with the above techniques, provides a powerful new concurrency control mechanism for groupware. dOPT abbreviates *distributed operation transformation algorithm*, and proceeds without locking or roll-back. This approach relies upon application specific semantic knowledge of the desired outcome of concurrent operations. For example, when two participants make concurrent edits to the same data structure, their local copies are updated immediately, and messages containing the edit operation and carefully selected local state are sent to all other sites. When each of these sites receives the other's message, they know if they are performing the pair of edit operations in different orders. Each first performs an application dependent transformation on the operation, and each applies the transformed operation to their local copy of the data structure. Voila! It can be shown that for a significant class of applications, all is guaranteed to end up consistent [Elli91].

A groupware system execution is defined to be *correct* if it guarantees that an initially consistent system will, at the conclusion of any admissible execution (called quiescence), still maintain consistency, semantic integrity, and temporal ordering.

8. Conclusions and Future Developments

This paper has introduced the notion of groupware, and presented a novel algorithm for concurrency control within real-time groupware. Groupware reflects a change in emphasis from using the computer to solve problems to using the computer to facilitate human interaction. For these systems to be accepted requires fine-granularity sharing of data, rapid response time, and rapid notification time. Therefore, the algorithm presented does not use locking, performs non-serializable sets of operations, and works in a workstation environment with replicated data and distributed control.

A general theorem has been developed which allowed us to prove consistency of the replicated object set by examining properties of the combined precedence graph. This consistency property, together with other properties of temporal ordering and semantic integrity, fulfill our criterion of correctness. This is done elsewhere [Elli91].

The algorithm and the proof were worked out in the context of GROVE and rIBIS, group editors which we have implemented within the Software Technology Program at MCC. In these systems, users frequently apply associative access techniques rather than accessing objects by name. Thus, within a text editing application, users browse and point at the items they want to update; the individual characters are not given separate immutable names or unique addresses.

A primary difference between these systems, and the majority of database systems, is the visibility criterion. DBMS's are constructed with an intent to hide the presence of other users (transactions, locking, etc.); groupware is constructed with the intent of making visible the presence and state of other participants. Future work of our research team includes embellishing the group user interface to better make this happen. There is also work in progress to generalize the characterization of our transform matrix, and extending our proof technique to encompass other systems. We will also continue to incorporate these ideas within other groupware which we are constructing, and will be constructing in the future. One challenging issue is the implementation of the UNDO function because there is only a partial ordering on previous operations. We also have some ideas for the implementation of further DWWM features. We have been developing and applying the notions of *team automata* to model and prove various other properties of groupware. We are also exploring alternative models. We hope that this will be useful in further correctness proof extensions. In conclusion, we believe that the new emphasis on groupware suggests a number of interesting and challenging frontiers. Perhaps this paper can help to stimulate work at these frontiers.

Acknowledgments

Special thanks to Simon Gibbs, co-author of our previous SIGMOD paper on this subject, and chief designer / programmer of GROVE. His creative ideas and programming have been critical to the success of this project. Thanks to Gail Rein, the other member of the groupware team who put in much effort to help implement and evaluate GROVE and rIBIS, and their novel user interfaces.

References

[CSCW86, CSCW88, and CSCW90] *Conferences on Computer-Supported Cooperative Work*; Proceedings editor: ACM, New York, N.Y.

[Elli87] Ellis, C.A., and Ege, A. Design and Implementation of Gordion, an Object Base Management System, *Proceedings of the International Conference on Data Engineering*, 1987.

[Elli88] Ellis, C.A., Gibbs, S. J., and Rein, G. Design and Use of a Group Editor, *Report STP-263-88, MCC Software Technology Program,* Austin, Texas, 1988.

[Elli90] Ellis, C.A., Gibbs, S.J., and Rein, G. Groupware: Some Issues and Experiences, *Communications of the ACM,* 34(1), January 1991, also available as MCC Software Technology Program technical report.

[Elli91] Ellis, C.A. A Model and Algorithm for Concurrent Access within Groupware, accepted for publication in the IEEE Journal of Office Knowledge Engineering, 1991.

[Grie86] Grief, I., Seliger, R., and Weihl, W. Atomic Data Abstractions in Distributed Collaborative Editing System, *Proceedings of the 13th Annual Symposium on Principles of Programming Languages,* pp. 160-172, 1986.

[Grud88] Grudin, J. Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces, *Proceedings of the CSCW'88,* 1988.

[Rein90] Rein, G., and Ellis, C. rIBIS: A Real-time Group Hypertext System, *Report STP-095-90, MCC Software Technology Program,* Austin, Texas, 1990.

[Sari85] Sarin, S. and Grief, I. Computer-Based Real-Time Conferences, *Computer,* 18(10), pp. 33-45, 1985.

[Stef87] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., and Suchman, L. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings, *CACM 30(1),* pp. 32-47,1987.

[Yeh89] Yeh, S., Ellis, C., Ege, A., and Korth, H. Performance Analysis of Two Concurrency Control Schemes for Design Environments, *Information Sciences Journal 49(1),* 1989.

ABOUT THE AUTHOR:

Clarence A. Ellis is a senior member of the technical staff in the Software Technology Program at the Microelectronics and Computer Technology Corporation (MCC), and an adjunct professor at the University of Texas. He has a Ph.D. in computer science with research interests in collaboration technology, office information systems, databases, distributed systems, software engineering, and humane interfaces to computers. Clarence (Skip) Ellis has previously worked at Xerox PARC, IBM, Bell Telephone Labs, and Los Alamos and Argonne National Labs. His academic experience includes teaching at Stanford University, MIT, University of Colorado, Chiaotung University, and University of Texas. He is a member of the National Science Foundation advisory committee, and chairman of the ACM Special Interest Group on Office Information Systems.

Modeling Long–Running Activities as Nested Sagas

Hector Garcia-Molina

Dept. of Computer Science, Princeton University, Princeton, NJ 08544

Dieter Gawlick, Johannes Klein, Karl Kleissner

Digital Equipment Corporation[†], Mountain View, CA 94040

Kenneth Salem

Dept. of Computer Science, University of Maryland, College Park, MD 20742

ABSTRACT

Long–running activities often consist of collections of related, simpler steps. We propose the use of nested sagas to model such activities. This model allows useful operations, such as aborts, to be performed on the activities, without requiring the high cost of long–lived transactions.

1. Introduction

Consider the process of registering an automobile with a state's department of motor vehicles. A vehicle registration is initiated by a request from the vehicle's owner. In the request, the vehicle's owner supplies relevant information about it, such as its make, model, age, and identification number. To perform the registration, a number of steps must be completed. For example, the department may require that the vehicle pass a safety and emissions inspection and may check its databases for any information about the vehicle, such as existing registrations. Next, a new, unique registration number must be assigned to the vehicle and a registration card and license plates must be generated for the car. Finally, the department collects a registration fee from the owner.

Many business activities can be modeled as collections of related sub–activities. Figure 1 illustrates this for the vehicle registration activity. This simple example illustrates that activities may be composed, or nested. In the example, the "inspection" sub–activity actually involves separate safety and emissions test activities, as well as the payment of an additional inspection fee.

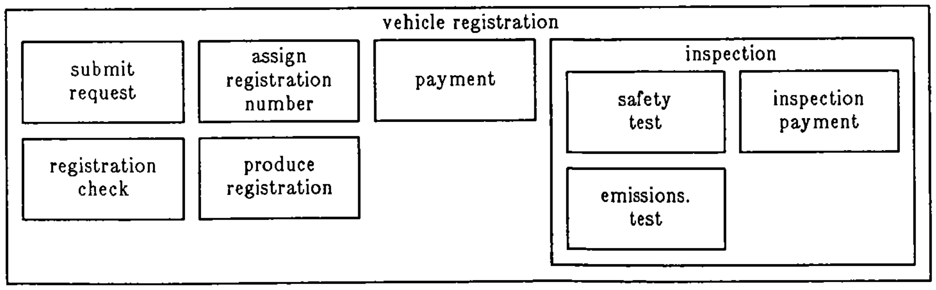


Figure 1 – Vehicle Registration Activity

Activities such as vehicle registration have a number of properties that suggest that transaction processing techniques could beneficially be applied to them. It is likely that concurrent activities will require concurrent access to shared databases, e.g., a database of license plate

† This paper expresses the views of the authors only. It does not express the opinions or future product plans of Digital Equipment Corporation.

numbers and their assignments. Furthermore, the effects of completed activities should not be lost as a result of system failures. The department should not "forget" that a vehicle has been registered once the registration process has completed. However, despite the existance of these properties, simple transaction models are inappropriate for such activities. One problem is that activities may be long-lived. Hours or days may elapse between the initiation of the registration activity and its completion, even if the individual steps are not themselves long-lived. For this reason, it quickly becomes impractical to treat activities as atomic transactions. Implementing this would require that resources (e.g., databases) accessed by the activity be held (e.g., locked) for long periods of time. Alternatively, the individual sub-activities could be treated as separate transactions. However, no services are provided for the entire activity in this case. In particular, it is not possible to "abort" or "commit" the activity as a whole, since it is not itself a transaction.

In this paper, we propose that activities be modeled as generalized *sagas* [Garc87,Giff85], rather than as transactions. A saga is a collection of atomic transactions, though the saga itself is not atomic. Thus, resources are released after each of the component transactions completes, allowing sagas to be long-lived. Although sagas are not atomic, they can be aborted. In general, the effects of a saga cannot be rolled back, as a transaction's can, because the affected resources may already have been released. Instead, a saga is "aborted" by executing *compensating transactions* for each transaction in the saga that has already commited and released its resources. A compensating transaction "undoes", according to the semantics of the application, the effects of the transaction it compensates for.

In the following sections, we present sagas in more detail and discuss some of the issues that arise when sagas are nested. We also discuss a simple system call interface that could be used by application programs (e.g., the vehicle registration code) to request services from a "saga processing system".

2. Sagas

Originally [Garc87], sagas were defined as collections of atomic transactions. By grouping transactions into a saga, an application gained the ability to abort the saga. A saga abort is processed according to two rules:

1) Active (uncommitted) transactions in the saga are aborted and rolled back.

2) A *compensating* transaction is initiated for each committed transaction in the saga.

Like the regular (forward) transactions that comprise the saga, compensating transactions are application programs that must be coded and supplied when the saga is created.

Applications similar to motor vehicle registration require a more general saga model. Since sub-activities may themselves be long-lived and involve a number of steps, modeling them as atomic transactions may be inappropriate. A natural solution to this problem is to allow sagas to be composed of transactions or of other sagas, i.e., to allow the *nesting* of sagas. Nested sagas can be defined recursively, as follows:

- A single, atomic transaction is a *primitive* saga.
- A collection of sagas is a *composite* saga.

The state of a saga is one of committed, aborted, or running. A primitive saga has the same state as the transaction that it is composed of. A composite saga that is not aborted is committed if all of it component sagas have committed or aborted. Otherwise it is running.

A saga can be requested to abort at any time. Primitive sagas are aborted by rolling back their effects, since they are atomic transactions. An abort of a composite saga is applied to each of its component sagas. Those that have committed are compensated for, while those that are running are recursively aborted. The compensating actions used to abort a saga must be specified when a saga is defined.

We can illustrate these ideas using the motor vehicle department example (Figure 1). Consider the situation in which a registration saga is running, and assume that the "submit request", "registration check" and "inspection" components of the registration have committed, and that the two remaining components are active (running). If the "registration" saga is aborted at this point, the following actions occur. First, the two active components ("assign registration number" and "produce registration") are aborted. Since these a primitive sagas they can be rolled back. Next, compensating actions are initiated for each of the three committed components of "registration". For example, the compensating actions for the "submit request" and "inspection" requests may simply mark the registration request and inspection records for this registration invalid in the department's databases, or they might delete the records. Some activities, such as "registration check", may not require any compensation. For such activities, a null compensation step can be specified.

If the "inspection" saga is not committed when "registration" is aborted, the abort procedure is somewhat different. Since the "inspection" saga is not committed, its compensation step is not executed. Instead, the "inspection" step is recursively aborted. Compensation steps would be initiated for any of the three inspection sub-activities that had already committed. Uncommitted sub-activities would be rolled back.

In summary, sagas provide the capability to abort long-running activities without incurring the costs of long-lived transactions. The nesting of sagas permits modular composition of activities. Since both sagas and transactions (primitive sagas) can commit and abort, nested sagas can be built up from smaller components, without regard for their nature, i.e., whether they are transactions or sagas.

3. An Environment for Activities

We have addressed the semantics of sagas and nested sagas, defining what it means to combine small sagas into larger ones. However, we have not considered a *mechanism* for sagas. In this section, we address the question of how an application might create, commit and abort sagas.

There are many ways to answer this question. The view we take here is that a "saga processing system" exists which implements sagas. The saga processing system has a simple interface, i.e., a set of system calls, which application programs can use to request saga services. A saga processing system is analogous to an operating system, which implements processes, or to a transaction processing system, which implements transactions.

In the remainder of this section, we will describe a simple interface to a saga processing system and show how it can be used to acquire saga services. We do not suggest that using the system call interface described here will, in practice, be the best way to "code up" an application. Graphical tools, additional system calls, and other enhancements would certainly make this process easier. However, our goal is to present a very simple set of primitives that provide the necessary services.

Primitive sagas, or *steps*, are simply executions of application programs. Application programs can be written in any language that has a small set of system calls (described below) embedded within it. Steps use these calls to request system services. The system calls at the interface include *Create*, *Commit*, *Abort*, and *Compensation_Bind*.

3.1 Create

The *Create* system call is used to create new steps. *Create* is invoked with an argument describing the program the new step should execute. (Several additional, optional parameters exist, whose purposes are described in Section 3.3, below.) Every step is executed as an atomic transaction, i.e., a primitive saga. Furthermore, when a step creates new steps, the new steps together with their creator constitute a saga. Thus, every newly-created step is a part of two sagas: a composite saga that includes its parent, and its own primitive saga. When a step creates new steps, its primitive saga "expands" to include the new offspring.

The *Create* system call is a useful tool for modular composition of activities because every created step can be considered a saga. It allows a step to create a child step to perform a sub-activity without worrying about whether the child will perform the activity itself, or create additional steps to do so. In our motor vehicle example, the "inspection" activity could be implemented by a program called "inspection". To perform the inspection, the "vehicle registration" program would *Create* a step running the "inspection" program. This program could implement the inspection activity itself, or it could, in turn, create new steps ("safety inspection", "emission inspection", and "payment") to do the necessary work. In either case, "inspection" appears to "vehicle registration" as a saga that accomplishes the necessary task.

3.2 Commit

The *Commit* system call allows a running step to commit itself. Committing a step makes that step's effects (e.g., database modifications) permanent. If the step created new steps during its execution, then the step's saga does not commit until all of these steps stop running. For example, if the "inspection" step simply creates three steps and then commits, the "inspection" saga is not considered committed because it includes the three created steps.

3.3 Abort

Any saga can be aborted. Aborting a primitive saga (i.e., a step) undoes any changes made by the step. An abort of a composite saga is handled recursively as described in Section 2. By default, the abort of a saga also causes the saga's parent saga to abort, i.e., the abort propagates "up" the tree of nested sagas, towards the root. In our example, this would mean that an abort of the "safety inspection" saga would cause the "inspection" saga (its parent) to be aborted. This, in turn, would cause the "vehicle registration" saga to abort, and so on.

Frequently, such cascading of aborts is a desirable behavior. For example, the motor vehicle department may find it reasonable to abort the entire "inspection" activity if the inspection payment cannot be made. However, this is not always the case. Sometimes it will be preferable to attempt some alternative activity if a step fails, or to retry the failed step, or perhaps to do nothing at all. To accomodate this, the ***non-vital*** option can be specified (along with the name of the program to execute) when a step is created. When a non-vital step is aborted, its parent saga is not automatically aborted. However, such a step is still considered a part of its parent's saga, so that an abort of the parent will result in the (recursive) abort of the child. The non-vital option provides a mechanism for breaking the default cascade of aborts "up" the tree. Alternatively, the ***independent*** option can be supplied when a step is created. The independent option breaks the default cascade of aborts up the tree and down the tree. Thus, an abort of the parent does not affect the child, and vice versa.

3.4 Compensation Bind

The *Compensation_Bind* call is used to specify the compensating steps that should be executed in case a saga aborts. Any step can have a compensating step specified for it. The compensation is executed if the step has committed and is part of the saga that later aborts.

The *Compensation Bind* call takes the names of two steps (which we will call "forward" and "reverse") as arguments. Both of these steps must already have been created using the *Create* call. The call indicates to the system that "reverse" should be used as the compensating step for "forward", in case compensation is necessary. In response to such a command, the system ensures that the "reverse" step does not run until such time as compensation is required for "forward".

3.5 An Example

Figure 2 shows a simple program called "vehicle registration" that, when run, produces the registration activity. The program creates a step for each of the five sub-activities of registration. In addition, it creates compensating steps for the "inspection" and "reserve_number" sub-activities, in case compensation is necessary. The calls to *Compensation_Bind* are used to relate these compensation steps to the corresponding forward steps. Finally, the "registration" program itself commits, indicating that the new steps that it has created can begin execution. If the registration program were to abort for any reason before the call to *Commit*, it would be as if none of the system calls in the program had ever been executed, i.e., none of the newly created steps would exist.

```
Program: registration;
[
        ...
        sr ← Create( program: "submit request" )
        rc ← Create( program: "registration check" )
        in ← Create( program: "inspection" )
        ar ← Create( program: "assign registration number" )
        pr ← Create( program: "produce registration" )
        ar' ← Create( program: "release registration number" )
        in' ← Create( program: "invalidate inspection" )
        Compensation_Bind( forward_step: ar  comp_step: ar' )
        Compensation_Bind( forward_step: in  comp_step: in' )
        ...
        Commit()
]
```

Figure 2 – Vehicle Registration using Saga System Calls

Most of the steps created by "vehicle registration", e.g., "generate_number", will themselves perform the work necessary to accomplish their task. However, the "inspection" step is itself composed of several sub-activities. Thus, "inspection" will be a program much like "registration", using the system calls to create new steps to do the necessary work.

The environment provides several additional system calls and services not discussed here. In particular, there are calls to "bind" related steps together for the purpose of sending and receiving messages, and a mechanism by which the system notifies a step of the commit or abort of a saga. Binding steps together creates persistent mailboxes into which the bound steps can place an retrieve messages. The persistence of the mailboxes ensures that messages produced by a step survive any failures that might occur after the step has committed. As is true of the other system services, step bindings and message delivery are not effective until the requesting step has committed. Further details concerning the environment and nested sagas, along with a discussion of related work, can be found in [Garc90].

4. Conclusion

We have described nested sagas and have illustrated how they can be used to model long-running, multi-step activities. Such activities are common in business data processing environments, but it is difficult to support them using existing transaction models. The saga model allows applications to create and abort long-running activities without the high cost of long transactions. In addition, we presented a simple environment for the specification and execution of these activities. The environment consists of calls to a "saga processing system" that allow applications to define sagas and to indicate when they should be committed or aborted.

References

[Garc87]
H. Garcia-Molina, K. Salem, "Sagas," *Proc. 1987 SIGMOD International Conference on Management of Data*, May 1987, pp. 249-259.

[Garc90]
H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem, "Coordinating Multi-Transaction Activities," CS-TR-2412, University of Maryland, Dept. of Computer Science, College Park, MD, February, 1990.

[Giff85]
D. K. Gifford, J. E. Donahue, "Coordinating Independent Atomic Actions," *COMPCON85 Digest of Papers*, San Francisco, CA, IEEE Computer Society Press, February, 1985, pp. 92-95.

Making Progress in Cooperative Transaction Models

Gail E. Kaiser
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
kaiser@cs.columbia.edu

Dewayne E. Perry
AT&T Bell Laboratories
Room 3D-454
600 Mountain Avenue
Murray Hill, NJ 07974
dep@allegra.att.com

In the classical transaction model, transactions are consistency preserving units: a transaction is made up of a series of actions which, when executed in isolation in a reliable environment, transforms the database from one consistent state to another [2]. A classical transaction management system guarantees the *appearance* of isolation and reliability, even though multiple transactions execute concurrently and hardware and software components fail. It does this by enforcing atomicity and serializability: atomicity means that either an entire transaction apparently executes to completion or not at all, while serializability means that the effects of the transactions are viewed as if the transactions had executed in some serial order, one completing before the next begins. This is accomplished by considering the objects read and written by concurrently executing transactions, and ensuring that either all updates are completed or none are, and that the read and write dependencies among the set of transactions correspond to some serial order of the transactions.

There have been many proposals for *extending* the transaction model from its original data processing applications to software development, CAD/CAM and other forms of cooperative work. The notion of ''transaction'' is intuitively appealing in these domains, since *forward progress* often depends on making a related set of changes to a program, design or document in such a way that it is transformed from one consistent state to another. As with data processing applications, ''consistency'' depends on the requirements of the domain, such as a new system configuration passing regression and acceptance tests before the changed modules can be committed.

However, it is well-known that atomicity and serializability are inappropriate for interactive cooperative work [9, 13]. A task that might be treated as a ''transaction'', such as responding to a modification request, may be of *long duration* — hours to days to weeks — while information must be *shared* among the perhaps large numbers of personnel who participate in the concerted effort during the process of making the changes. Such a task is typically broken down into several subtasks, one per developer, that are carried out in parallel, and together bring the software system under development from one consistent state to another.

But even these subtask ''transactions'' may be very long and need to exchange information while they are in progress, and in any case one subtask is unable by itself to maintain global consistency. Thus the conventional understanding of transactions as failure recovery and serialization units is unacceptable for cooperative work applications. In particular, recovering from failures by rolling back to the beginning of a (sub)task and starting again is rarely appropriate for human developers who would then have to redo much of their work, while serializing developer (sub)tasks does not permit cooperation while they are in progress.

The notion of *cooperative transactions* has been devised (under many different names) not only to provide the same intuition as conventional transactions but also to support the requirements of cooperative work. The gist of many cooperative transactions models is that transactions are assigned to groups, where the transactions within a group employ a different concurrency control policy among themselves than with respect to transactions in other groups [4, 14]. Concurrency control is typically *relaxed* within a group, *e.g.*, allowing simultaneous updates to multiple versions of an object or allowing reads of uncommitted updates. Among groups, however, a stricter policy such as serializability is common. Cooperative transaction groups allow collaboration among the designated members of a group, through exchange of partial results, but enforces isolation among groups. See [1] for a survey of the literature on concurrency control policies and mechanisms for cooperative work.

We are concerned in this paper with one particular shortcoming of nearly all the cooperative transaction schemes. The shortcoming we have in mind is due to the problem of *human management* of *in-progress* software development processes. Cooperative transactions are designed to isolate groups from other groups and individuals, to allow cooperation among the members of a group while they carry out their tasks, and to implement the notion that the finalized set of updates made by group members is completed and released atomically. The *external view* outside the group sees only the released system, and not the partial results of in-progress work. However, the human managers of a software development project cannot wait for the released version of the system! They must be able to determine progress at a much finer granularity, dependent on the policies of the organization.

One possible approach would be to place the managers and all the personnel they manage in the same group, but for large organizations this defeats the purpose of concurrency control. Everyone may arbitrarily overwrite everything, without satisfying the consistency constraints of *forward progress*! Alternatively, each manager could be placed individually in every relevant group, as is possible in the participant transactions model which permits overlapping groups [7], but the symmetric cooperation implied between these managers and the other group members may be undesirable. Not only can the manager see the up-to-the-minute work of the developers, but the developers may inspect whatever the manager is doing; this is not usually acceptable for real projects.

Further, it would be preferable to provide the managers with an abstract view concerned with the software development *process* as opposed to the details of development *products* [10]. Managers are generally concerned with the results that have been accomplished thus far, and are not terribly interested in which versions of which files are currently being edited. Thus we propose that any cooperative transaction model should be augmented with a distinct *internal view* that supports the human management process, rather than trying to impose such access within the particular cooperative transaction model.

We have devised a general solution to this problem by analogy to the internal view of conventional transaction management systems that implement the classical transaction model. A transaction manager monitors the status of the currently in-progress transactions as well remembering the updates made by the previously committed transactions. The transaction manager interacts with various resource management systems, since transactions compete for computation cycles, primary memory, etc., and maintains internal structures to guarantee atomicity and serializability, including locks, logs, queues, shadow copies of data items, timestamps, distributed transaction coordinators and cohorts, and so on. The transaction manager sees a highly dynamic database, while an end-user of even a cooperative transaction system sees a relatively static database, since the database appears to change only when an update to an individual data item is completed (e.g., cooperating users probably would not share editing buffers, but only saved files).

We propose to solve the *impedance mismatch* between cooperative transaction models and the requirements of human management by *unveiling* the internal transaction management view at an appropriate level of abstraction. Like the transaction management system, the human manager can then use this internal view to manage resources, detect errors, recover from faults and in general monitor progress, but in terms of the software development process rather than data management structures. That is, the human manager manages human and machine resources, detects erroneous interface and work breakdown assumptions by development staff, recovers from these faults by reassigning responsibility and correcting misconceptions, and in general keeps track of how far behind the schedule has slipped.

The human manager can go further than most transaction management systems, to *restructure* and change the direction of software development tasks, for example, to dramatically scale down the planned software product. On the other hand, transaction management systems can *guarantee* certain invariants, such as "all faults will be recovered" and "all deadlocks will be detected", while human management is itself fallible.

More specifically, we notice that every traditional transaction management system has two views, the *transaction view* of committed transactions and the *transcendent view* of in-progress transactions. The transaction view is explicit and externally visible; the transcendent view is implicit and internal to the transaction manager itself. In traditional applications, there is no reason to make the transcendent view explicit, and many good reasons to keep system implementation details hidden from applications programmers as well as end-users. The goal of a traditional transaction management system is that if the application programmer defines his program according to the transaction view using the primitives provided (*e.g.*, begin-transaction/end-transaction blocks), then the transaction manager will use its transcendent view to monitor global progress in order to guarantee atomicity and serializability. It is not necessary for the application programmer to be concerned with the details of how this is accomplished, and hiding this transcendent view is generally believed to ease the application programming effort.

A cooperative transaction manager should have similar transaction and transcendent views that support a cooperative work concurrency control policy and a consistency model specific to the particular cooperative work domain. Our solution to the human manager's dilemma is to *extend* such systems to uncover a portion of the already existing transcendent view to make it explicit and visible to selected end-users (*i.e.*, the managers). In particular, we provide an *abstract view* of the internal structures and mechanisms that implement the cooperative transaction manager's concurrency control protocol, while continuing to abstract away from the lower level layers that support the failure recovery protocol, physical data management, and so forth, which vary with the implementation. The result enables the human manager to monitor the progress (or lack thereof) of the software project, and take appropriate action, in the context of whatever cooperative transaction mechanism is employed.

Our preliminary ideas have been implemented in the INFUSE software development environment [6], which supports change management and integration testing. INFUSE is intended to support very large teams of developers, where it is crucial for the environment to enforce policies regarding the degree and style of cooperation among the developers [11]. This is achieved in INFUSE as follows.

A set of modules is selected in advance to be modified as part of a scheduled change. This change set is automatically partitioned into a hierarchy of what we call experimental databases using a simple module interdependency metric [8]. Developers make their changes to leaf experimental databases consisting of one or a few modules, invoke static semantic consistency checks and perform unit testing, and deposit their changes to the parent experimental database only when their work satisfies pre-specified constraints. Within a shared experimental database, the multiple developers *integrate* their changes, invoking inter-module static semantic consistency checks and applying regression and integration test suites, before the set of modules can be deposited to the next level. At the top of the hierarchy, acceptance tests must be passed before all the changes can be deposited to form the new baseline version of the system. The use of a module interdependency metric in forming the hierarchy follows the theory that changes to strongly coupled modules are more likely to affect each other, and thus should be integrated as early as possible, while changes to weakly coupled modules can be delayed until later on when higher levels of the hierarchy are integrated.

Thus, INFUSE isolates individual developers and groups of developers following an isolationist concurrency control policy. The hierarchy is strictly partitioned, so it is never the case that more than one writer has access to a given module in a leaf experimental database, and readers who own sibling experimental databases cannot access the module until all changes have been completed in the leaf and deposited. Whenever it becomes clear at higher levels of the hierarchy that additional changes are required, the current experimental database is repartitioned below that point to enforce such isolated access while further changes are made.

However, INFUSE recognizes the pragmatic requirements of human managers to keep tabs on the progress of the scheduled change. Managers and other privileged users may make queries that cut across the hierarchy of experimental database, in order to determine the up-to-the-minute status of all modules in the change set. The managers can display a snapshot of the hierarchy at any given moment, or request a history of previous partitionings and repartitionings due to repeated changes (the "yo-yo" effect) required below a selected node in the hierarchy. Further, any developer may request creation of a workspace [5] that gathers together a selected set of disjoint experimental databases, with the permission of the other developers affected, in order to carry out early consistency checking and testing with respect to modules otherwise isolated from each other until higher levels of the hierarchy. This is useful when it is known that the scheduled changes will result in greater coupling among these modules or otherwise specifically involve these modules' interfaces.

Thus INFUSE supplies a transcendent view to managers of software development projects, to aid them in monitoring the *progress* of the software development process. This transcendent view is lacking in all other cooperative transaction models that we know of, even though it is clearly required for most practical applications. Unfortunately, the transcendent view we have developed is not sufficient in itself since the access it provides is read-only. In particular, it is not possible in INFUSE for a manager to modify the *organization* of in-progress transactions when a problem is discovered. Therefore, the transcendent view capability must be coupled with some facility for restructuring in-progress transactions, such as the split-transaction and join-transaction operations [12].

The split-transaction operation allows one on-going transaction to be split into two or more transactions as if they had always been independent, separating the data items accessed by the original transaction among the new transactions in a serializable manner. New developers may take over the new transactions, to improve progress towards the goal of a coherent working system. The join-transaction operations permits two or more on-going transactions to be joined into one, combining the data items accessed by the originally separate transactions as if they had always been part of the same transaction, so that the changes are released together.

The join-transaction operation is relatively easy to implement, but the split-transaction operation requires support to aid in the determination as to whether the desired split is valid. Both operations require aid from the software development environment in notifying affected developers of the potential changes to their work assignments and checking whether these changes make sense from the viewpoints of the individual developers. In INFUSE, the split-transaction and join-transaction operations would be implemented by dividing and merging experimental databases, respectively.

The transcendent view augmented with these transaction restructuring operations seems sufficient to support practical human management considerations towards making forward progress during software development. It seems possible to apply these ideas to a range of cooperative transaction models, not just INFUSE, and to support other cooperative work applications, not just software development. It should not be very difficult to implement the transcendent view as part of a transaction manager already supporting a cooperative model, by making available an abstraction of the existing internal processes and structures. The feasibility of augmenting another transaction model with the split-transaction and join-transaction operations has previously been shown [3].

Acknowledgments

Yoelle Maarek, Bill Schell and Bulent Yener worked with us on the implementation of INFUSE. We would also like to thank Andrea Skarra for her comments on a draft of this paper.

Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, DEC, IBM, SRA, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

References

[1] Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *Computing Surveys*, 1991. In press.

[2] Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading MA, 1987.

[3] Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In Hector Garcia-Molina and H.V. Jagadish (editor), *1990 ACM SIGMOD International Conference on Management of Data*, pages 194-203. Atlantic City NJ, May, 1990. Special issue of *SIGMOD Record*, 19(2), June 1990.

[4] Amr El Abbadi and Sam Toueg. The Group Paradigm for Concurrency Control Protocols. *IEEE Transactions on Knowledge and Data Engineering* 1(3):376-386, September, 1989.

[5] Gail E. Kaiser and Dewayne E. Perry. Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution. In *Conference on Software Maintenance*, pages 108-114. IEEE Computer Society Press, Austin TX, September, 1987.

[6] Gail E. Kaiser, Dewayne E. Perry and William M. Schell. Infuse: Fusing Integration Test Management with Change Management. In *COMPSAC 89 The 13th Annual International Computer Software & Applications Conference*, pages 552-558. IEEE Computer Society, Orlando FL, September, 1989.

[7] Gail E. Kaiser. A Flexible Transaction Model for Software Engineering. In *6th International Conference on Data Engineering*, pages 560-567. IEEE Computer Society, Los Angeles CA, February, 1990.

[8] Yoelle S. Maarek and Gail E. Kaiser. Change Management for Very Large Software Systems. In *7th Annual International Phoenix Conference on Computers and Communications*, pages 280-285. Computer Society Press, Scottsdale AZ, March, 1988.

[9] Erich Neuhold and Michael Stonebraker (editors). Future Directions in DBMS Research. *SIGMOD Record* 18(1), March, 1989.

[10] Dewayne Perry (editor). *5th International Software Process Workshop: Experience with Software Process Models.* IEEE Computer Society Press, Kennebunkport ME, 1989.

[11] Dewayne E. Perry and Gail E. Kaiser. Models of Software Development Environments. *IEEE Transactions on Software Engineering*, March, 1991. In press.

[12] Calton Pu, Gail E. Kaiser and Norman Hutchinson. Split-Transactions for Open-Ended Activities. In Francois Bancilhon and David J. Dewitt (editor), *14th International Conference on Very Large Data Bases*, pages 26-37. Los Angeles CA, August, 1988.

[13] Lawrence A. Rowe and Sharon Wensel (editors). 1989 ACM SIGMOD Workshop on Software CAD Databases. February, 1989.

[14] Andrea H. Skarra and Stanley B. Zdonik. Concurrency Control and Object-Oriented Databases. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, New York, 1989, pages 395-421, Chapter 16.

An Interactive Transaction Model for Distributed Cooperative Tasks

K. C. Lee, W. H. Mansfield and Amit P. Sheth

Bellcore
kasey,whm,amit@thumper.bellcore.com

Abstract

We investigate transaction management support for applications in a multimedia telecommunication environment. Implementing the applications as distributed cooperative tasks operating on shared objects is expected to provide the needed flexibility and reliability. We propose an interactive transaction (ITX) model that allows definition, monitoring, and real-time control of the cooperative tasks. To support the desired level of consistency in managing the shared objects, it provides a framework for using correctness criteria in various extended transaction models and supports a new correctness criterion, called *fixed point*, that is motivated by the applications of interest. By enforcing a *partial fixed point* execution criterion, we can also achieve the effects of many extended active database mechanisms such as triggers, constraints, and active views, in a unified conceptual framework. An extended version of this paper can be obtained from the authors.

I. Introduction

With advances in communication and computing technologies, current voice-based telecommunication networks are expected to evolve towards multimedia communication environments. An example of a basic application (also called "service") in this evolving environment is that of a multimedia conference application. It supports multimedia communication among a number of users at different locations at the same time.

We investigate implementing applications as a set of cooperative tasks manipulating shared objects to achieve the application objectives. Shared objects can be application objectives, state information and media resources. Since the shared objects can be stored reliably in databases, and the control is distributed, the distributed cooperative task model is robust to local failures and allows more flexibility.

Transaction support for cooperative tasks on shared objects is a complicated and application dependent problem [3]. To support applications in a multi-media communication environment, we propose the interactive transaction (ITX) model that supports several features not found in earlier extended/long-running transaction models (e.g., [6] [4] [8] [2] [9] [1] [5]).

Unlike most of these models which execute subtransactions only once, an interactive transaction, ITX, is a feedback control process that interacts with the environment iteratively to satisfy (possibly user defined) cooperative objectives. Cooperative objectives are defined in terms of the observations (of the types defined later) on the objects shared by the cooperating ITXs. The system appears to be in a stable state to an ITX when its observations remain unchanged. Such stable states are characterized by the correctness criterion of *fixed point.*

To achieve the cooperative objectives, multiple ITXs indirectly interact with each other by issuing transactions (TXs). TXs are atomic, but need not be serializable. An ITX remains active until some predefined termination condition is met. While an ITX is active, it monitors and reacts to the changes in the states of the shared objects, possibly by repeated execution of its TXs. The execution of the TXs is controlled by the correctness criteria of the ITX. Both the termination condition and the correctness criteria are specified by the application logic and the users. In the multimedia conference application, a set of ITXs, one for each user participating in the conference call, monitor and manipulate the shared objects to achieve the next objective. Authorized users can add change media type or request addition of a user or media by simply changing the states of shared objects. In response, ITXs execute some or all of its TXs to allocate or deallocate the resources, or propose alternatives by updating shared objects affecting other ITXs. Eventually, the system will reach a fixed point that agrees with the cooperative objective.

The contributions of the ITX model are:

- A powerful high level feedback control framework supporting correctness criteria used in several extended/long-running transaction models [2] [7] [1] [9]. A correctness criterion is defined over execution states (as in most previous models) as well as input and output.
- A new *fixed point* correctness criterion for the definition, monitoring, and control of the distributed cooperative tasks implementing applications of our interest.
- A *partial fixed point* based execution control criterion supporting active database features such as trigger, constraints, active views, and snapshots.

Section 2 presents the proposed interactive transaction model and discusses how various correctness criteria can be integrated in the ITX model. Section 3 presents the partial fixed point control criterion and discusses why various extended database functionalities can be achieved by enforcing the partial fixed point correctness criterion. Section 4 identifies some of our future work.

II. Interactive Transaction Model

An application is implemented as a set of cooperative tasks, one for each participating user or user agent. In Figure 1, two cooperative tasks represented by $ITX1$ and $ITX2$ perform a cooperative activity by issuing a set of TXs. The shared objects are stored possibly in multiple and heterogeneous databases. The effects of the committed transactions lead to changes in the states of the updated shared objects and, in turn, are observed by all interested ITXs. An ITX observes the state changes (shown by observation arrows) by submitting a TX that returns the information about the states of the shared objects or by other implementation mechanism such as triggers (this is discussed further in subsection IIIB).

Definition: An ITX is (statically) defined as a tuple (ID, $\{TX_n\}$, ACC), where ID is the identifier, $\{TX_n\}$ is the set of n transactions, and ACC is the acceptable correctness criteria for the ITX.

Dynamic behavior of an ITX can be represented as a feedback controller using the ACC as a time-varying control function. In other words, an ITX uses its ACC to control the execution of the TXs. The ACC is composed of one or more correctness criteria defined over three types of observations, denoted $\{O_i\}$: input (i.e., values of the objects in the read set), output (i.e., values

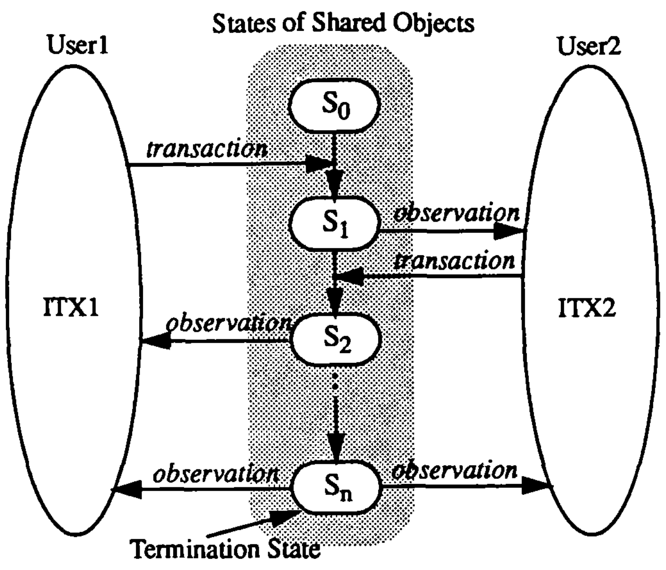


Figure 1: Execution of Two Cooperative Tasks

of the objects in the write set), and execution state (status) of each TX in $\{TX_n\}$. ACC can represent different correctness criteria used by various transaction models. Examples include:

1. **Goal States:** The set of observations, $\{O_n\}$, is the fail/succeed/executing/not-yet-issued (f/s/e/n) status of $\{TX_n\}$ [2]. Goal states allow for specifying a termination criterion such as, "If $\{O_n\}$ matches the specified pattern then terminate the execution of transactions." For example, the ACC of an ITX consisting of three transactions can be defined as a $\{O_1, O_2, O_3\}$ = $\{s, f, s\}$ or $\{s, s, s\}$. This ITX should terminate when the executions of the TXs result in either of the two execution statuses.
2. **Scheduling Dependency:** The set of observations, $\{O_n\}$, are the same as in the previous case. However, the ACC specifies a partial order on $\{TX_n\}$ based on $\{O_n\}$ to enforce *precedence predicates* and *temporal predicates* [2].
3. **Commit/Abort Dependency:** The set of observations, $\{O_n\}$, are either commitment or abortion states of the TXs. The ACC is defined as a partially ordered dependency graph involving the *commit dependencies* and *abort dependencies* among transactions [7].
4. **Data Access Control:** One can define access control criteria for the elements of $\{O_n\}$ in an ACC. For example, one can specify (a) data delegation criteria, as in the ACTA framework [7], (b) invariants between transactions, as in the migrating transaction model [1], or (c) the patterns and conflicts using finite state machines, as in the cooperative design model [9].

III. Dynamic Behavior and the Fixed Point Criterion

In this section, we present the intuition and definition of a *fixed point*, a new correctness criterion motivated by telecommunication applications, and discuss the dynamic behavior and control of ITX supported by a partial fixed point criterion.

A. Definition of Fixed Points

A distributed cooperative activity is accomplished by executing a set of ITXs. Cooperative objectives define fixed points to be reached iteratively by each ITX. For each iteration, the ITX

obtains a set of observations (either returned by its TXs or by a system supported mechanism). The observations resulting from (i.e., obtained during) the latest iteration are compared to the previous iteration. When two consecutive observations of all the TXs are equivalent, then, the system is at a fixed point. At a fixed point, the ITX remains active to detect further changes until a termination condition occurs. If a change is observed, the ITX will be reexecuted iteratively to reach a new fixed point.

We now define the fixed point criterion of an ITX. Each *execution trace* (i.e., iteration) of an ITX involves execution of some or all of its transactions in a partial order[1]. The mth execution trace of an ITX, $\{TX_i(m)\}$, is defined as $\{TX_1(m), ..., TX_i(m)\}$ where TX_i $(i \leq n)$ is the last transaction executed for the mth trace (m is called the trace number, i is called the transaction label). Let $AS(m) = \{O_i(m)\}$ denote the *observation set* of the mth trace of the ITX. It is a set of observations obtained corresponding to $\{TX_i(m)\}$.

Definition: An $AS(m)$ is at a *partial fixed point* up to $TX_j(m)$ iff $\{O_i(m-1)\} = \{O_i(m)\}$ for $i = 1$ *to* j and $j < n$ (where both observation sets are obtained by executing the TXs in the same order). $AS(m)$ is a *fixed point* of the ITX iff $\{O_n(m-1)\} = \{O_n(m)\}$.

To further improve the dynamic adaptation capability, we propose a partial fixed point criterion by specifying that a $TX_{j+1}(m)$ can only be executed when a partial fixed point is reached for $AS(m)$. This results in allowing an ITX to continue only if the previous transactions observed no state changes and prevents the ITX from progressing if the states of the shared objects have changed. Optionally, we can also specify that $AS(m)$ should reach a fixed point within time T_{out} to provide a time out (for deadlock resolution). The partial fixed point criterion is application-independent, and is orthogonal to the application-dependent components of the correctness criteria described in the previous section. Thus, the ACC for an ITX can be defined using the fixed point criterion along with zero or one or several application-dependent correctness criteria. Furthermore, the ACC can define temporal relations among multiple fixed points to govern the execution behavior of a long-running task. The following conference call example demonstrates this for observations involving output of transactions.

Example: Let an ITX be composed by two transactions. TX_1 observes the set of active conference call participants, L, and TX_2 controls whether the user is to drop out or join the conference. We can define the application-dependent ACC as a partially ordered sequence of conference objectives $\{\{john, mary, gary\}, \{\{mary, gary, X\}$ or $\{john, gary, X\}\}\}$. ITX will first try to reach the objective $\{john, mary, gary\}$, then to reach $\{mary, gary, X\}$ or $\{john, gary, X\}$. As a result, two sessions of a conference call can be defined in the ITX. When "john" or "mary " drop out of the first session of the conference call, a third person "X" must be connected.

B. Dynamics of Fixed Point Criterion

While the observations of fixed points are used for feedback control of the overall behavior of an ITX, the partial fixed point criterion provides a unified *conceptual* framework for supporting triggers, integrity constraints [5], intertransaction invariants [1], and active views. The state changes that affect the observation set of some transactions may be detected as violations of a partial fixed point. On detecting these violations, ITX reexecutes transactions to satisfy the partial fixed point criterion. The reexecution of a transaction based on state changes, can be used to implement

[1]The partial ordering of the executed transactions is determined by the application logic and may also be limited by the parallelism allowed in the implementation.

triggers, constraints, invariants, and active views. Such an implementation many not be efficient, however.

The partial fixed point criterion can be supported by observing changes in the observation set of the previously executed transactions. Observations can be obtained by (a) an iterative strategy that involves repeated execution of TXs executed so far, or (b) by system supported observation mechanism (such as trigger). If no change in the observation set is noted, ITX may progress by issuing additional transactions.

In the second strategy, when different observations are received corresponding to the objects accessed or manipulated by one or more previously completed transactions, the currently running transactions are aborted, and the affected transactions are reexecuted. This can transitively result in reexecution of additional transactions that depend on the reexecuted transactions. For ITXs consisting of large number of transactions, the second strategy is expected to be more efficient.

IV. Future Work

Planned future work include, (a) understanding various correctness criteria with respect to the multimedia communication applications and the ability to represent them in this model, (b) prototyping an application, (c) investigating efficiency and robustness issues of alternative implementations, and (d) implementing the ITX model and the prototype application using Prolog or C++, and using an object oriented data store for shared objects.

We would like to thank Nancy Griffeth, Will Leland, Brian Coan, Linda Ness, Jane Cameron and Ming Lai for their valuable comments that helped us to significantly improve the technical report and this short paper.

References

[1] J. Klein and A. Reuter, "Migrating Transactions," in *Proc. of Future Trends in Distributed Computing Systems in the 90's*, Hong Kong, 1988.

[2] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, "A Multidatabase Transaction Model for InterBase," in *Proc. of the 16th VLDB*, 1990.

[3] I. Greif and S. Sarin, "Data Sharing in Group Work," in *Computer-Supported Cooperative Work*, ed. Irene Greif, Morgan Kaufman Publishers, 1988.

[4] W. Litwin and H. Tirri, "Flexible Concurrency Control using Value Data," Technical Report No. 845, INRIA, May 1988.

[5] U. Dayal, M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions," in *Proc. of the ACM SIGMOD*, 1990.

[6] B. R. Badrinath and K. Ramamritham, "Performance Evaluation of Semantic-Based Multilevel Concurrency Control Protocols," in *Proc. of the ACM SIGMOD*, 1990.

[7] P. K. Chrysanthis and K. Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior," in *Proc. of the ACM SIGMOD*, 1990.

[8] H. Garcia-Molina and K. Salem, "Sagas," in *Proc. of the ACM SIGMOD*, 1987.

[9] M. Nodine and S. Zdonik, "Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications," in *Proc. of the 16th VLDB*, 1990.

Composing Multidatabase Applications using Flexible Transactions*

Yungho Leu
Department of Computer Science
Purdue University

1 Introduction

A transaction constitutes a unit of work in a database system. Systems that use transactions have to guarantee four basic transaction properties, namely, atomicity, consistency, isolation and durability (also called the ACIDity properties). Most DBMSs strive to guarantee these properties through the use of concurrency, commitment, and recovery algorithms. Transactions have been a successful technology for building meaningful and extensive applications over the last few decades. However, with the wide spread use of DBMSs in advanced applications, the suitability of these transaction properties has come under question. It has lately been argued that while it is desirable for the system to guarantee the ACIDity properties, it should be up to the applications to decide which of these properties they need to enforce and which they can trade for more flexibility or higher performance.

This paper introduces a new transaction model, called *Flexible* model, which relaxes two of these properties, namely, atomicity and isolation. While the *Flexible* model is studied in the context of the InterBase project[1], it is formulated and is intended for general use.

The new model outlines the goal of *flexible execution control.* Three important notions of the *Flexible* model are: function replication, dependencies (both external and internal) and compensatability. Suppose that an application is implemented as a transaction which has to perform a set of tasks. Function replication states that it is usually possible to perform a specific task of an application in more than one way (see the example in the next section). Therefore, we can compose a set of subtransactions to implement a specific task of an application. The set of subtransactions which implements the same task are said to be *functionally replicated* (or *functionally equivalent*)[RELL90]. Function replication enables us to have more than one acceptable execution path within a single transaction. As a result, transaction execution becomes resilient to failure. In the event that some subtransactions fail, the transaction can still be "successfully" executed. The execution paths of a transaction in the *Flexible* model are *nondeterministic*, i.e. the actual path of execution depends on the patterns of subtransaction failure occurring during the transaction execution. It is also possible to leave it up to the user to decide which path of execution to commit by scanning through all paths of the transaction execution.

*N. Boudriga, A. Elmagarmid, E. Kuhn and M. Rusinkiewicz have all worked on this project and contributed to various stages of the work reported in this paper.

[1]InterBase is a project in the Indiana Center for Database Systems that studies issues of transaction management and consistency in the multidatabase area. The InterBase prototype has been built and it currently includes Sybase, Ingres, Guru, Dbase IV, and Oracle. In addition it also integrates various other non-database packages.

The notion of dependency is very important in the *Flexible* model. Two categories of dependencies, *external dependency* and *internal dependency* can be specified over the set of subtransactions of a global transaction. External dependency specifies the dependency of the execution of subtransactions on events or objects outside the transaction. For example, we can specify the dependency of subtransactions on time or on cost functions. The external dependency provides useful information for scheduling the execution of subtransactions to preserve the *execution autonomy*[2]. The *Flexible* model also allows a user to specify the internal dependencies which relates the subtransactions of the same transaction. The internal dependencies explained in this paper are *success dependency* and *failure dependency.* Other useful internal dependencies are *commit dependency* and *abort dependency* [CR90].

Finally, a transaction in the *Flexible* model can have two types of subtransactions, compensatable and non-compensatable. This results in mixed transactions. The mixed transaction concept allows flexible control of the isolation granularity of transactions. Those subtransactions which are non-compensatable must run in isolation of the rest of the system (i.e. maintaining isolation property), while the compensatable subtransactions can be committed once they are completed and, therefore, reveal their effects to other transactions before their composing transactions commit (i.e. compromising isolation property) [ELLR90]. By properly specifying the types of subtransactions, we can control the isolation granularity of a transaction to be as large as the whole transaction (as in nested transactions [Mos81]) or as small as subtransactions (as in Sagas [GMS87]).

This paper is organized as follows. In Section 2, we present the *Flexible* model by describing the form of a *Flexible* transaction and giving an example of a *Flexible* transaction. In Section 3, we present two methods for implementing the *Flexible* model. Section 4 summarizes this paper.

2 The Flexible Model

Dependencies:
Let us consider a transaction which consists of a set T of subtransactions, $T = \{t_1, t_2, \cdots, t_n\}$. The execution of a subtransaction t_i can depend on the failure or the success of the execution of another subtransaction. Furthermore, it can be dependent on some external parameters (such as time). More precisely, we define:

Success dependency: A subtransaction t_i is success dependent on subtransaction t_j if t_i can be executed only after t_j is successfully executed.

Failure dependency: A subtransaction t_i is failure dependent on subtransaction t_j if t_i can be executed only after t_j is executed and failed.

External dependency: Let X be a set of parameters (X is disjoint from T). A subtransaction t_i is externally dependent on X if the execution of t_i depends on the truth of a predicate on X.

Form of Flexible Transactions:
In order to capture the notion of compensatability of subtransactions, we use the concept of type: a subtransaction is said to be of type C if it is compensatable, it is of type NC if it is non-compensatable.

A *Flexible* transaction is formally defined as follows:

[2]The local database system decides when to execute a subtransaction for a global transaction [VPZ86].

Definition 1 *A Flexible transaction T is a 5-tuple (B, S, F, Π , f) where*

- *$B = \{t_1, t_2, \cdots, t_n\}$ is a set of typed subtransactions called the domain of T;*
- *S is a partial order on B called the success order of T;*
- *F is a partial order on B called the failure order of T;*
- *Π is a set of external predicates on B;*
- *f is an n-ary boolean function defined on the set $\{1, 0\}$ and is called the* acceptability function *of T.*

To illustrate the above definition, we use as an example the following travel agent transaction. *Example:* Consider a travel agent (TA) information system[Gra81]; a transaction in this system may consist of the following tasks:

1. TA negotiates with airlines for flight tickets.
2. TA negotiates with car rental companies for car reservations.
3. TA negotiates with hotels to reserve rooms.

Let us assume, now, that for the purpose of this travel, two airline companies (Northwest and United), one car rental company (Hertz) and three hotels (Hilton, Sheraton and Ramada) can be involved in this trip. The travel agent can implement these tasks as

1. Order-a-ticket from either Northwest or United airlines.
2. Rent-a-car from Hertz.
3. Reserve-a-room in any one of the three hotels.

These three tasks can be implemented respectively by three sets of functionally equivalent subtransactions: $\{t_1, t_2\}$, $\{t_3\}$ and $\{t_4, t_5, t_6\}$, where the t_i's are defined as follows:

t_1	Order a ticket at Northwest Airlines;
t_2	Order a ticket at United Airlines;
t_3	Rent a car at Hertz;
t_4	Reserve a room at Hilton;
t_5	Reserve a room at Sheraton;
t_6	Reserve a room at Ramada.

In addition, we assume the following: (1) the order-a-ticket subtransactions can not be compensated; (2) the order-a-ticket subtransactions must run within business hours from 8AM to 5PM and t_2 will be executed only after t_1 is executed and fails; (3) the rent-a-car subtransaction must be executed after the order-a-ticket subtransaction and the reserve-a-room subtransaction must be under the budget of \$100; (4) the transaction succeeds when order-a-ticket, rent-a-car and reserve-a-room succeed. We propose the following *Flexible* transaction for the travel agent transaction.

$$B = \{t_1(NC), t_2(NC), t_3(C), t_4(C), t_5(C), t_6(C)\}$$

$$S = \{t_1 \prec t_3, t_2 \prec t_3\} \qquad F = \{t_1 \prec t_2\}$$

$$\Pi = \{ P, Q \} \text{ where P and Q are two predicates defined by}$$
$$P = \{ 8 < \text{time}(t_1) < 17, \quad 8 < \text{time}(t_2) < 17 \}$$
$$Q = \{ \text{cost}(t_4) < \$100, \quad \text{cost}(t_5) < \$100, \quad \text{cost}(t_6) < \$100 \}$$

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 \wedge x_3 \wedge x_4) \vee (x_1 \wedge x_3 \wedge x_5) \vee (x_1 \wedge x_3 \wedge x_6) \vee (x_2 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_5) \vee (x_2 \wedge x_3 \wedge x_6)$$

Execution of a Flexible Transaction:
To discuss the execution of a *Flexible* transaction, we first introduce the notion of transaction execution states.

Definition 2 *For a Flexible transaction T with m subtransactions, the* transaction execution state *x is an m-tuple $(x_1, x_2, ..., x_m)$ where*

$$x_i = \begin{cases} N & \textit{if subtransaction } t_i \textit{ has not been submitted for execution;} \\ E & \textit{if } t_i \textit{ is currently being executed;} \\ S & \textit{if } t_i \textit{ has successfully completed;} \\ F & \textit{if } t_i \textit{ has failed or completed without acheiveing its objective;} \end{cases}$$

While successfully completed for a compensatable subtransaction means that the subtransaction is committed, successfully completed for non-compensatable subtransaction means that the subtransaction is in a *prepared state* [Gra78]. The transaction execution state is used to keep track of the state of execution of subtransactions in a *Flexible* transaction. The acceptability function appears as a partial function defined on the set of execution states. It is computable whenever all x_is occurring in its expression are equal to either S or F. Hence, the acceptability function reflects the acceptability of an execution state. Whence the following definition

Definition 3 *Let T be a Flexible transaction and X the set of its execution states. The acceptable state set, A, of the Flexible transaction is the subset*

$$A = \{ x \in X \mid f(x) = 1\}$$

In the previous example, the set of acceptable states is defined by

$$A = \{ (S, \_, S, S, \_, \_) \} \cup \{ (S, \_, S, \_, S, \_) \} \cup \{ (S, \_, S, \_, \_, S) \} \cup \{ (\_, S, S, \_, S, \_) \} \cup \{ (\_, S, S, S, \_, \_) \} \cup \{ (\_, S, S, \_, \_, S) \}$$

The execution of a *Flexible* transaction start from the initial transaction execution state with all state variables x_i $(i = 1, m)$ set to N, scheduling subtransactions for execution, and terminates either when no subtransaction can be scheduled or an acceptable state is reached. In the former case, we say that the transaction fails; while in the latter case, we say that the transaction succeeds. For a detailed scheduling algorithm, we refer the reader to [LEB90].

3 Implementation Methods

Two approaches have been proposed to implement the *Flexible* model [LEB90, KELB90]. The first approach is based on the *Predicate Transition Nets.* In this approach, a *Flexible* transaction is mapped into a Predicate Transition Net. We then use the derived Predicate Transition Net as an internal data structure to control the execution of the *Flexible* transaction. We refer to [LEB90] for the details of the control algorithm. The second approach is to use the *logic paradigm.* We have designed a parallel logic transaction language, called PLTL, which is an extension of the sequential PROLOG for parallel programming. We can specify a *Flexible* transaction program using the PLTL. Currently, we are prototyping an execution environment for the PLTL. For a detailed description of PLTL, we refer to [KELB90].

4 Conclusion

In this paper, we presented a new transaction model to support flexible execution control over transactions. The features of this model are useful for general applications. Especially, it is useful for transaction processing in multidatabase systems which is characterized by the requirement of local autonomy. As in a multidatabase system, the local database systems may decide whether or not and when to execute subtransactions for a global transaction. Flexible scheduling of the transaction execution is a useful means to facilitate transaction processing in this environment.

References

[CR90] P. K. Chrysanthis and K. Ramamtitham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 194–203, 1990.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. E. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of 16th VLDB conference*, August 1990.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM Conference on Management of Data*, pages 249–259, May 1987.

[Gra78] J. Gray. *Notes on database operating systems. Operating Systems: An Advanced Course.* Springer-Verlag, Berlin, 1978.

[Gra81] J. Gray. The transaction concepts: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154, 1981.

[KELB90] E. Kuhn, A. K. Elmagarmid, Y. Leu, and N. Boudriga. A parallel logic language for transaction specification in multidatabase systems. Technical Report CSD-TR-1031, Purdue University, October 1990.

[LEB90] Y. Leu, A. Elmagarmid, and N. Boudriga. Specification and execution of transactions for advanced database applications. Technical Report CSD-TR-1030, Purdue University, October 1990.

[Mos81] J.E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing,*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, April 1981.

[RELL90] M. Rusinkiewicz, A. Elmagarmid, Y. Leu, and W. Litwin. Extending the transaction model to capture more meaning. *ACM SIGMOD RECORD*, 19(1), March 1990.

[VPZ86] J. Veijalainen and R. Popescu-Zeletin. On multi-database transactions in a cooperative, autonomous environment. Technical report, Hahn-Meitner Institut, Berlin GmbH, Glienickerstrasse 100, D-1000 Berlin 39, FRG, 1986.

A Transaction Model for an Open Publication Environment

Peter Muth, Thomas C. Rakow, Wolfgang Klas, Erich J. Neuhold

GMD
Integrated Publication and Information Systems Institute (IPSI)
Dolivostrasse 15, D-6100 Darmstadt, West Germany
e-mail: {muth, rakow, klas, neuhold} @ darmstadt.gmd.dbp.de

Abstract

Different users of a publication environment, e.g., authors, editors, layouters, typesetters, will together produce and distribute a publication. We consider three different kinds of components in such an environment: assistants, agents, and tools. Assistants provide users with homogeneous access to local tools, such as text editors, information bases etc. In addition, they support the cooperation between users. Agents provide direct access to non-local tools, originally available only to the assistants of other users. We propose a transaction model with the following properties: easy integration of existing tools, guarantee of ACID properties by assistants and agents if tools supporting ACID transactions are accessed, and a higher degree of parallelism compared to conventional transaction models.

1 Introduction

In recent years the wide availability of desktop publishing systems has given the impression that new ways and processes for publishing will replace the traditional complicated multi-person publishing activities. However, it turned out that in most cases an author is unable to maintain the quality of publication that was ensured by many specialists in traditional publishing, e.g., co-authors, editors, layouters, typesetters etc. To support the traditional publishing process electronically by an open distributed system has become a major research and development topic.

In this paper, we investigate an open distributed publication environment for multi-media products that is being developed at our institute. It supports a multi-user environment, where individuals or groups play specialized roles but want to cooperate and use the system concurrently. They all require durability for their work, e.g. new parts of a document. Powerful versioning and undo-facilities are also needed if different versions of documents are to be explored or changes are subject to rejection. It turns out that traditional transaction management concepts offered by conventional database systems are not sufficient to cover these environments:

- The publication environment consists of lots of components. New components may be added any time. Users of the system need an integrated system interface as they may not be aware which components are activated by their operations. A consistent transaction behavior is required, even if some components have their own, non-changeable behavior.
- The various users of the system need to utilize existing information sources – databases, knowledge bases, even active system components. Many, but not all of them will offer atomic, consistent, isolated, and durable (ACID) transactions [BHG87]. The system should support global ACID properties if the involved tools provide them.
- Publication activities consist of many sub-activities, some of them of short duration, others lasting a long time. Some of them are quite independent, some are heavily dependent on each other. Both need transaction support.

We propose a modular, extendible transaction model that makes use of 'encapsulated' transaction management modules of existing components, but also allows the integration of new transaction concepts. We use open nested transactions as our basic model. By constructing trees of subtransactions that are managed relatively independent of each other, it allows the integration of existing components. In addition, using the semantics of operations provides for a higher degree of concurrency compared to transaction models that support read/write operations only.

Many other groups also investigate non-traditional transaction models. [WR90] proposes petri-net like synchronization mechanisms for intra-transaction parallelism (as suggested in [ELLR90] also) and persistency for every, even intermediate transaction state. Sagas [AGK87] provides a transaction model for heterogeneous systems, but global serializability is not guaranteed. [DE89] discusses a new correctness criterion for transactions in heterogeneous database systems by relaxing the serializability definition. This approach reflects the problems of integrating unchangeable existing transaction managers in presence of local autonomy. A concurrency control algorithm for cooperative editing, discarding serializability, is presented in [EG89].

The paper is organized as follows. After introducing the general architecture we describe the requirements of transaction management in our system. A solution based on open nested transactions is presented. We describe our extensions to the open nested transaction model which are needed to support the requirements. In addition, we point out some open issues for further research in this area. We will not describe algorithmic solutions. Instead, we will refer the interested reader to our other publications where appropriate.

2 Publication Environment

In this section, we define the architecture of our publication environment. It has to support the complete publication process, beginning with the authors work and ending with the distribution of the resulting publication. Typically, these tasks will take place at different locations on different nodes of a computer network. Such a large and complex environment cannot be defined as a closed system. Instead, we propose an open architecture, i.e., a system that consists of independently developed modules. This approach allows us to extend the system dynamically with new modules, and replace old ones by modules with richer and improved functionality.

2.1 Architecture

In order to support a high degree of flexibility, we define two major system components: assistants and agents. Assistants are system modules that support the users directly in their fields of expertise by providing access to tools such as text editors, information bases, video production environments, etc. Each user has his own assistant. If a user's task needs services from tools which are not directly accessible by the user's assistant, another assistant with access to these tools has to be consulted. An assistant may concurrently consult different other assistants for different parts of his task. These assistants may also consult other assistants for subtasks of the subtask, etc. For some tasks, no appropriate service might be found or exist in the system. In this case, the assistant detecting this lack of functionality will ask his user for help.

Because assistants are directly assigned to users, users may not want their assistant to be consulted frequently by other assistants. Instead, a user may offer independent agents to perform these tasks. Agents can access a subset of the tools which are available to the corresponding assistant. Therefore, the services provided by an agent are a subset of the services of the assistant, but are independent of the assistant. That is, agents can be purchased by other users and used directly by their assistants instead of consulting the foreign assistant. Foreign assistants have to be consulted only if no corresponding agent is available or the corresponding agent is not able to perform the requested task.

In general, assistants will reside at different nodes of a network. Agents which were purchased by a user for a specific task will typically reside on the same site as the user's assistant. Hence, consulting an agent instead of the corresponding assistant also saves communication costs.

New tools are included in the publication environment either by extending the functionality of existing assistant modules, or by defining a new assistant module. Corresponding extensions in the agent modules offer the facilities of the new tools directly to other assistants.

An Example

Assume an author uses the publication environment for writing a document (Fig. 1). His assistant provides an appropriate environment, but during writing, the author needs some additional facts about a topic. Assume further, these facts are stored in an information base which has been integrated into the publication environment, and the user owns an agent with access to this information base. The assistant will call the agent to execute the appropriate query, i.e., retrieve the additional facts. If the execution succeeds, the author will get the required facts and may continue to write the document. If not, the user's assistant will consult the assistant of the information base expert, who offered the agent. For example, this may happen due to a mismatch of entity names in the query and in the schema of the information base. Let us assume that the assistant of the information base expert has an agent which provides knowledge about the schema of the information base. In turn, this information is provided by access to a Knowl-

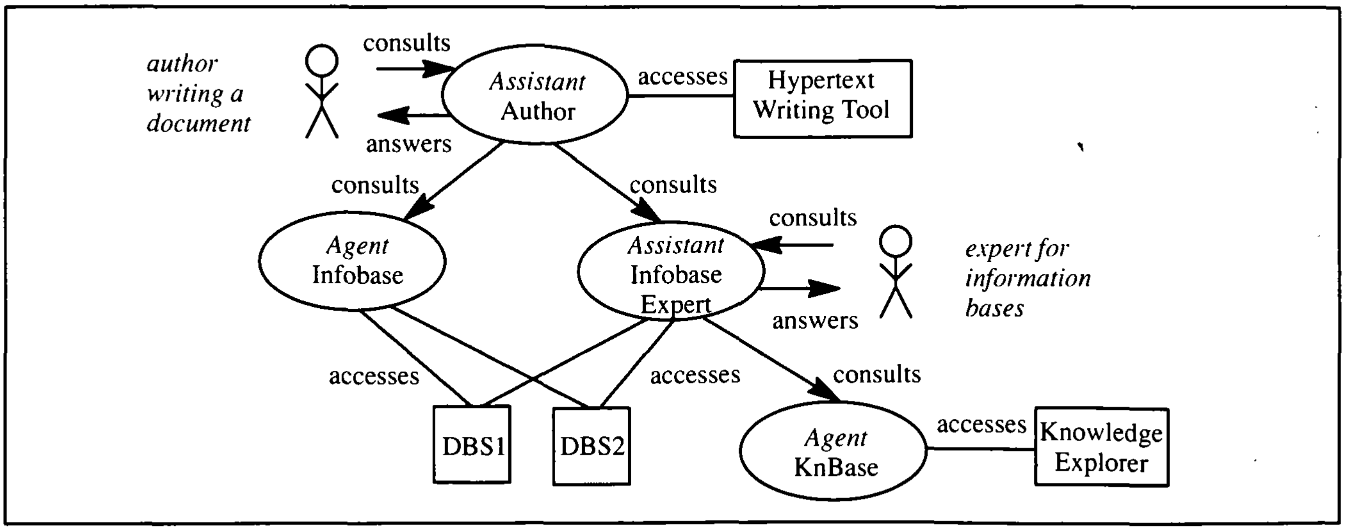


Fig. 1: An Example for an Author's Request.

edge Explorer [KN89]. If the Knowledge Explorer is able to find an appropriate transformation of the query, the query will be executed by the assistant of the information base expert, and the result will be sent to the assistant of the author. If all this fails, the assistant of the information base expert may contact the human information base expert for help.

The same procedure will be used for the next steps of the publication process. For example, the editor will request through his assistant parts of a book from the assistants of the authors. In turn, the type-setter will request the contents of the document from the editor's assistant.

2.2 Requirements with Respect to Transactions

From the above scenario, we can derive the requirements for the transaction model in our publication environment. Different from the use of isolated tools for editing, retrieving information from databases, defining the layout, etc., the persons involved in the publication process will interact through system facilities during the whole process. As a consequence, there is a need to control information exchange in order to avoid unintended visibility of intermediate data, and there is a need to control changes to the data, in order to avoid inconsistencies like lost updates and inconsistent reads [BHG87]. In addition, the system must provide an undo-operation for each user-operation, because complex operations may involve a lot of system components and may cause a lot of changes which are not known to the user. Undoing parts of the work of a user must not affect the whole global transaction, because too much previous work would be lost. Partial abort of user transactions is required.

Considering the technical level, new problems arise by the above requirements. The transaction model must support the whole publication process and all the involved components in the same way. Otherwise, a user has to deal with different transaction models and interfaces, which is not acceptable.

Since our architecture is open for the integration of new tools, the transaction model must also be able to support the integration of the transaction management schemes coming with the new tools. If it is not allowed or not possible to apply changes to a new tool, for instance, if it had not been designed to be used in the publication environment, the integration will be difficult. Sometimes the newly integrated systems will not provide an internal transaction management at all. In this case, we have to cover this lack of functionality as far as possible in order to provide the required homogeneous interface.

Existing database systems mostly provide ACID transaction properties for their internal transactions. In many cases, a user of the publication environment also needs ACID properties for his transactions. Hence, a transaction model must support the combination of ACID subtransactions into a global ACID user transaction. If autonomous transactions in an existing system are still allowed after the integration into the publication environment, special correctness criteria for the dependencies between the autonomous local transactions and the global transactions of the publication environment should be considered [DE89].

Other components may need a different notion of transactions [KS90]. For example, hypertext writing tools should support cooperation between authors editing the same document. Hence, a definition for cooperation between transactions of different users is needed. Because of this requirement, such transactions should be designated as *co-transactions* in contrast to ACID transactions. For co-transactions, the isolation property will be violated in a controlled way.

Therefore, the global transaction model in our environment has to support the combination of transactions with different correctness criteria into global transactions. The properties of these global transactions depend on the transaction properties of the involved systems. For example, if a user makes changes to data that is stored in a database and to text maintained by a hypertext editor inside of one transaction, ACID properties should be ensured for the users view of his changes to the database, independent of the non ACID properties for the changes in the hypertext editor.

3 Transaction Model

All these requirements can only be fulfilled by a modular, extendible transaction model, which is able to incorporate existing transaction management schemes such as ACID transactions, and new ones specially designed for components of the publication environment. We propose the use of open nested transactions [BBG89, MR91, RGN90, Wei86, WHBM90] as the general principle of our transaction model. After introducing the principle of open nested transactions we explain the usage in our environment.

3.1 The Principle of Open Nested Transactions

Open nested transactions constitute transaction trees (Fig. 2). The root node is defined as the user transaction, also called top-level transaction. The sons of a node are defined by the actions of the transaction assigned to this node. We call them subtransactions, because they can be divided into actions, too. This scheme continues until the actions are indivisible, e.g., page operations. In general, the depth of the tree varies for different transactions.

In contrast to closed nested transactions [Mos85], open nested transactions make the changes of a subtransaction visible at the end of the subtransaction, not at the end of the top-level transaction. But the results must not be visible to every other transaction. In order to avoid inconsistent use of the results of committed subtransactions, only those (sub-)transactions which commute with the committed one are allowed to use the results. Commuting means that the result of the execution of both transactions is independent of their execution order. At the end of the global transaction, the results of all subtransactions become unconditionally visible to other transactions, as it is the case in flat transaction models.

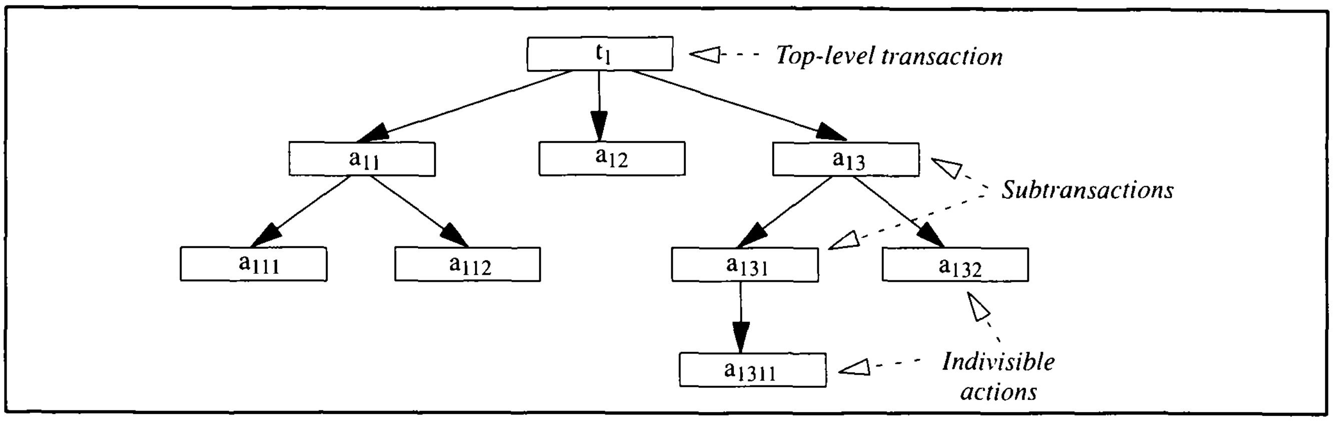


Fig. 2: An Open Nested Transaction.

Open nested transactions provide a higher degree of parallelism than conventional models, because of the controlled visibility of immediate subtransaction results. For example, two increments of a counter commute and can therefore be executed concurrently. If only read/write semantics are considered, two concurrent increments (=writes) cannot be allowed. A drawback of open nested transactions is the more complex recovery compared to conventional flat transactions. Recovery for open nested transactions requires the ability to undo committed subtransactions, if the calling transaction is aborted. Compensating subtransactions which execute inverse operations are used for this purpose [WHBM90]. Each subtransaction can be aborted independently of other subtransactions and its calling transaction, i.e., partial abort of global transactions is possible.

3.2 The Usage of Open Nested Transactions

In our publication environment, an assistant creates a top-level transaction for a specific task. Those parts of the task which can be executed locally by the assistant are executed as subtransactions in the corresponding tools. The other parts are given to agents of the assistant – if applicable – or to other assistants. An agent calls subtransactions on his tools, which will finally call indivisible actions. An assistant acts like the requesting assistant and calls his agents or other assistants. Recursive calls are allowed, but must not lead to an ultimate calling loop, i.e., an infinite depth of the transaction tree.

Example (continued)

The author's assistant creates a top-level transaction for getting the facts (Fig. 3). This results in subtransactions executed by the requested agents and assistants, and subtransactions executed in the accessed tools. The tools create actions on their own decision, which are independent from agents and assistants. The example only represents a querying function. However, notice that our transactions may also involve update actions, e.g., the growth of the knowledge base in the knowledge explorer.

The most important advantage of open nested transactions for homogeneous database systems is the concurrency gain. For the use in the publication environment, a second advantage becomes important: The concurrency control and recovery for different nodes of the transaction tree can be done by separate modules. For a more detailed discussion of transaction management at different nodes, we have to distinguish between two kinds of nodes:

1. nodes 'in the middle' of transaction trees and
2. leaf nodes.

Nodes 'in the middle' of transaction trees consist of subtransactions performed by agents and assistants. They execute at least some of their actions as subtransactions and have to check for the commutativity of these subtransactions with other subtransactions. Therefore, the concurrency control has to ensure serializability regarding the commutativity. By this scheme, a high degree of concurrency can be achieved, if the commutativity is defined with respect to the semantics of the subtransactions. For example, consider two transactions working on the final version of a document. Assume, the first transaction represents applying a spelling checker to the entire document, and the second transaction inserts a new section, which has already been checked for spelling errors. In conventional systems, the transactions cannot run concurrently, because both apply changes to the document. But the transactions commute, since the result of first executing the spelling checker and after that inserting a correct section is the same as first inserting and then checking for spelling errors.

Subtransaction calls between assistants may cross site boundaries. That is, a protocol for atomic commitment in the open nested model has to be used [MR91]. Even though the distribution makes the transaction model more complex, distributed transactions give us the chance of intra-transaction parallelism [Mos85].

In the accessed tools, no specific transaction management is necessary. Because no further subtransactions are executed, concurrency control can be done according to conventional, flat transactions. The same is true for recovery. Redo and undo can be done by simply using after and before images. Hence, existing transaction managers can be used for transactions that consist of leaf actions without any changes.

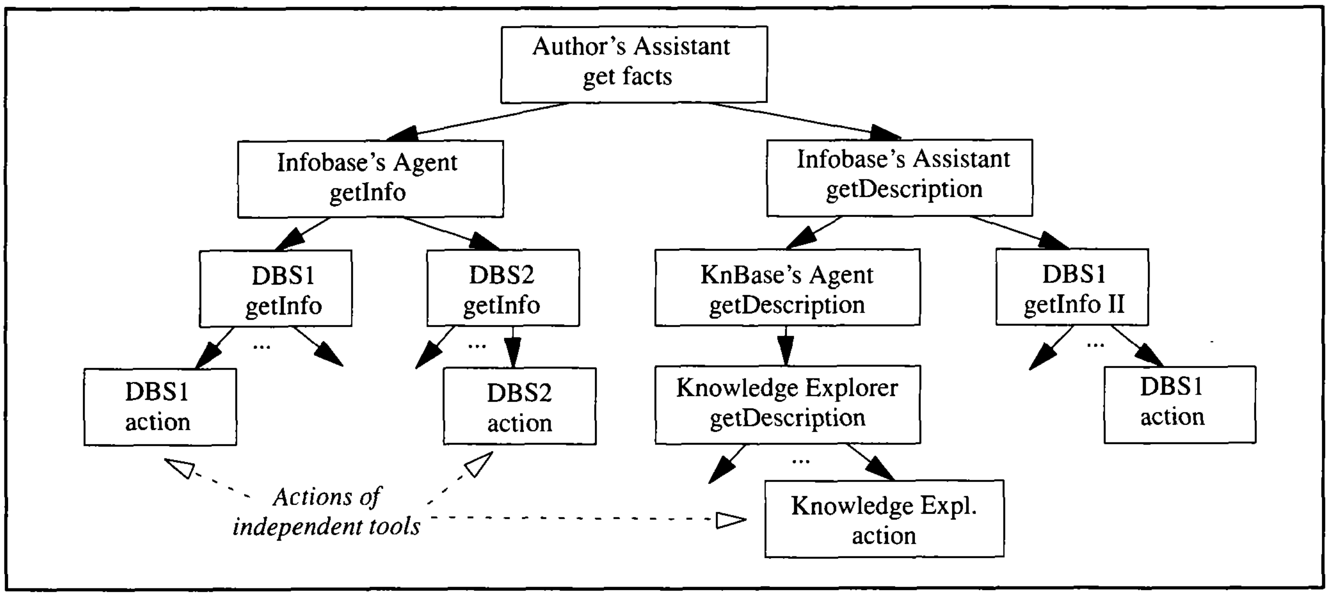

Author's Assistant get facts
Infobase's Agent getInfo
Infobase's Assistant getDescription
DBS1 getInfo
DBS2 getInfo
KnBase's Agent getDescription
DBS1 getInfo II
DBS1 action
DBS2 action
Knowledge Explorer getDescription
DBS1 action
Knowledge Expl. action
Actions of independent tools

Fig. 3: Transaction for the Example.

Different concurrency control and recovery algorithms may be used for transaction management of the tools. If ACID transactions are provided by a transaction manager, e.g., by a database system such as DBS1 of Fig. 1, these properties can be preserved by the transaction management of the agents and assistants. That is, our model provides ACID user transactions if all the involved tools also provide ACID properties for their transactions. If other transaction models, e.g., for hypertext systems, do not provide such properties, they may also be used as bottom-level transactions. The impact of the mixture of different bottom-level transaction properties on the properties of the top-level transaction will be subject of further studies, especially the consequences of integrating transaction models that allow cooperation between co-transactions.

4 Conclusion

We propose the use of open nested transactions for a publication environment. As a major achievement, our model allows the integration of new tools into the publication environment without changing the overall architecture. The integration is done by providing appropriate assistant and agent modules. The transaction managers of the new tools are simply plugged in to the global transaction management. If all the connected systems provide ACID properties for their transactions, the transactions of the publication environment users will also have these properties. In addition, our transaction model provides a higher degree of parallelism compared to flat models, and allows partial transaction undo by aborting subtransactions which are relatively short compared to whole user transactions. The proposed transaction model is currently under implementation within the project VODAK which goal is the development of an open object-oriented and distributed database system as a part of our publication environment.

References

[AGK87] R. Alonso, H. Garcia-Molina, K. Salem: Concurrency Control and Recovery for Global Procedures in Federated Database Systems. *IEEE Data Engineering* 10(3), Special Issue on Federated Database Systems, 1987.

[BBG89] C. Beeri, P. A. Berstein, N. Goodman: A Model for Concurrency Control in Nested Transaction Systems. *JACM* 36(2), 1989.

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

[DE89] W. Du, A. K. Elmagarmid: Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. *Proc. VLDB*, 1989.

[EG89] C.A. Ellis, S.J. Gibbs: Concurrency Control in Groupware Systems. *Procceding SIGMOD*, 1989.

[ELLR90] A.K. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz: A Multidatabase Transaction Model for InterBase. *Proceeedings VLDB*, 1990.

[KN89] M. Kracker, E. J. Neuhold: Schema Independent Query Formulation. *Proc. Conf. on Entity–Relationship Approach,*, 1989.

[KS90] H. F. Korth, G. D. Speegle: Long-Duration Transactions in Software Design Projects. *Proc. Conf. on Data Engineering*, 1990.

[Mos85] J. E. B. Moss: *Nested Transactions – An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, Mass., 1985.

[MR91] P. Muth, T. C. Rakow: Atomic Commitment for Integrated Database Systems. Accepted for: *Proc. Conf. on Data Engineering*, 1991.

[RGN90] T. C. Rakow, J. Gu, E. J. Neuhold: Serializability in Object-Oriented Database Systems. *Proc. Conf. on Data Engineering*, 1990.

[Wei86] G. Weikum: A Theoretical Foundation of Multi-Level Concurrency Control. *Proceedings PODS*, 1986.

[WHBM90] Weikum, G., Hasse, C., Broessler, B., Muth, P.: Multi-Level Recovery. *Proceedings PODS*, 1990.

[WR90] H. Wächter, A. Reuter: Grundkonzepte und Realisierungsstrategien des ConTract-Modells. *Informatik Forschung und Entwicklung* 5(4), 1990.

The ConTract Model

Andreas Reuter, Helmut Wächter

Institute for Parallel and Distributed High Performance Computers (IPVR)
University of Stuttgart, D - 7000 Stuttgart 1, Herdweg 51, Germany
email: {reuter, waechter}@ipvr.informatik.uni-stuttgart.dbp.de

1 Introduction

The limitations of classical ACID transactions have been discussed extensively in the literature [1]. Developed in the context of database systems they perform well when the controlled units of work are small, access only a few data items, and therefore have a short system residence time. Given this assumption, transactions could be made atomic state transitions. Atomicity, taken verbally, means that there is no structure whatsoever that can be perceived and referred to from the outside. In other words, if there is a unit of work that has a structure, say, in terms of control flow, which needs to be maintained by the system, it cannot be modeled as a transaction - and current database systems, operating systems, etc. have no other means for dealing with that.

Now in distributed systems and in so-called non-standard applications like office automation, CAD, manufacturing control, etc. one frequently finds units of work that are very long compared to classical transactions, touch many objects and have a complex control flow which may include migrations of (partial) activities across the nodes of a network [2]. Because the lack of appropriate system mechanisms to support this processing characteristics, controlling such activities requires organizational means or enforces the application itself to take care of, e.g. recovering the activity from a crash. But even simple examples like the mini-batch [3] demonstrate that then substantial parts of the code are not application-specific, but have to do flow control.

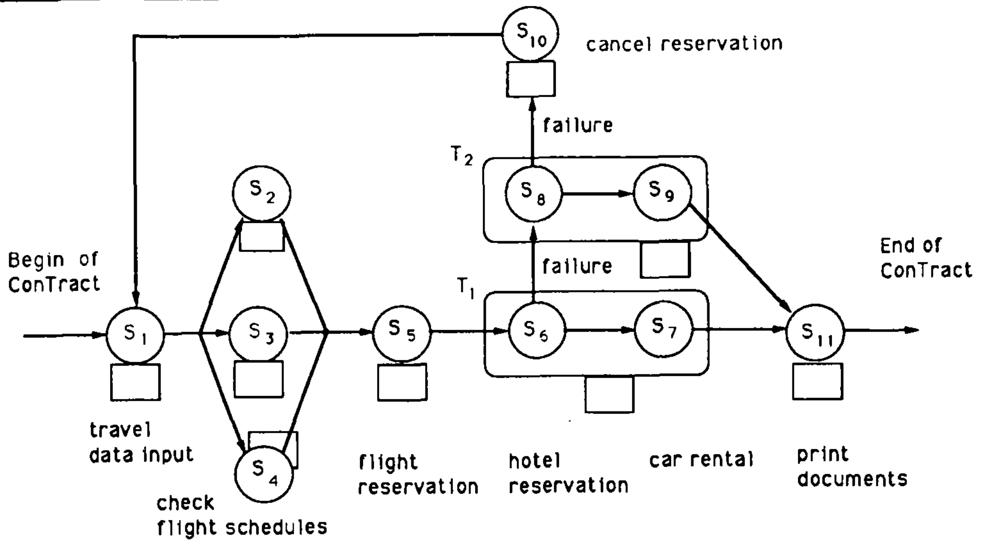
The ConTract-model, first proposed in [4], tries to provide the formal basis for defining and controlling long-lived, complex computations, just like transactions control short computations. It was inspired by the concept of spheres of control [5], and by the mechanisms for managing flow that are provided by some TP-monitors [3].

The key observation is quite simple: Since we want to control long-lived activities, the computation itself must be a recoverable object, and not just the state manipulated by it, as is the case with classical transactions. So any execution suitable for non-atomic computations must have the following properties:

- There must be a way to describe control flow in both static and dynamic terms.
- The computation must be forward-recoverable, i.e.rather than rolling back the whole thing in case of a component crash - which is unacceptable for long-lived activities - the computation must be re-instantiated and continued according to its specification.
- Since each computation needs its own local state (variables), this state must be recoverable, too. Of course, the database is assumed to be recoverable anyway.
- Long computations will have to externalize results before they are completely done. This implies that unilateral roll-back is no longer possible [6]; one rather needs to specify compensating actions as part of the control flow description.
- For the same reason, consistency definitions can no longer be based on serializability; they rather have to use invariants defined on global state [2], [7].
- In most cases, it is not feasible to let some activity wait (in case of a resource conflict) until a long-duration activity has completed. Therefore, part of the control flow description has to specify what should be done, if a resource conflict occurs, how it can be resolved, etc.

These key ingredients of the ConTract model are explained in some detail in the following sections by a (simplified) travel planning activity as illustrated in Fig. 1.

Doing flight, hotel and car reservations for a business trip is a typical activity that can be very long and sometimes needs more than one session to be completed. It is therefore not possible to do the whole reservation procedure within one transaction. To keep things simple there are only three airlines to be consulted for a flight and only two hotel resp. car rental companies. These give an exclusive discount to each other (in this example) and therefore are only booked in combinations (Cathedral Hill Hotel, Avis) or (Hilton, InterRent). We assume this application to be run on a terminal of a travel agency connected to a worldwide network of heterogeneous computers running the various databases.



```
ConTract Business_Trip

CONTEXT_DECLARATION
  cost_limit, ticket_price:     dollar;
  from, to:                     city;
  date:                         date_type;
  ok:                           boolean;
    :

CONTROL_FLOW_SCRIPT
  S1: Travel_Data_Input( in_context: ; out_context: date, from, to, cost_limit );
  PARBEGIN
     S2: Check_Flight_Schedule( in_context: "Lufthansa", date, from, to; out_context: flight, ticket_price );
     S3: Check_Flight_Schedule(    ...      "British Airways",  ...     );
     S4: Check_Flight_Schedule(    ...      "PanAm",      ...           );
  PAREND
  S5: Flight_Reservation( in_context:  flight, ticket_price;     ...    );
  S6: Hotel_Reservation( in_context:  "Cathedral Hill Hotel";  out_context:  ok, hotel_reservation );
  IF ( ok ) THEN  S7: Car_Rental(  ... "Avis" ... );
  ELSE BEGIN   S8: Hotel_Reservation( ... "Hilton"  ... );
               IF ( ok )  THEN      S9:  Car_Rental( ... "InterRent"  ... );
               ELSE  S10: Cancel_Flight_Reservation_&_Try_Another_One(.. );
       END
  S11: Print_Documents( ... );
END_CONTROL_FLOW_SCRIPT

TRANSACTIONS
        T1 ( S6, S7 ),  DEPENDENCY( T1:abort |---> begin:S8 );
        T2 ( S8, S9 ),  DEPENDENCY( T2:abort |---> begin:S10 );
END_TRANSACTIONS
SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTION
        S1: EXIT_INVARIANT( budget > cost_limit ), POLICY( check/revalidate );
        S5: ENTRY_INVARIANT( (budget > cost_limit)  AND  (cost_limit > ticket_price) );
            EXIT_INVARIANT( budget > cost_limit - tickte_price );
        S6, S8: ENTRY_INVARIANT( hotel_price < budget ),
                CONFLICT_RESOLUTION: S110: Call_Manager_to_Arise_Budget( .. );
        S7, S9: ENTRY_INVARIANT( car_price < budget ),
                CONFLICT_RESOLUTION: S120: Cancel_Car_Rental( .. );
END_SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTION
```

Figure 1: A sample script „Business Trip“. (a) Graphical, (b) textual representation.

2 The ConTract Model

A **ConTract** is an invulnerable execution of an arbitrary sequence of predefined actions (called *steps*) according to an explicitly given control flow description (called *script*).

2.1 Programming Model: Scripts and Steps

Scripts[1] constitute the central ConTract mechanism to extend control beyond transaction boundaries. Control flow between related steps can be modeled by the usual elements: sequence, branch, loop and some parallel constructors. It is also possible to specify a loop over a tuple set forming a query result, e.g. to consult n dynamically computed airline timetables in parallel.

The ConTract manager internally uses some sort of predicate transition net to specify activation and termination conditions for a step. For example, step S_5 in Fig. 1a is triggered when all three steps S_2, S_3 and S_4 are finished.

The basic idea is that with a script the programmer gives only a declarative description of how to run an application, e.g. how to concatenate steps to implement an application's task, to define application specific synchronization and recovery requirements and so on (see below). All aspects concerning execution control at runtime, however, have to be done by an application independent system service called ConTract manager.

Steps are the elementary units of work in the ConTract model. Each step implements one basic computation of an application, e.g. booking a flight, cancelling a reservation and so on. There is no internal parallelism in a step and therefore it can be coded in an arbitrary sequential programming language. Its size is determined by the amount of work an application can tolerate to be lost after a system failure.

In the ConTract programming model, coding these algorithmic parts is separated from defining an application's control flow. So the programming of a reservation step and the concatenation of steps to form the business trip script of Fig. 1 are two different tasks, which may be even performed by different people. The consequence, though, is that there are at least two ,,levels" of programming. The hypothesis is that this will be inevitable when specifying and implementing long-lived, complex applications, no matter which framework one uses.

The idea behind this separation is to keep the programming environment for the actual application programmer as simple as possible: Steps are coded without worrying about things like asynchrony, parallelism, communication, resource distribution (localization), synchronization and failure recovery. In particular, the programmer of a step does not have to consider whether a step or a set of steps (for instance (S_6, S_7) or (S_8, S_9)) is executed as an ACID transaction. This decision is made at the script level in the *Transaction* part of the specification, see Fig. 1b. Moreover, logical and temporal dependencies between the outcome (resp. activation) of transactions and/or steps can be defined there. *If in the example transaction* T_1 *fails, then* S_8 *should be started.* These dependencies are controlled by the runtime system, the ConTract manager.

From the programmer's view, steps will be run on a virtual machine which is arbitrarily reliable and executes in single user mode. How to achieve this is discussed in the next sections.

2.2 Forward Recovery and Context Management

System reconfiguration, communication failures, node crashes and other failures should not cause an application to turn undefined or, even worse, vanish without trace.

But that is what normal transactions would do for you without further application programming:

- An ordinary operating system process running application code is killed and forgotten after reboot. The user has to know which application was killed and has to recover it manually.

- A transaction system rolls back all uncommited operations, which doesn't matter for short transactions but is unacceptable for long lived activities.

A reliable system, on the other hand, would resume (automatically after system restart or on user demand) all ongoing computations and try to minimize the loss of work. In case a local computer fails during the sample

[1] A graphical editor would be the optimal choice for a user-friendly script definition. Since here we are not concerned about semantical aspects, a Concurrent Pascal like textual language is used (Fig. 1b). Of course, there are other syntactic means for specifying control flow, but this is not the point of this paper.

ConTract, the agent would like just to turn to another terminal and to continue the suspended reservation procedure right from the last valid ConTract state. The ConTract manager therefore tries to overcome resource failures and re-instantiates an interrupted ConTract by restoring the recent step consistent state and then continues its execution according to the specified script.

The realization of this forward oriented recovery scheme implies that all state information a step's computation relies upon has to be recoverable. This so-called *Context* comprises the set of data defining an application specific computation state. To re-instantiate an interrupted ConTract the following information is required

(a) the global state of the application, i.e. the involved databases;
(b) the local state of the ConTract, e.g. the program variables, sessions, windows, cursors etc. used by more than one step;
(c) the gobal computation state. This means a stable bookkeeping in the ConTract system of which event has triggered, which step has (or has not yet) been executed etc.

The context programing model is quite simple: Each step declares which context elements it expects to be available when it executes and what relevant state information for further steps is produced with its completion.

In essence, the need for robust context management comes in as soon as one wants to have guaranteed stability for long lived activities covering a set of related (trans)actions and finally ending up with the computation itself becoming a recoverable object.

2.3 Synchronization and Conflict Resolution

ACID transactions deal with concurrency by ignoring it; this is what serializability is all about. Since each transaction is atomic (and therefore small and short), it makes sense to maintain an execution order that is free of influences of one atomic state transition on another. Creating a serializable schedule is the simplest way to do that.

ConTracts are neither atomic nor short. They externalize some of their updates as they go (the term "commitment" should be avoided, because it has two aspects to it: updates are externalized and the right to revoke them is waived), but there is still a chance that these updates will be rolled back later on. Consequently, a ConTract might operate using data that have been externalized early by other ConTracts.

This problem can be solved by generalizing an idea that was already proposed for special types of hot spots [7]. Rather than holding locks on objects, one remembers the predicates that should hold on the database in order for the activity to work correctly. Put in a more application oriented style: No program needs serializability or even worries whether or not it is serializable. Its only concern is to keep the database free of unsolicited changes in the parts it works on. If this is guaranteed, this is isolated execution from that program's point of view. Now this observation is more than just another phrase for the same thing. Keeping the database free of unsolicited changes generally means much less than preventing all the attributes, tuples etc. that have been used from being modified at all. In many situations it is sufficient, e.g. to make sure that a certain tuple is not deleted; that a certain attribute value stays within a specified range; that there are no more than x of a certain type of tuples, etc. To implement synchronization based on the idea of "environmental invariance", the ConTract system needs two things:

First, it must be able to state the invariance predicates on the database defining its view of the world.
Second, it must be able to specify which of these invariants must be fulfilled for the next step to execute.

In the example of Fig. 1b, step S1 establishes that the travel budget (a tuple in the database) of the department was higher than the cost limit allowed for that trip. Before a flight can be booked (S5), this must still be true. At the end of this step the budget has been debited the ticket price, and so it only needs to be higher than the cost limit minus the ticket price. The other invariants follow the same logic.

Since there are purely declarative specifications, some hints are needed to tell the ConTract manager and the database system how to handle these invariants. One way to keep things "as they are" is to lock all objects as in today's systems. The ConTract manager at the DBMS would then have to manage long locks, i.e. locks that are held beyond transaction boundaries. Instead of locks, the DBMS could use semantic synchronization techniques like escrowing [8], if the operations have the necessary properties. The most liberal approach is to use no locks at all; this requires the check/revalidate technique [7].

Now if one accepts that the world might change while executing a ConTract, one has to cope with the situation of an invariant having changed, such that the next step cannot be executed - like the department budget being overdrawn in the example. In ACID transactions, such conflicts are not communicated to the application; rather the system decides whether the transaction is rolled back or just has to wait. Both approaches do not make much sense for long-lived activities. Therefore, ConTracts allow to explicitly talk about conflicts, and to specify actions for conflict resolution. In the example, somebody must increase the budget in case it does not hold enough money, or the whole business trip must be cancelled.

Of course, some ultimate resort must be built into the system to take over when all conflict resolutions have failed to re-establish the invariant.

3 Conclusions

There is one aspect of ConTracts that has not been discussed in this paper, because many aspects are the same as for the Saga mechanism [6]: Since updates can be externalized early, for each step (or group of steps), there must be a compensation step to semantically undo the original operations (rectangular boxes in Fig. 1a).

Implementing ConTracts as an execution environment for long-lived, consistent distributed applications requires a good deal of cooperation by the major system components. Without elaborating the details, here is a list of the most important requirements:

- The database systems must be able to act as resource managers, i.e. they must accept external commit coordination.
- Database systems must be able to understand long, recoverable locks, i.e. locks that are not bound to an ongoing transaction.
- Database systems must notify callers about synchronization conflicts.
- There must be a transactional RPC-mechanism that is able to schedule and migrate tasks and processes for requests.
- The operating system must provide relocatable processes.
- An naming service for steps, users and objects must be able to handle value-dependent roles.
- All components must support existence locks for preventing an object from being discarded while its existence is assumed elsewhere. Additionally all objects must have a global "eternal" identity.
- Logs must be able to move from one node to another in the network.

This looks like a very demanding list. But if you realize that the ConTract mechanism is nothing less than a general run-time system for reliable distributed applications, then this is exactly the set of problems that need to be addressed.

This work was supported by the *Deutsche Forschungsgemeinschaft* under contract Re 660-2/2.

References

[1] Gray, J.: *The Transaction Concept: Virtues and Limitations.* Proc. VLDB, Cannes, Sept. 1981

[2] Klein, J., Reuter, A.: *Migrating Transactions.* Proc. Workshop on the Future Trends of Distributed Computing Systems, Hong Kong, IEEE, Sept. 1988

[3] Gray, J., Reuter, A.: *Transaction Systems.* to appear 1991

[4] Reuter, A.: *ConTracts: A Means for Extending Control Beyond Transaction Boundaries* Proc. 2nd. Int. Workshop on High Performance Transaction Systems, Asilomar, Sept. 1989

[5] Davies, C.T.: *Data Processing Spheres of Control.* IBM Systems Journal, Vol. 17, No. 2, 1978

[6] Garcia-Molina, H., Salem, K.: *Sagas.* Proc. ACM SIGMOD 1987

[7] Peinl, P., Reuter, A., Sammer, H.: *High Contention in a Stock Trading Database - A Case Study.* Proc. ACM SIGMOD 1988

[8] O'Niel, P.: *The Escrow Transaction Method.* ACM Transactions on Database Systems *11*, 405 – 430 (1986)

Polytransactions for Managing Interdependent Data

Marek Rusinkiewicz
University of Houston
marek@cs.uh.edu

Amit Sheth
Bellcore
amit@ctt.bellcore.com

1 Introduction

Many large companies use multiple databases to serve the needs of various application systems. One of the significant problems in managing these databases is maintaining the consistency of inter-related data in an environment consisting of multiple semi-autonomous and heterogeneous systems. We use the term *interdependent data* to imply that two or more data items stored in different databases are related through an integrity constraint that specifies the data dependency and the consistency requirements between these data items. Management of such data implies that the requirement of mutual consistency among the interdependent data is recognized and maintained by the system. Therefore, the manipulation (including concurrent updates) of the interdependent data must be controlled to ensure that the mutual consistency of data is preserved.

With the current emphasis on data as a corporate asset, whose integrity is of basic importance, the management of interdatabase consistency is receiving more attention. Distributed transaction technologies can be used to address some of these problems. However, distributed concurrency control and commitment present serious problems when long-lived transactions span across systems with vastly different capabilities. In this paper, we propose a transaction model which may be more suitable for maintaining consistency among interdependent data stored in multiple systems. An important feature of the model is that the declarative definitions of data dependencies and the mutual consistency requirements can be used to automatically generate a set of related transactions that manage interdependent data.

The paper is organized as follows. In Section 2, we briefly discuss the specification of interdatabase dependencies for data managed by multiple, autonomous databases. This information is stored in the interdatabase dependency schema. In Section 3, we introduce the concept of polytransactions and discuss how they can be derived from transaction specifications and interdatabase dependency schema. Finally, we discuss a strategy for executing polytransactions.

2 Specification of Interdatabase Dependencies

The relationships between data items stored in multiple databases can be described by defining the data dependencies and the mutual consistency requirements [SR90, RSK91].

Data Dependency is characterized by two aspects, structure of data dependency and control. The *structural dependencies* may include full or partial replication, overlap of the informational content of the data, vertical or horizontal partitioning, value or existence constraints, etc. The *control* aspect specifies the constraints on updating the interdependent data. For example, the *derived data* may always be extracted or aggregated from one database and stored in another one which is not directly updatable. In the case of *primary-secondary* copies, the updates to the primary database are propagated to the secondary databases through a coordinator-subordinate relationship.

Mutual Consistency Criteria specify the consistency requirement among related data items. In general, the consistency requirements can be specified using three parameters: time, data states, and operations. The *immediate consistency* [SK89] requirement specifies that as soon as a transaction completes, all inter-related data are also mutually consistent. The immediate consistency can be provided by multi-site transactions that commit updates at multiple sites together. The *deferred consistency* allows specification of various levels of time-delayed consistency, where it is possible that one of the interdependent data items is up-to-date while the others may not be fully up-to-date (e.g., *eventual* or *lagging consistency* [SK89, WQ90, SR90]).

The requirements related to the data states determine "how far" the related data may be allowed to diverge before the mutual consistency must be restored [ABG88, BG90]. They may be specified by limiting the number of data items that can be changed, specifying maximum allowed change in the value of a data item, or limiting the number of operations allowed during the period the related data are inconsistent. The consistency requirements may also state that the mutual consistency should be restored before or after a specific operation is performed (or, before or after a specific event takes place).

Data Dependency Descriptors (*DDDs*) can be used to capture data dependencies and mutual consistency requirements among interdependent data items [RSK91]. A *DDD* is a triple $< D, C, A >$, where D is the interdatabase dependency specification, C defines mutual consistency requirements, and A is the action that must be performed when the consistency requirements are violated. The dependency specification defines the relationship that exists between the set of the *source* data objects and the *target* data object. The action routines can be used to specify complementary transactions that must be invoked when an udate performed on an interdependent data item violates the mutual consistency requirements.

A *Interdatabase Dependency Schema* (IDS) is a set of all *DDDs* to be enforced in a multidatabase system.

3 The Concept and Properties of Polytransactions

The fundamental requirement to define a sequence of operations as a transaction is the *correctness* of the update. We will assume that an update is correct if it does not result in a transformation of a data from a consistent state to an inconsistent state [1]. The interactions among concurrent updates and retrievals must also be correct, as determined by the semantics of the application (this involves both, transaction specification and *IDS*).

Transactions we are interested in may not have all the *ACID* properties. Depending on the application, requirements other than correctness may optionally be imposed on updates performed on interdependent data. In the case of *compensatable* operations, their effects can be "undone" by issuing compensating operations. For example, the *flexible transactions* in [ELLR90] allow specification of alternative actions that can be invoked when some actions fail. A multidatabase transaction may accomplish its objectives and complete successfully, even if some of its actions are not executed, thus relaxing the requirements of atomicity.

Sometimes, the interactions of concurrent update transactions require transaction isolation. In many traditional applications, such as banking, this requirement is a very natural one. However, this requirement becomes completely impractical in long-lived transactions where an operation may last several hours and involve multiple data items of large granularity. Enforcing isolation in such cases may mean that no concurrent activities are allowed. *Sagas* [MS87] divide long-lived transactions into subtransactions, such that each subtransaction can be compensated, if necessary, and the isolation property of the transactions is relaxed. Flexible transactions [ELLR90] may also relax isolation by allowing both compensatable and noncompensatable subtransactions within a single global transaction.

Another frequent requirement towards transactions is *durability*, which states that once the transaction is committed, its results must survive successive system failures. However, durability may not be required for transactions that do not manipulate persistent data (e.g., a flight reservation transaction may specify an expiration date, after which the reservation is withdrawn, if not confirmed.)

In a multidatabase environment consisting of multiple autonomous systems, the concept of global (multidatabase) transaction that is composed of well-defined subtransactions may be too restrictive. For example,

[1] We say that interdependent data is in an inconsistent state if the consistency criteria (including interdatabase dependencies in corresponding *DDDs*) are not satisfied.

the assumption that all subtransactions are defined at the time the global transaction is specified may be unrealistic and impossible to enforce. Therefore, we introduce a more flexible notion of a ***polytransaction*** to describe a sequence of related update activities.

A polytransaction (T^+) is a "transitive closure" of a transaction T submitted to an interdependent data management system. The transitive closure is computed with respect to the interdatabase dependency schema IDS. A polytransaction can be represented by a tree in which the nodes correspond to its component transactions and the edges define the "coupling" between the parent and children transactions. Given a transaction T, the tree representing its polytrasaction T^+ can be determined as follows. For every data dependency descriptor DDD, such that the data item updated by T is among source objects of the DDD, we create a new node corresponding to a (system generated) new transaction T' to update the target object of the DDD. T' is a child of T and updates the related data item(s) specified in the D-component of DDD.

When a user submits a transaction that updates a data item that is related to other data items through one or more DDDs, this transaction becomes the root of a polytransaction. Subsequently, the system responsible for the management of interdependent data uses the dependency schema to determine what descendent transactions should be generated and scheduledthem in order to preserve interdatabase dependency. Execution of a descendent transaction, in turn, can result in generating additional descendent transactions. This process continues until the consistency of the system is restored as specified in the IDS.

The ways by which a child transaction is related to its parent transaction within a polytransaction is specified in IDS, possibly indirectly through the consistency requirements. This relationship is indicated as a label of the edge between them in the polytransaction tree. A child transaction is ***coupled*** if the parent transaction must wait until the child transaction completes before proceeding further. It is ***decoupled*** if the parent transaction may schedule the execution of a child transaction and can proceed without waiting for the child transaction to complete.

If the dependency schema requires immediate consistency, the nested transaction model may be used, in which the descendent transactions are treated as subtransactions that must complete before the parent transaction can commit. Two-phase commit protocol may be used in this case. A coupled transaction can be *vital* in which case the parent transaction must fail if the child fails, or $non-vital$ in which case the parent transaction may survive the failure of a child [GGKKS90].

Several new transaction paradigms have been proposed recently in the literature that are based on various degrees of decoupling of the spawned activities from the creator (e.g., [KR88]). Triggers used in active databases [DHL90] are probably the best known mechanism in this group. The main problem with asynchronous triggers is that the parent transaction has no guarantee that the activity that was triggered will, in fact, complete in time to assure the consistency of the data. An early application of this idea in the management of interdependent data was discussed in [WT89]. They used a table driven approach to schedule complementary updates whenever a data item involved in a multi-system constraint was updated. The parent transaction would then terminate, without waiting for a chain of complementary actions to take place.

To allow the parent transaction some degree of control over the execution of a child transaction, the concept of a VMS *mailbox* has been generalized in [GGKKS90]. Similar ideas have been presented in [BHM90], and in [HS90], where the notion of a *persistent pipe* has been introduced. Both the generalized mailboxes and persistent pipes allow the parent transaction to send a message to a child process and know that the message will be eventually delivered. If such a guarantee is sufficient, the parent transaction may then commit, without waiting for the completion of the actions that were requested. The parent or its descendant may check later if the message has been indeed received and take a complementary or corrective action.

4 Executing Polytransactions

Most of the work on multidatabase transaction management assume the existence of a *multidatabase management system* (MDBS) which is responsible for the processing of all global transactions. Such MDBS architectures have transactions that are either local (execute at a single database without passing through the MDBS interfaces) or are global [CR90]. We propose an architecture that uses a dependency schema at each site that stores the DDDs involving the interdependent data that can be updated at that node. Using the concept of polytransactions, many of the transactions that are global transactions in the traditional architecture can be performed as a collection of related single database transactions.

Two approaches to control the execution of multidatabase transactions have been discussed in the literature. Under the first approach, the MDBS controls the scheduling of all subtransactions of a transaction. A disadvantage of this approach is that the set of all subtransactions and the precedence dependencies between them must be known in advance. The second approach is used in active databases and uses triggers to asynchronously schedule subtransactions based on some events, usually in a decoupled fashion [DHL90]. This approach involves procedural specification of triggers and is event driven.

We propose to schedule the polytransaction activities based on the information in the dependency schema and the database states. This approach allows the transaction schedule to be determined dynamically based on the information stored declaratively in the dependency schema. Unlike triggers, this approach is state driven and not event driven. We will illustrate this approach using an example.

Example: Consider a collection of telecommunication databases used by applications[2] for planning and establishing new services. Let us consider four databases as follows:

- $DB1$ contains information about each switch (an equipment that establishes circuits and routes telephone calls) and its contents (e.g., the equipment each of its slots contains).
- $DB2$ contains summary information about the equipments used in different switches for use by a statistical application.
- $DB3$ is an operational database containing status information about each switch.
- $DB4$ contains planning information about the switches whose capacities are close to being exhausted.

Now consider the transaction $T1$ submitted to $DB1$ that modifies the status of one of the slots in the switch as a result of installing new equipment in an available slot. Let us suppose that the interdatabase dependency specifies that database $DB2$ should be eventually updated to reflect the changes of the status of each of the switches. Hence, transaction $T2$ will be scheduled to make required changes in $DB2$. Due to the eventual consistency requirement of data between the data updated by $T1$ in $DB1$ and the interdependent data in $DB2$, $T2$ can be a decoupled transaction. Let us further suppose that $DB3$ must be updated immediately to reflect the change in the status of the switch. Therefore a transaction $T3$ must be scheduled. Because of the immediate consistency requirement between the data $T1$ modifies in $DB1$ and the interdependent data in $DB3$, $T3$ should be a coupled transaction and $T1$ cannot commit until $T3$ does. To continue our example, if the change of the status of the switch (as stored in $DB3$) brings its capacity above a threshold specified in the dependency schema, transaction $T4$ will be scheduled to add the relevant switch information to $DB4$. If there is a lagging consistency requirement between the data in $DB3$ that is modified by $T3$ and the interdependent data in $DB4$, transaction $T3$ can terminate before $T4$ is executed. When all transactions resulting from $T1$ complete, the polytransaction completes.

Summary

Managing the consistency of interdependent data in a multidatabase environment is one of critical data management issues. We introduced the concept of database dependency descriptor, that can be used to specify the relationships between data in multiple databases together with the mutual consistency requirements. These descriptors constitute the Interdatabase Dependency Schema. The information stored in the dependency schema can be used to convert a transaction updating a data item in a single database into

[2]No implication about real applications in Bellcore or its clients is intended.

a *polytransaction* that spawns various activities needed to maintain the consistency of the interdependent data. The activities constituting a polytransaction, can be either coupled to the parent transaction, or decoupled from it. In the latter case, we can guarantee better response times if the weaker consistency guarantees (e.g., lagging consistency) are sufficient in a given application. We believe that the polytransaction paradigm provides an attractive alternative to the traditional multitransaction models, since it provides the flexibility needed to support complex interdatabase consistency requirements. The concepts presented in this paper are preliminary and we are currently investigating their applicability.

References

[ABG88] R. Alonso, D. Barbara, and H. Garcia-Molina. Quasi-copies: Efficient Data Sharing for Information Retrieval Systems. Proceedings of the International Conference on Extending Database Technology, 1988.

[SK89] A. Sheth and P. Krishnamurthy. Redundant Data Management in Bellcore and BCC databases. Bellcore Technical Memorandum TM-STS-015011/1, December 1989.

[CR90] Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. SIGMOD, 1990.

[SR90] A. Sheth and M. Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. Proceedings of the Workshop on the Management of Replicated Data, Houston, TX, November 1990.

[BHM90] P. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. SIGMOD, 1990.

[RSK91] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying Interdatabase Dependencies in Multidatabase Systems. Bellcore Technical Report, February 1991.

[WQ90] Gio Wiederhold and Xiaolei Qian. Consistency Control of Replicated Data in Federated Databases. In *Proc. of 1st IEEE Workshop on Management of Replicated Data*, Houston, November 1990.

[BG90] Daniel Barbara and Hector Garcia-Molina. The Case for Controlled Inconsistency in Replicated Data (Position Paper). In *Proc. of 1st IEEE Workshop on Management of Replicated Data*, Houston, November 1990.

[DHL90] U. Dayal, M. Hsu and R. Ladin Organizing Long-Running Activities with Triggers and Transactions. SIGMOD, 1990.

[GGKKS90] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-Transaction Activities. Technical Report CS-TR-247-90, Dept. of Computer Sc., Princeton University, February 1990.

[SLE91] A. Sheth, Y. Leu, and A. Elmagarmid. Maintaining Consistency of Interdependent Data in Multidatabase Systems. Technical Report, Dept. of Computer Sc., Purdue University, January 1991.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[WT89] R-J. Wang and G. Thomas. Personal Communication. February 1989.

[HS90] M. Hsu and A. Silberschatz. Persistent Transmission and Unilateral Commit- A Position Paper. Presented at *Workshop on Multidatabases and Semantic Interoperability*, Tulsa, OK, October 1990.

[MS87] G. Molina and K. Salem, Sagas. In Proc. of ACM SIGMOD, May 1987.

[KR88] J. Klein and A. Reuter. Migrating Transactions. In *Future Trends in Distributed Computing Systems in the 90's*, 1988.

A Flexible and Adaptable Tool Kit Approach for Transaction Management in Non Standard Database Systems*

Rainer Unland, Gunter Schlageter

University of Hagen, Department of Computer Science I
P.O. Box 940, D-5800 Hagen 1, Germany
EMail: unland@dhafeu51.bitnet

1. Introduction

The traditional approach to transaction processing works very well as long as transactions are rather short and simple. However, it falls short of meeting the much more sophisticated demands of advanced database applications, as has been stated in numerous papers (see, e. g., /Katz84/, /KLMP84/, /KoKB85/, /KSUW85/, /UnSc89/). Some buzzwords in this context are long-duration, interactive transactions, synergistic cooperative work, and application or user supported consistency. A first answer to these requirements was an extension of the traditional concept of transactions in the way that the otherwise flat transaction model is allowed to include transactions within transactions. This kind of transaction is called **nested transaction.** In the commonly known approach of Moss (/Moss82/) a nested transaction recursively consists of a set of children that execute atomically with respect to their parents and their siblings. The main achievements of this approach are the support of **modularity** and **failure handling** since the nesting allows the user to structure and delegate his work and to define more graceful units of recovery (namely subtransactions), and a higher degree of **parallelism** since subtransactions can be executed concurrently. The nested transaction model is the fundamental basis of all 'advanced transaction models' which are proposed in literature. The difference between the various models lies in the number and meaning of the constraints and rules which they place on the way how (nested) transactions have to look like and how they have to interact with each other. Prominent examples of these constraints and rules are:

* Most approaches provide different transaction types. However, these types can only be *nested* in a *special, predefined order.* For example, /KoKB85/ present a model in which a design transaction consists of a number of project transactions each of which consists of a set of cooperating transactions, and so on. /KSUW85/ define a database transaction to be the basis for a set of user transactions. /UnSc89/ add to this model group transactions.
* Some proposals require *all* transactions to run a *strict* two-phase lock protocol (locks can not be released before end-of-transaction (EOT)), e. g., Moss (/Moss82/) or Katz (/Katz84/).
* Almost all proposals restrict children to *commit objects* (or locks) to their *parents* only, e. g., /KLMP84/, /KoKB85/, /KSUW85/, /Moss82/, /UnSc89/.
* Some approaches, e. g., /KLMP84/, /KoKB85/ or /UnSc89/, only allow *leaf transactions* to *perform operations* on data. *Inner transactions* only serve as a kind of *database* for their children.

Most of these rules and constraints reflect the individual view of the authors on the requirements and conditions of the application area which is supposed to be the target class of the respective transaction model. Consequently, the transaction model may be suitable for a number of application areas, whereas it may be inappropriate for others.

Another observation is that a large number of approaches to advanced transaction modeling concentrate first of all on modularity, failure handling and parallelism while a support of cooperative work is subordinated to serializability. Mostly, the assumption is still made that (sub)transactions are competitors rather than partners. However, appropriate support of cooperative work can only be achieved if the still predominant rigid concurrency control measures (strict isolation) are weakened, e. g., by moving some responsibility for the integrity of the data from the database management system to the application. Of course, this has to be done in a controlled and application-specific way and the database management system has to offer as much help as possible.

Finally, it has to be considered that the field of new application areas is diversified to a large extent. Applications may differ substantially in their demands and even in their understanding of consistent operations on data. This has led to a number of different, sometimes even contradictory demands on transaction management.

* This work has been supported by the German Research Community (DFG) under Contract No. Schl 209/2-1

At the point of time when we started our project we had several basic requirements in mind which we wanted to be fulfilled by our approach. The most important are that the approach should not be suited to a special application area but should be application independent, and that it should be capable of intensively supporting synergistic cooperative work. However, as the above discussion already indicates, there are a lot of serious arguments which strongly suggest that *one* given transaction manager can only be a more or less satisfactory compromise. Therefore, we came to the conclusion that the most promising solution is a concept very similar to the tool kit approach of database systems (e. g., EXODUS (/CaDe87/)): a tool kit for transaction management. This tool kit is meant to be part of the general tool kit (or erector set of modules) of the database (kernel) system; i. e., it serves a sophisticated applications designer or database implementor (DBI) to model 'his' application-specific transaction manager in an appropriate and natural way.

An application-specific transaction manager is supposed to provide the application with a number of transaction types, which are especially suited to the requirements of the application. Transaction types may differ, e. g., in their structure, their behaviour or the way in which they fulfill their task. Of course, we want these different transaction types to be usable in any order within a nested transaction to form a heterogeneously structured transaction tree which is capable of supporting such different concepts as strict isolation of (sub)transactions (in the sense of serializability) and non-serializable cooperative work in one hierarchy. Therefore, we need to define some general rules which have to be obeyed by each transaction type.

The remainder of this paper is organized as follows. The general rules of our approach are presented in section 2. The characteristics which make up different transaction types are discussed in section 3. Section 4 gives a brief overview of the structure of the tool kit. Due to space restrictions we can only outline the most salient features of our approach. A more comprehensive discussion can be found in /Unla90-1/ (lock modes), /Unla90-2/ (transaction model) and /Unla91/ (general concepts of the tool kit and implementation concepts). Moreover, it should be acknowledged that part of this work profited from the fundamental discussion of properties of nested transactions in /HäRo87/.

Before we discuss the general rules of the approach some simplifications are to be clarified first:

* The current version of the tool kit concentrates on locks as the means for concurrency control.
* The following discussion focuses on long duration transactions. Other transaction types (e. g., conventional transactions) are not considered.
* For reasons of simplicity we assume that long duration transactions maintain their own object pool (an **object pool** is a (logical) container for the set of objects which are associated with the transaction at a given point of time).

2. Fundamental rules of the tool kit approach

In order to support the essential semantics involved in advanced database applications the serializability-based transaction models have to be replaced by models which make it possible to express semantics beyond serializability; e. g., a transaction manager which asks for human or applications involvement to ensure correctness of the system as a whole. On the basis of the locking approach this goal can, in principle, be achieved in two different ways:

1. By using lock protocols which allow transactions to exchange or release data at an earlier point of time.
2. By offering lock modes which facilitate a higher degree of concurrent work on data; i. e., which allow the concurrency control component to exploit application-specific semantics.

A first step in the direction of alternative 1 would be to run a simple two-phase lock protocol instead of a strict one. This means that a transaction consists of a growing phase in which objects can be acquired and a following shrinking phase in which objects can be released. However, an early release of objects has to be handled with care since it can violate the principles of two-phase and isolation if a transaction acquires an object from an ancestor different from its parent without considering the status of the transactions on the path to the ancestor (for more details, see /Unla90-2/). To avoid these problems and to facilitate the installation of different concurrency control schemes for different transaction types the rule of stepwise transfer was introduced.

Stepwise transfer

The general principle of the stepwise transfer is that a transaction T can directly acquire objects from its parent only. However, if T needs an object O from some other ancestor T^A this is realized by a stepwise check-out of O from the object pool of T^A via the transactions on the path to T (successions of downward check-outs). If, for example, T12 wants to acquire an object O from the parent of T3 such a demand is satisfied by a stepwise check-out of O from the parent of T3 to T3 to T5 to T12. On each level the concurrency control scheme of that level (transaction) is employed to safely transfer O to its destination.

In a similar way we define a stepwise check-in of an object O. This means, that we do not require a transaction T to pass O to its parent. T may transmit O to some superior T^S if the status (lock protocol) of each transaction T^P on the path to T^S allows such a proceeding.

Besides the stepwise transfer the two-stage control-sphere is another fundamental concept of our approach.

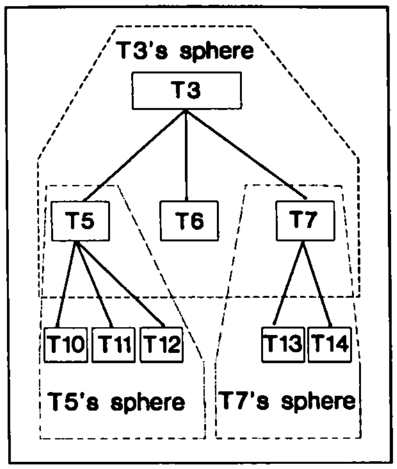


Figure 2.1: Two-stage control-sphere

Two-stage control-sphere

The underlying principle of the **two-stage control-sphere** is that a parent is only responsible for the correct coordination and execution of the work (task) on **its** level. It may define subtasks and start children to deal with these subtasks. Each child, again, is by itself responsible for the correct coordination and execution of its task and, therefore, can decide autonomously, how this task can be executed best. In other words, the characteristics which were established on the level of the parent are only valid for its children. The children, in turn, may establish a different environment for their children (see fig. 2.1). The two-stage control-sphere establishes the foundation for the possibility to execute transactions of different types within one transaction hierarchy.

The two-stage control-sphere requires that the acquisition (release) of an object from (to) a superior (other than the parent) is realized by a stepwise transfer. However, whether an object is accessible to a transaction depends on the states of the transactions which are affected by the stepwise transfer. In general, the object pools which are accessible to a transaction are described by the access (release) view of a transaction.

Access and release view

The set of objects which are accessible to a transaction T is described by the access view of T. The **access view** consists of all objects of the object pools of the chain of ancestors (inclusively the public database) up to the first ancestor which runs a two-phase lock protocol and is already in its shrinking phase. The access view changes dynamically during the lifetime of T (more precisely: it decreases) since, at any point of time, an ancestor may start its shrinking phase.

In figure 3.1 the access view of T38 consists of the objects of the object pools of T38, T26, T15, T8, T4, T1, and the database, if T8, T4 and T1 are not in their shrinking phase (**extended predeclaring** means that objects can be requested as long as no lock was released. If the requested objects are not locked in an incompatible mode they are granted. Otherwise, the lock request is rejected but the transaction is not blocked).

Similar to the notion of access view we define the notion of release view. A **release view** defines up to which object pool an object of a given transaction T can be released at most (if no other lock on the object prevents this). The release view consists of all object pools of the chain of ancestors (inclusively the public database) up to the first ancestor which runs a two-phase lock protocol and is still in its growing phase.

3. Characteristics of transaction types

In this section the various characteristics which make up a transaction type in the tool kit approach will be discussed. These characteristics can be subdivided into two parts: characteristics which describe the physical structure of a type and characteristics which describe its behaviour and semantics.

Structure of transaction types

As already mentioned, we want the tool kit to provide a wide range of different transaction types. This set is meant to include conventional (short duration) transactions as well as all kinds of long duration transactions or compensating transactions. This requires that transaction types are made up of different components. For example, a long duration transaction may maintain its own object pool and lock table for this object pool. Or, a transaction type may run an optimistic concurrency control scheme instead of a pessimistic one. Moreover, since the tool kit supports a large number of fine grained lock modes we allow a transaction type to maintain its own compatibility matrix for its object pool. By this, the access to the objects of the object pool can be individually suited to the requirements of a specific environment (more or less restrictive).

Concurrency control scheme

The two stage control-sphere stands for the possibility that each transaction type T can establish its own concurrency control scheme. This means, that each T can independently determine according to which rules descendants of T have to acquire objects from T. Such a free choice of concurrency control scheme is possible since the stepwise transfer of objects guarantees that each transaction which is involved in the stepwise transfer will use the concurrency control scheme which is required by its parent. For example, in figure 2.1, if transaction T3 wants locks to be used to synchronize access to its object pool its children T5, T6, and T7 need to run a lock protocol if they want to acquire objects from T3. However, each of the children may apply its own type of lock protocol, e. g., T5 may run two-phase locking with predeclaring, T6 simple two-phase locking, and T7 strict two-phase locking. On the other hand, each child may run a different concurrency control scheme for its own object pool. T5, for instance, may run optimistic concurrency control (OCC) with the consequence that the children of T5 (T10, T11, T12) have to run an optimistic concurrency control scheme if they want to acquire objects from T5. For example, if T12 wants to acquire an object O from T3 the stepwise transfer ensures that O is first transferred from the object pool of T3 to the object pool of T5 by using a lock protocol and then from the object pool of T5 to the object pool of T12 by using OCC.

Lock modes

One major reason why traditional transaction management fails in the context of advanced database applications is that it can not consider the application-specific semantics of operations. If such semantics were exploited, the concurrency control scheme would be able to provide higher concurrency than by simply looking at the operations as reads and writes (using only exclusive and shared locks). The tool kit approach indicates a possible solution for the inclusion of such application-specific semantics since it provides a rich set of fine grained lock modes which can individually be adapted to the semantics of the operations of a given application. Moreover, the tool kit does not only allow to link locks temporarily to transactions (as in the classical case) but also temporarily to users / applications, and permanently to objects. Due to space restrictions we can't discuss the different lock modes in more detail here. The interested reader is refered to /Unla90-1/.

Task

Some approaches to nested transactions require work on objects to be exclusively performed in the leaf transactions of the transaction tree. Inner transactions only serve as a kind of database for their children. However, in many applications it is desirable that an inner transactions T can also perform operations on its objects. Of course, in such a case the work of T on its objects has to be synchronized with the work of T's children on these objects. The tool kit allows the DBI to define a transaction to be of type **service transaction** (transaction which only serve as a database for its children) or **operational transaction** (transaction which is, additionally, allowed to operate on its objects) independently of the position of the transaction in the transaction tree.

Parallelism

A transaction T must define whether its children (inclusively T itself) can execute concurrently. If concurrent execution is prohibited no synchronization measures are established on the level of the object pool of T (since no concurrency is possible). Again this characteristic is only valid on the level of T and the children of T. A child may decide to allow its children to execute concurrently.

Explicit Cooperation

The support of cooperation is a mandatory feature, especially in design environments. Our approach supports the possibility to explicitly install a direct cooperation between two or more transactions from different branches of the transaction tree (however, certain conditions must be fulfilled). Transactions which are involved in a cooperation may directly lend, transfer, or exchange objects among themselves.

Recovery

To exploit the advantages of nested transactions, recovery has to be refined and adjusted to the demands of the control structure of nested transactions. Moreover, it must be suitable to fulfill the requirements of the various applications. Due to space limitations and the inherent complexity of recovery we don't want to discuss this topic here. The interested reader is referred to /Unla91/.

Figure 3.1 gives an example of a heterogeneously structured transaction tree (in which all transactions run a lock protocol). Higher levels of the transaction hierarchy use stricter types of lock protocols (e. g., the first three

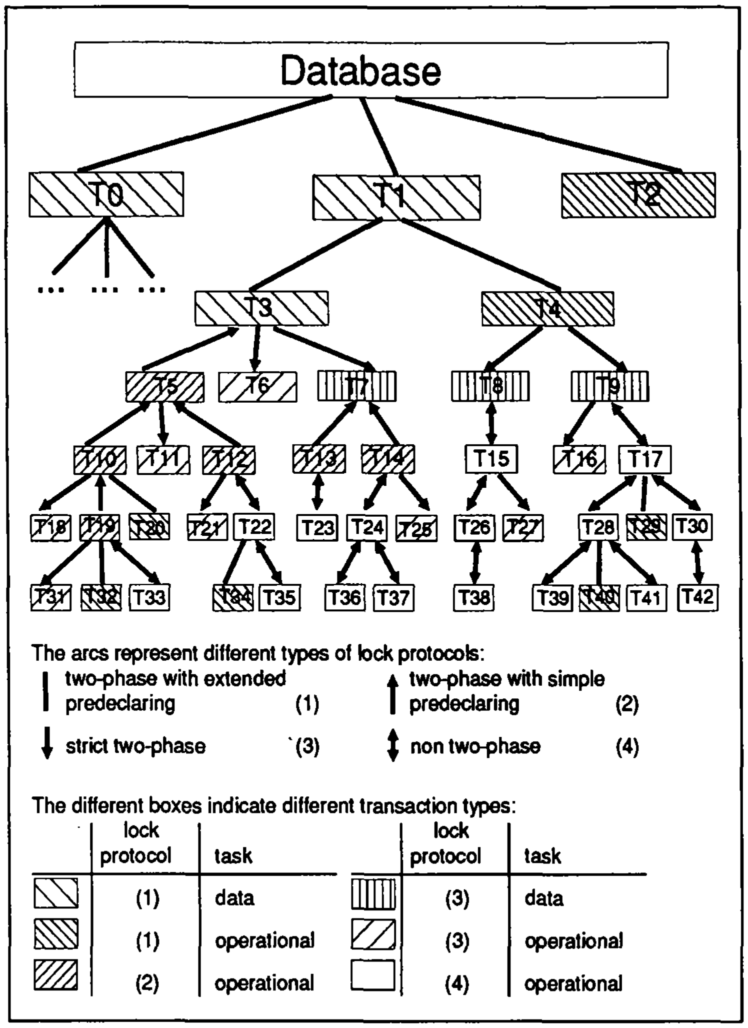


Figure 3.1: Heterogeneously structured transaction

levels of the nested transaction T1) while lower levels partly provide a more cooperative environment (since they use the non two-phase lock protocol; for a discussion of different lock protocols see e. g. /BeHG87/).

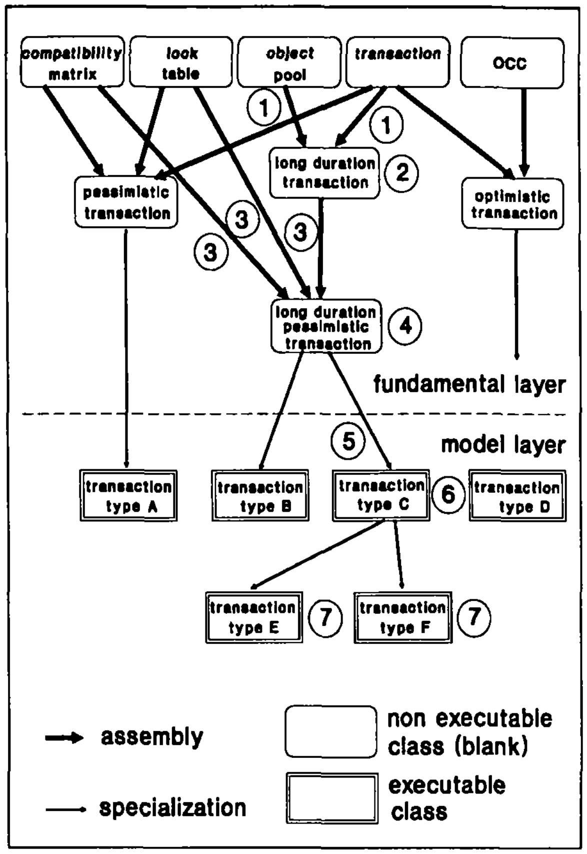


Figure 4.1: Structure of the tool kit

4. Brief overview of the structure of the tool kit

The tool kit can be regarded as a kind of object-oriented transaction manager development facility for the following reasons:

* Each component of the tool kit belongs to a class; each class represents a different type of component.
* Components are realized as abstract data types. This means in particular that transaction types are characterized by the operations which come with them. A number of operations are common to all transaction types; e. g., operations to suspend, continue and commit the transaction and to acquire or release objects. However, these operations may be implemented differently for different transaction types (with different semantics or with different implementation part (data structure)). Other operations are only specific to some transaction types since they come with the features by which these transaction types differ from other transaction types. Note, that the concept of abstract data type makes it possible to easily react to changing requirements. If, e. g., in a workstation / server environment the maintenance of an object pool is to be moved from the server to the workstation this can easily be realized due to the locality of such changes. Moreover, this concept leaves some leeway for the implementation of logical structures; e. g., while the logical structure of a transaction type may require a local object pool the implementation may lean on a global object pool (one pool for all transactions).
* The assembly and refinement of a transaction type is realized via (multiple) inheritance.

A transaction type is developed in the following way: First, the basic constituents are chosen from the set of basic components of the tool kit. These constituents are suited to each other to form a kind of **blank** of a first basic transaction type. For example, in figure 4.1 a long duration transaction (2) is assembled from the basic components object pool and transaction (at this level a transaction is nothing more than a frame within which operations can be executed) (1). A blank can be specialized to more specific blanks by integrating additional components. In figure 4.1, the long duration transaction is specialized to a long duration pessimistic transaction (transaction which runs a lock protocol, (4)) by adding a compatibility matrix and a lock table (3). A blank corresponds to a general transaction type which is not yet executable since it does not realize the specific semantics of any given transaction model. For example, a blank already provides a check-out operation. However, the rule that objects can only be acquired from an ancestor is not yet laid down. This kind of semantics is added in the next step in which blanks are equipped with semantics (5) to constitute executable transaction type (6) which are especially adapted to a specific transaction model. Of course, by adding further rules or constraints executable transaction types can b specialized to more specific transaction types (7). The basic components and the blanks make up the **fundamental layer** of the tool kit while the executable transaction types constitute the **model layer**.

Altogether the tool kit consists of a number of basic components, a set of blanks and a set of executable transaction types. An application specific transaction manager consists of a subset of the executable transaction types. In case the tool kit does not provide all transaction types of interest the missing types have to be added; i. e., the tool kit has to be extended. This can be achieved in several ways: The fundamental layer can be augmented either by new basic components (e. g., a new concurrency control scheme) or by newly constructed blanks. The model layer can be augmented by newly constructed executable transaction types (e. g., more specialized ones). If a different global transaction model is to be installed (e. g., a model which allows a nested transaction to be netlike instead of treelike) a new model layer has to be developed; i. e., the blanks have to be equiped with different semantics.

A first prototype of the tool kit, which was implemented on top of the relational DBMS ORACLE, is currently in its test phase. This prototype serves as a testbed for the investigation of the weakness and strong points of our approach. We intend, as a second step, to integrate a revised version of the tool kit, which reflects the experiences with the prototype, into a database kernel system.

Literature

/BeHG87/ Bernstein, P.; Hadzilacos, V.; Goodman, N.: *Concurrency Control and Recovery in Database Systems;* Addison-Wesley Publishing Company; 1987

/CaDe87/ Carey, M.J.; DeWitt, D.J.: *An Overview of the EXODUS Project*, IEEE Database Engineering; Vol. 10, No. 2; June 1987

/HäRo87/ Härder, Th.; Rothermel, K.: *Concepts for Transaction Recovery in Nested Transactions;* Proc. ACM-SIGMOD; San Francisco, California; 1987

/Katz84/ Katz, R.H.: *Transaction Management in the Design Environment*, in: 'New Applications of Data Bases'; G. Gardarin, E. Gelenbe (Ed.); Academic Press; 1984

/KLMP84/ Kim, W.; Lorie, R.; McNabb, D.; Plouffe, W.: *Nested Transactions for Engineering Design Databases*; Proc. 10th Int. Conf. on VLDB; Singapore; Aug. 1984

/KoKB85/ Korth, H.F.; Kim, W.; Bancilhon, F.: *A Model of CAD Transactions*; Proc. 11th Int. Conf. on VLDB; Stockholm, Sweden; 1985

/KSUW85/ Klahold, P.; Schlageter, G.; Wilkes, W.; Unland, R.; *A Transaction Model Supporting Complex Applications in Integrated Information Systems*; Proc. ACM-SIGMOD; Austin, Texas; 1985

/Moss82/ Moss, J.E.B.: *Nested Transactions and Reliable Distributed Computing;* Proc. 2nd Symp. on Reliability of Distributed Software and Database Systems; Pittsburgh, PA; July 1982

/Unla90-1/ Unland, R.: *A Flexible and Adaptable Tool Kit Approach for Concurrency Control in Non Standard Database Systems;* Proc. 3rd Int. Conf. on Database Theory (ICDT); Paris, France; Dec. 1990

/Unla90-2/ Unland, R.: *Transaction Types for Non-Standard Database Systems;* Research-Report (submitted for publication); University of Hagen; Dec. 1990

/Unla91/ Unland, R.: *A Flexible and Adaptable Tool Kit Approach for Transaction Management in Non Standard Database Systems;* Research-Report (in preparation); University of Hagen; 1991

/UnSc89/ Unland, R.: *A Multi-Level Transaction Model for Engineering Applications*; Proc. 1st Int. Symposium on Database Systems for Advanced Applications; Seoul, South-Korea; April 1989

The S-transaction Model

Jari Veijalainen
Technical Research Centre of Finland
Laboratory for Information Processing
Lehtisaarentie 2A, SF-00340 Helsinki, Finland
veijalainen@tik.vtt.fi

Frank Eliassen
University of Tromsø
Department of Computer Science
N-9001 Tromsø, Norway
frank@cs.uit.no

Abstract

The database transaction model and its implementations were originally developed for the business environment where atomicity, consistency, isolation, and durability (ACID) are important properties. The semantic transaction model adopts such transactions as the basic building blocks, combining them, the "higher level" computation, and control flow, into distributed hierarchical transactions, s-transactions. Isolation among different s-transactions is abandoned and the traditional atomicity is replaced by a probabilistic one. S-transactions are dynamically generated, they preserve local consistency and are exactly as durable as usual transactions they are based on. In s-transactions, the local transactions are the concurrency control units and any sub-s-transaction can be dynamically chosen to become a recovery unit.

1. Introduction

The s-transaction model discussed here was initially developed in a European project[1] during 1985-1988. The environment was S.W.I.F.T. II network, that is to replace the current S.W.I.F.T. I network. The latter has been operable in the international banking environment since the late seventies, being one of the first large distributed systems supporting what is currently known as Electronic Data Interchange (EDI). Conceptually, such systems are based on a small set of standardized *message types*, whose instances are shipped from site to site through a communication network.

The practical goal of the project was to develop mechanisms that would help to associate the processing of different messages together and to offer multidatabase services through an open network [MAP86]. *Autonomy* was recognized as an important subject in such an environment [Eli87], [Vei88], and its preservation is pervasive in the s-transaction model. Technically, the initial ideas for the model came and from the (hierarchical) transaction models, notably those in [Gray81], [Banc85] and [Moss85], from the notion of objects, and multidatabases [Lit86]. The implementation of the model constitutes the kernel of the developed prototype multidatabase system called MUSE [Ell88]. A thorough analysis of autonomy in distributed system design and the rationale for the s-transaction model can be found in [Vei90].

2. Banking environment and its requirements

The international banking environment served by S.W.I.F.T. consists of over 2000 *organisationally autonomous (O-autonomous)* banks connected to the network, meaning that no bank is *controlled* by another bank. Indeed, banks might cooperate and compete with each other at the same time. This implies that a cooperative system can only have such functionality that the interests of a participant bank cannot be not harmed. O-autonomous banks want to decide themselves what kind of data processing facilities they are going to use and who gets access to them, i.e. they are *design autonomous (D-autonomous)* with respect to their computer systems. D-autonomy implies that the environment is definitively *heterogeneous* in all respects (different local hardware and software, database management systems, database schemas, and different consistency constraints embedded in application programs). For international banking, this situation is likely to be a permanent state of affairs, i.e. banks will express *effective D-autonomy* by constantly developing their systems as they like. The s-transaction model must consequently cope explicitly with heterogeneity in order to be applicable for a longer period of time.

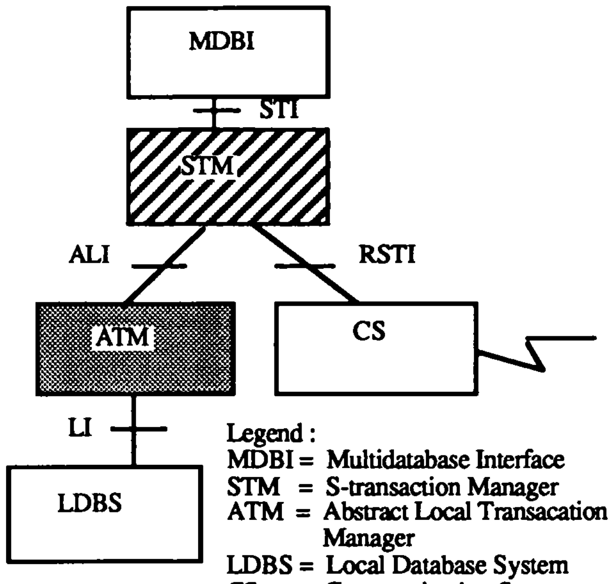
Banking systems typically should have *communication autonomy (C-autonomy)*, i.e. they should have a possibility to decide when to communicate through the network and with whom. A direct communication between two banks

[1] Multidatabase Services on ISO/OSI Networks for Transnational Accounting (MUSE), performed by I.N.R.I.A.(France), Gesellschaft für Mathematik und Datenverarbeitung mbH (Germany), Society for Worldwide Interbank Financial Telecommunications s.c.(Belgium), and University of Dortmund (Germany), and financed partly by the Commission of the European Communities under contracts MAP761 and MAP761B.

in different time zones is impossible during working hours, unless the network stores the messages and forwards them when the receiver is willing to accept them. Such a network supports the *effective C-autonomy* of the banking systems and it is typical in cooperative environments.

Another consequence of O-autonomy is that an organisation does not need to execute all messages it receives, i.e. a banking system has *execution autonomy (E-autonomy)* with respect to the corresponding message types. Thus, it can refuse the service for variety of reasons, like mistrust of the business partner, erroneous information in the message, authorization failure - or because the message does not always require further actions (e.g. an offer to buy or sell currency).

Effective E-autonomy means making use of the possibility not to execute a message. Its implications become clearer when considering e.g. the 2PC protocol [Gray78]; if a participant system really refuses to execute COMMIT or ABORT message sent by the Coordinator, the protocol does not meet its goals. The 2PC protocol is, in this environment, useless also because the s-transactions can last hours or even weeks. Blocking pertinent data of a bank is unacceptable for a long time [Vei88]. Further, a bank involved in an s-transaction decides during execution, which other banks it would like to involve in the s-transaction, and so on. In response to these requirements, the *s-transaction protocol,* supports *long lived, dynamically generated, tree-structured* executions, called s-transactions (see Fig. 2).



Legend :

MDBI =	Multidatabase Interface
STM =	S-transaction Manager
ATM =	Abstract Local Transacation Manager
LDBS =	Local Database System
CS =	Communication Server
LI =	Local Interface
RSTI =	Remote S-transaction Interface
STI =	S-Transaction Interface

Fig. 1 The site architecture

3. An architecture supporting s-transactions

The software architecture supporting s-transactions (Fig.1) preserves maximally all autonomy aspects of a banking system. Only the local database system (LDBS) is a pre-existing module, other modules are added to support the cooperation of organizations - without modifying the existing database system software or applications. The s-transaction Managers (STM) are able to interpret/compile the system-wide homogeneous, Turing-equivalent *transaction language.* A particular *s-transaction definition* is collectively designed by the *transaction designer.* The agreed upon definition is represented as a finite set of programs $P_1,...,P_n$, called *program parts.* The computation, parameters, and the calling relationships between programs are all embedded in the set of program parts.

The *global program parts* are programs represented with the transaction language. The *abstract local transactions (ablocal transactions)* are the smallest, indivisible global program parts. Their interface is of form P_i(IN;OUT), where IN is the (relational) *input parameter schema* and OUT is the (relational) *output parameter schema.* Ablocal transactions are hosted by the *Ablocal Transaction Manager (ATM)* modules. Their interfaces, taken together, form the *Abstract Local Interface (ALI)* at a site. All other global program parts refer directly or indirectly to ablocal transactions, which are the only global gate into/from the LDBS. ALI thus hides all aspects of heterogeneity.

An ablocal transaction P_i can be seen as a saga [Gar87] and ATM as a saga manager [Vei90]. Thus, P_i refers to a *positive local transaction program* P_{i+}, and possibly to a *compensating local transaction program* P_{i-}. P_{i+} and P_{i-} form the *local program parts* of the s-transaction definition and only they access the local database. To ensure the local correctness, both of them are programmed by the D-autonomous organisation controlling the LDBS. The programs are permanently stored at the local database system, or are part of the database management system, like an SQL-interpreter. P_{i+} and P_{i-}, if it exists, *implement* P_i(IN;OUT), i.e. they give the operational semantics for the ablocal transaction P_i.

The global program parts P_j, that are *not* hosted by any ATM, are real programs written in the transaction language. They are stored permanently at STMs or shipped dynamically to them, through the communicating Communication Servers (CS) from other sites, or through the multidatabase interface from the same site (in case of query processing). In order to cope with the effective C-autonomy of sites, CS modules are based on X.400 protocols [X.400].

S-transaction in the narrower sense is a *distributed execution* of an s-transaction definition. It always begins at the *root program* started by a user. Root program possibly requests execution of some ablocal transactions at the same site and remote program parts, *sub-s-transactions*, at *other* sites. The latter ones can again request further ablocal transactions and remote program executions, and so on. This invocation relationship between the *superior* and the sub-s-transactions is modeled through the *spanning tree* of the s-transaction. It is rooted at the root program execution and it contains all global, ablocal and local transactions invoked (including the compensating ones). Note that several program parts P_j, P_k,... might semantically be equivalent with each other, which indeed makes it possible to choose dynamically the execution site of a sub-s-transaction.

Operationally, a sub-s-transaction is invoked by sending an s-transaction protocol message carrying the actual input parameters in_i. and the name of the global program part P_i. The response carrying the output parameters out_i is communicated from the sub-s-transaction to the superior. The response can be "positive" or "negative". In the latter case the sub-s-transaction has abolished all its results, if any, and in the former case it might have already *committed* its results at the local databases that it did manipulate. The results of a sub-s-transaction remain valid, unless a superior decides to force an abolishment of the results. Since the abolishment is always possible, the superior must always inform the sub-s-transaction of the ultimate decision either by an ABORT-message (abolish) or by an ACK message (run to end). Only the root transaction can decide that the whole s-transaction is definitively successful, but any sub-s-transaction is entitled to decide that it itself is definitively not successful. This is different from e.g. nested transactions [Moss85]. In particular, each local trans–action might either commit or abort its results in the local database, before issuing the response to the ATM. This expresses the effective E-autonomy. Consequently, when the root-s-transaction decides on the fate of the whole s-transaction, the results are often already released for other s-transactions and local transactions (local users). In this way the global isolation is abandoned, although the local one can at the same time be kept.

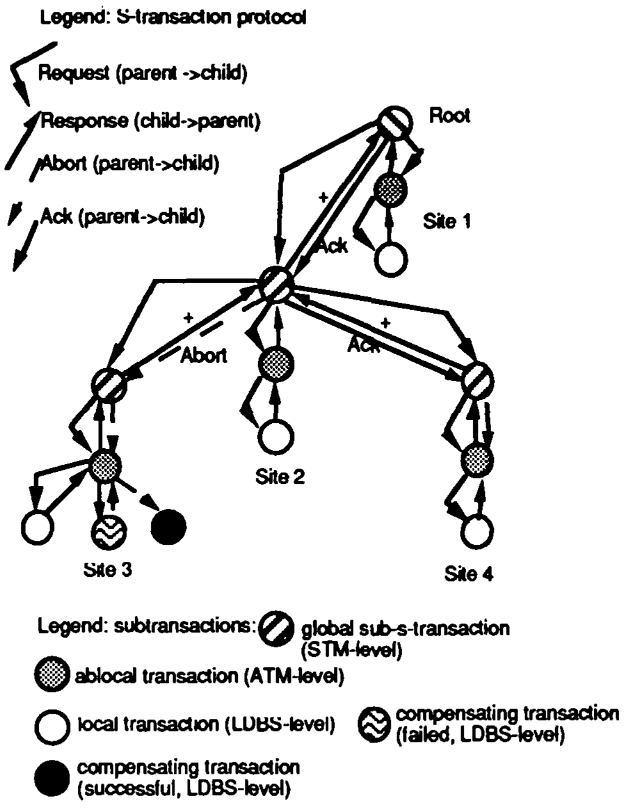

Legend: S-transaction protocol
Request (parent ->child)
Response (child->parent)
Abort (parent->child)
Ack (parent->child)
Root
Site 1
Ack
+
Abort
Ack
+
+
Site 2
Site 3
Site 4
Legend: subtransactions:
global sub-s-transaction (STM-level)
ablocal transaction (ATM-level)
local transaction (LDBS-level)
compensating transaction (failed, LDBS-level)
compensating transaction (successful, LDBS-level)

Fig 2. An s-transaction spanning four sites

Abandoning global isolation is reasonable if the purpose of the databases involved is for example to record purchase orders, or any kind of reservations (flight, hotel, etc.) that by definition can be canceled. There is no reason to hide the reservations or orders from the other transactions (i.e. from the local staff), but rather make them visible as quickly as possible, since they must have an immediate impact within the organization.

In Fig. 2 we have an example of an s-transaction having sub-s-transactions at four sites. Let site 1 be a travel agency ordering flights for a customer from city A to city C, via city B. Site 2 is an airline that only can offer a flight from A to B. Site 2 records it in its local database (the positive local transaction, white circle) and asks site 3 (another airline) for a connecting flight from B to C (a request from site 2 to site 3). Site 3 can offer a late flight which it records a reservation for (local transaction) and responds to site 2 positively (thin arrow upwards). The flight is too late for the customer, and so site 2 asks site 4 (yet another airline) for a reservation (request from site 2 to site 4). This records a reservation for a flight starting earlier from B (local transaction) and responds positively (arrow from site 4 to site 2). Site 2 now chooses this flight and asks site 3 to cancel the reservation (dotted arrow downwards). Site 3 starts a canceling (compensating) local transaction which first fails because of a deadlock problem at a local database (the striped ball). Another attempt succeeds (the black ball) and site 3 ceases to execute the s-transaction. Meanwhile, site 2 issues a positive response to site 1 (arrow upwards) with the complete flight information. Site 1 (the travel agency) accepts the reservations and acknowledges them (a thin arrow downwards). Site 2 informs site 4 of the fact that the reservation is in effect (a thin arrow downwards).

This example exemplifies several things. The sub-s-transactions (at site) 3 and (at site) 4 are *alternative* s-transactions, since they both offer a suitable connecting flight. They could be started also in parallel, since the sub-s-transactions are not *conditionally dependent* on each other, neither is site 2 is restricted by the travel agency to apply an order. However, if the first reservation did not succeed, the sub-s-transaction 4 would be *failure dependent* on sub-s-transaction 2. Because a good business practice requires to cancel superfluous reservations, there is a dependency between the sub-s-transactions 3 and 4: only one *should* remain effective. In general, a successful s-transaction can thus contain sub-s-transactions that have been compensated (in the example the whole sub-s-transaction at site 3). Further a trial to compensate a transaction can fail either for semantical reasons or for technical reasons (the striped ball in Fig. 2).

4. Consistency preservation of s-transactions

The purpose of s-transactions is to preserve both *local* and *global* consistency. We separate between *static* and *dynamic* consistency. The dynamic local consistency at the ith database can be expressed by a binary *state transition relation* Dev_i. The state transition from a local database state s_1 to state s_2 is legal iff $Dev_i(s_1,s_2)$. Since all databases are empty at the beginning, the possible states can be represented as the set of *statically consistent* states CS_i, reachable from the empty database by legal state transitions. It holds $s \in CS_i$ iff $Dev_i^*(\emptyset,s)$, where Dev_i^* is the transitive closure of Dev_i. CS_i expresses the static consistency and can also be understood as a *predicate on states*, as was done e.g. in [Esw76]. Thus, dynamic consistency is more expressive than the static one. Note, that Dev_i is always *reflexive*, since aborts and read-only transactions are always allowed.

Only the local transaction programs manipulate the local databases and obey - or violate - any local and global consistency, expressed by the state transition relations. Let $LP_{ij}(IN;OUT)$ denote the jth local transaction program at ith site and let $LT_{ij}(in,s_1;\ s_2,out)$ denote its complete execution. s_1 is the database state it is started at, s_2 is the state after its execution, *in* is the input parameter, and *out* the values output by it to ATM or local user. Program P_{ij} that only causes state transitions occurring in Dev_i, no matter which input parameter is used, is *locally sound.*

Let us assume that each local transaction program P_{xy} executed as part of an s-transaction is locally sound at database x. Then, it can be shown by induction on the number of the local transactions in the spanning tree (c.f above) that any *s-transaction preserves local dynamic and static consistency at each local database.* This result can be extended in an obvious way for any set of interleaved s-transactions and other sound local transactions. The latter result relies on the assumption that each LDBS only allows correct interleaving of the local transactions, i.e. produces view serializable R-W histories [Bern87] among the local transactions hosted by it. Thus, if the STMs (or ATMs) do not perform any kind of concurrency control, the class of R-W histories achieved is a *proper superset of quasi-serializable histories* [Du89]. It is for further study, whether, and in which cases, it makes sense to prohibit some histories by ordering or aborting sub-s-transactions at STM or ATM level. It is also unclear how this could be done.

Because no isolation between s-transactions is enforced, the atomicity of s-transactions and their correctness are related in an unexpected way: The compensating local transaction can fail because it would violate consistency. As a result, the global s-transaction cannot run into completion, i.e. it "crashes". On the other hand, if the compensating local transaction is guaranteed not to abort for semantical reasons, e.g. by denying abort operations in the corresponding programs [Gar87], then one can show that it is not always sound [Vei90, p. 217]. The consistency embedded in local transaction programs is primarily a semantical issue and thus influences the D-autonomy.

The trade-off atomicity vs. correctness is a new phenomenon in the context of transactions and it is not well understood. The further exploration requires a formal analysis of the local and global consistency and the behavior of the local and global transactions. A complete model and the first results can be found in [Vei90].

5. Implementation experiences and application areas

The s-transaction model was implemented in the MAP761B project in a Micro-VAX II (VMS) environment [Hol88]. The software developed in the project consists of about 100000 lines of C, linked with the ORACLE database management system, and an X.400 compatible message handling system. The main part of the code is needed to implement the STM, which is a full-scale compiler and a run-time language interpreter for the transaction language called STDL. The MDBI module consists of about 20000 lines of code and it supports a multidatabase query language, facilities to import schemas, and to produce execution plans for queries, expressed in the transaction language. The ATM module mainly contains code supporting the export schema mechanism, query processing and a simple compensation support. The CS module consist of programs implementing the set of X.400 protocols [X.400] and, on top of it, a protocol that supports REQUEST, RESPONSE, ACK and ABORT messages of the s-transaction protocol.

The basic ideas of the s-transaction model have been adopted in other projects, too. E.g. in conjunction with the development of a flexible transaction model for a distributed infrastructure providing language support for inter-operable heterogeneous applications [Eli90], s-transaction concepts like autonomy preservation, alternative transactions and compensation are central. Similar properties can also be found in the transaction model for the Interbase system [Elm90]. The s-transaction model has been considered to be used in CIM environment [Hol87]. Usual EDI environments are also a promising field.

Acknowledgements. The authors wishes to thank Dr. Antoni Wolski from the Technical Research Centre of Finland (VTT) for commenting this paper.

6. References

[Banc85] F. Bancilhon,W.Kim, H. Korth: ***A Model of CAD Transactions.*** Proceedings of 11th VLDB.VLDB Endowment, USA 1985, pp. 25-33.

[Bern87] P.Bernstein, V.Hadzilacos, N.Goodman: ***Concurrency Control and Recovery in Database Systems.*** Addison-Wesley Publishing Company, USA 1987, ISBN 0-201-10715-5.

[Du89] W. Du and A.K. Elmagarmid: ***Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase.*** Proc. 15th VLDB Conf., Amsterdam, August 1989, pp. 347-355.

[Eli87] F.Eliassen, J. Veijalainen: ***Language Support for Multidatabase Transactions in a Cooperative, Autonomous Environment.*** Proc of TENCON'87, IEEE Comp. Society Press, 87CH2423-2, Vol 1, pp. 277-281.

[Eli90] F. Eliassen, R. Karlsen, ***Providing Application Interoperability using Functional Programming Concepts,*** CSR 90-06, Dept. Comp.Sc., University of Tromsø, 1990 (to appear in EurOpen '91).

[Ell88] D. Ellinghaus, M. Hallman, B.Holtkamp, K.-D.Kreplin,***A Multidatabase System for Transnational Accounting.*** Proc. of EDBT'88, Schmidt& Missikoff (eds.), Lecture Notes in Computer Science Nr. 303, Springer Verlag, Germany 1988, pp. 600-605.

[Elm90] A.K. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz, ***A Multidatabase Transaction Model for Interbase,*** Proc. 16th VLDB, 1990.

[Esw76] K. Eswaran, J. N. Gray, R. Lorie, I. Traiger: ***On the Notions of Consistency and Predicate Locks in a Database System.*** Comm. ACM, Vol. 19, No. 11, (November 1976), pp. 624-633.

[Gar87] H. Garcia-Molina, K. Salem: ***SAGAS.*** Proc. ACM SIGMOD Conf., San Francisco, May 1987, pp. 249-251.

[Gray78] J. Gray: ***Notes on database operating systems.*** In: Operating Systems: An Advanced Course, R. Bayer (ed.), Lecture Notes in Computer Science, Nr. 60, Springer-Verlag 1978, pp. 393-481.

[Gray81] J. Gray: ***Transaction Concept: Virtues and Limitations.*** Proc. of 7th VLDB Conf., pp. 144-154.

[Hol87] B. Holtkamp: ***Semantic Transactions in a CIM Environment.*** Proc. of Int. Conference on Data and Knowledge Systems for Manufacturing and Engineering. IEEE 0-8186-0820-X, pp. 23-29.

[Hol88] B. Holtkamp: ***Preserving Autonomy in a Heterogeneous Multidatabase System.*** Proc. of the 12th International Computer Software & Application Conf., COMPSAC '88. IEEE 88CH2611-2, pp. 259-266.

[Lit86] W. Litwin, A. Abdellatif: ***Multidatabase Interoperability.*** IEEE Computer, Vol. 19, No. 12 (December 1986), pp. 10-18.

[MAP86] Multidabase Services on ISO/OSI Networks.for Transnational Accounting. ***The final report of MAP761 + Annex on S-transactions..*** I.N.R.I.A (ed.), France, Apr. 1986, ISBN 2-7261-0442-8.

[Moss85] J. E.B. Moss: ***Nested Transactions, An Approach to Reliable Distributed Computing.*** The MIT Press, USA 1985, ISBN 0-262-13200-1.

[Vei88] J. Veijalainen, R. Popescu-Zeletin, ***Multidatabase Systems in ISO/OSI Environment.*** Standards in Information Technology and Industrial Control, N. Malagardis,T. Williams (eds.), IFIP,North-Holland, ISBN 0-444-70403-5, pp. 83-97.

[Vei90] J. Veijalainen: ***Transaction Concepts in Autonomous Database Environments.*** (Ph.D. thesis) GMD-Bericht Nr. 183, R. Oldenbourg Verlag, München, Germany 1990, ISBN 3-486-21596-5.

[X.400] CCITT Recommendation X.400, Message Handling Systems. CCITT, June 1984.

Multi-Level Transactions and Open Nested Transactions

Gerhard Weikum, Hans-J. Schek
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
E-Mail: {weikum,schek}@inf.ethz.ch

1 Introduction

Advanced transaction models and new correctness criteria for transaction executions have been proposed for the following reasons:

1) to provide better support for long-lived activities in advanced DBMS applications,
2) to relax the classical ACID paradigm, e.g., provide more flexibility as to when updates are made visible to concurrent transactions,
3) to support cooperation between the members of a group of designers (in CAD or CASE),
4) to fit more smoothly with object-oriented data models,
5) to capture more semantics about the operations of advanced DBMS applications,
6) to enhance (inter-transaction) parallelism by exploiting these semantics,
7) to model intra-transaction parallelism,
8) to allow user-defined or system-defined intra-transaction savepoints ("partial rollbacks"),
9) to deal with conversational transactions, and
10) to deal with multiple autonomous subsystems in a federated DBMS environment.

This paper gives an overview on multi-level transactions (MLTs) and its generalization toward open nested transactions (ONTs). These models meet some of the above demands, namely items 5, 6, 7, 8, 10, and, to a large extent, items 1, 2, and 4. Moreover, ONTs can serve as a basic building block in more sophisticated models for cooperative transactions and other long-lived activities.

Unlike many other advanced transaction models, MLTs and ONTs preserve two fundamental virtues of the classical transaction concept:

- they are based on a rigorous theoretical foundation that preserves the classical serializability theory as a special case, and
- they can reuse much of the well-proven implementation techniques that account for the high performance of current transaction processing systems.

This paper does not discuss implementation issues of ONTs, though (see [11, 16, 24, 26, 27]). Rather its main purpose is to show that even a relatively moderate extension of the classical transaction concept already provides a fairly powerful model that meets many of the above listed requirements yet stays within a well-defined theoretical framework.

2 A Summary of the Multi-Level Transaction Model

The idea of ONTs [8, 19] grew out of the seminal work on "spheres of control" by Bjork and Davies [3]. The special case of MLTs (also known as "layered transactions") is a variant of nested transactions where the nodes in a transaction tree correspond to operations at particular levels of abstraction in a layered system. The edges in a transaction tree represent the implementation of an operation at level L(i + 1) by a sequence of operations at the next lower level Li (for i = 0, ..., n-1 in bottom-up order). The key idea of multi-level concurrency control is to exploit the semantics of operations in **level-specific conflict relations** CON_i that reflect the commutativity or compatibility [6] of operations. This idea is applied to each of the levels.

Fig. 1 shows a schedule, i.e., concurrent execution of two MLTs (with execution order from left to right). Both transactions T_1 and T_2 withdraw some money from bank accounts *a* and *b* and deposit the money in account *c*. These high-level operations are implemented by read and write accesses to the underlying records. In terms of these read/write operations at the bottom level L0, the schedule of Fig. 1 is not (conflict-) serializable with respect to T_1 and T_2. However, at the higher level L1, one can exploit that the two *Deposit* operations commute, so that the L0 conflict on *c* becomes a **pseudo-conflict**. Therefore, the schedule can be regarded as serializable at the top level L1.

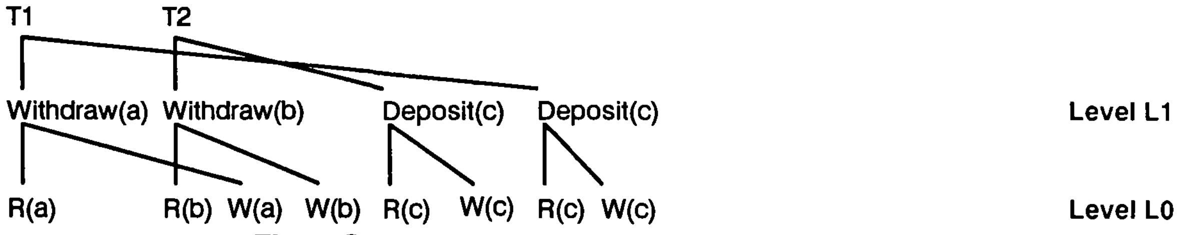


Fig.1: Concurrent Execution of MLTs

Casting these considerations into a more rigorous framework renders the following definition: a multi-level schedule is **multi-level serializable** iff between each pair of adjacent levels L(i + 1) and Li, the L(i + 1) serialization order $\preceq_{i+1}$ is acyclic and compatible with the execution order $<_{i+1}$ of conflicting L(i + 1) operations. In this definition, the order $\preceq_{i+1}$ corresponds to the usual serialization graph that is obtained by viewing L(i + 1) operations as transactions and Li operations as actions.

A practically important result of the multi-level serializability theory [1, 2, 10, 13, 23, 24] is the following. Suppose that for each pair (f,g) of conflicting L(i + 1) operations, there is at least one pair of Li operations among the children of f and g that conflict with respect to CON_i. Under this reasonable assumption, multi-level serializability is already guaranteed by the property of level-by-level serializability, that is, by requiring that the serialization graph between each pair of adjacent levels is acyclic. This means that different concurrency control mechanisms can be used at different levels as long as each of them ensures (conflict-) serializability in the classical sense.

MLT management allows more concurrency compared to conventional (single-level) concurrency control. On the other hand, transaction aborts can no longer be implemented by restoring the pre-transaction state of the modified L0 objects, since low-level modifications become visible to concurrent transactions at the end of each subtransaction (ST). The solution to this problem is to perform transaction aborts by means of inverse high-level operations that **compensate** complete STs rather than backing them out [2, 9, 13, 24, 25]. In the scenario of Fig. 1, for example, aborting transaction T2 after T1 is committed requires two compensating STs *Withdraw(c)* and *Deposit(b)*.

By viewing the compensating STs as additional regular operations of the transaction that is to be aborted, transaction aborts can be handled within the framework of multi-level concurrency control, too. The resulting correctness criterion is **complete serializability**, that is, multi-level serializability of the schedule in which the compensating STs are explicitly represented [2, 24, 25]. Complete serializability is an elegant treatment of recoverability in that it does not require a new correctness criterion, but rather extends the scope of serializability to the compensating STs.

3 The Generalization Toward Open Nested Transactions

The difference between MLTs and ONTs is that, in the more general model, the structure of the transaction trees is not restricted to layering; that is, siblings in the transaction tree are allowed to have different nesting depths. This generalization of MLTs is illustrated in the example of Fig. 2. Compared to Fig. 1, the example transactions have additional *Insert* operations that write into an application-managed transaction journal.

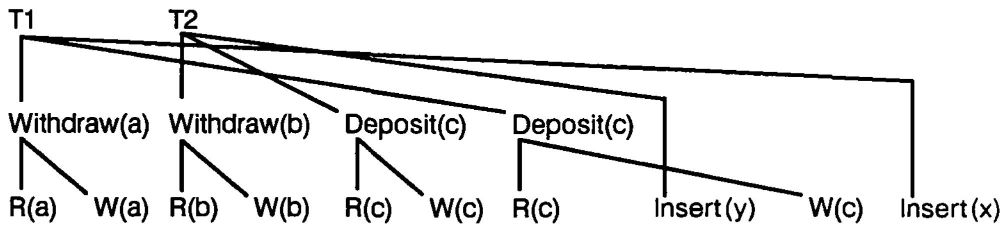


Fig.2: Concurrent Execution of ONTs

Because of the structural limitations of MLTs, the correctness of the concurrent execution shown in Fig. 2 cannot be proven within our multi-level serializability theory as sketched in Section 2. However, a rigorous correctness proof can be conducted in the general nested transaction models developed in [1] and [5]. Note that these models go beyond the conventional notion of nested transactions [12] in that they allow for STs that make updates visible to other top-level transactions before their parents commit.

A proof that the schedule of Fig. 2 is serializable, based on the model of [1], would run as follows. Because of the commutativity of *Insert(y)* and *W(c)*, we can exchange the order between these two nodes, thus separating the *Deposit* ST of T1 and the *Insert* action of T2. Next, because the set of nodes labeled *Withdraw*, *Deposit*, or *Insert* now form a front of serial computations, we can reduce the *Withdraw* and *Deposit* STs, i.e., prune their descendants. This results in a flat schedule that can be shown to be serializable by commutativity arguments. In general, the serializability proof for a schedule of arbitrarily deep ONTs involves the alternating application of commutativity and reduction.

4 Relaxing the ACID Paradigm

The so-called ACID paradigm of the classical transaction concept comprises the four properties atomicity, consistency-preservation, isolation, and durability. In the classical transaction concept, all four properties are bundled together. Ideally, however, these properties should be orthogonal in the sense that each of them can be waived without affecting the remaining properties (cf. [20]). The purpose of this section is to show that, in the ONT model, orthogonality can be achieved to some extent.

4.1 Consistency-preservation

In the classical transaction concept, only consistency-preservation can be viewed as orthogonal to the other three properties. Existing transaction managers take care only of atomicity, persistence, and isolation, whereas application programs are responsible for consistency-preservation. This is still true for ONTs.

4.2 Isolation

The strict isolation of transactions can be relaxed in the following two ways:

- by exploiting the semantics of the operations in the definition of conflicts, and
- by specifying which STs are "open" and which STs are "closed".

The first step toward more semantics is to incorporate general or state-dependent **commutativity** in the conflict definition between operations [14, 22]. For example, two *Deposit* operations on the same account are generally commutative, and two *Withdraw* operations are commutative in a state that allows both withdrawals to succeed (i.e., if there are sufficient funds). A more aggressive approach is to substitute commutativity by **compatibility** [6, 18], where two operations can be specified as compatible if their execution order is insignificant from the application point of view.

Exploiting commutativity or compatibility allows that incomplete transactions can make uncommitted updates visible to those transactions that perform a commutative or compatible operation. Note that this weaker notion of isolation does not leave the solid ground of serializability, since the conflict definition between pairs of operations is essentially a module that is "imported" into serializability theory. Furthermore, exploiting semantics to relax the definition of isolation has no impact on the validity of the other three ACID properties.

The programmer of an ONT can specify which STs do effectively make updates visible to compatible operations. He or she simply attaches one of the attributes "open" or "closed" to each node in the transaction tree (see also [4, 21] for similar approaches). An **"open" node** spawns a new "sphere of control"; that is, all updates of its descendants become visible to other transactions when the node's computation is completed. It is up to the node's parent to take measures that further protect the uncommitted updates, e.g., by acquiring a semantically higher lock. A **"closed" node**, on the other hand, simply extends the "sphere of control" of its parent; that is, its updates are still isolated against concurrent transactions as in the conventional model of nested transactions. Note that "sagas" [7] correspond to a special case of this model: the restrictions are that the children of a top-level transaction must not have further descendants, and that the operations that correspond to open STs are compatible with all possible operations.

Fig. 3 shows an example of a nested transaction with both open and closed STs, illustrated by striped and grey patterns, respectively. Suppose a transaction the purpose of which is to pay all monthly bills of a person, using electronic funds transfer rather than mailing checks. The *Transfer* transaction of the previous examples is now used as an operation of the entire transaction, i.e., as a ST. *Withdraw/Deposit* semantics is not exploited here; i.e., updates on accounts are modeled as closed STs. Suppose further that withdrawals make an entry in a special journal that is kept for security auditing. This *Append* operation is executed as an open ST so that its effect becomes immediately visible.

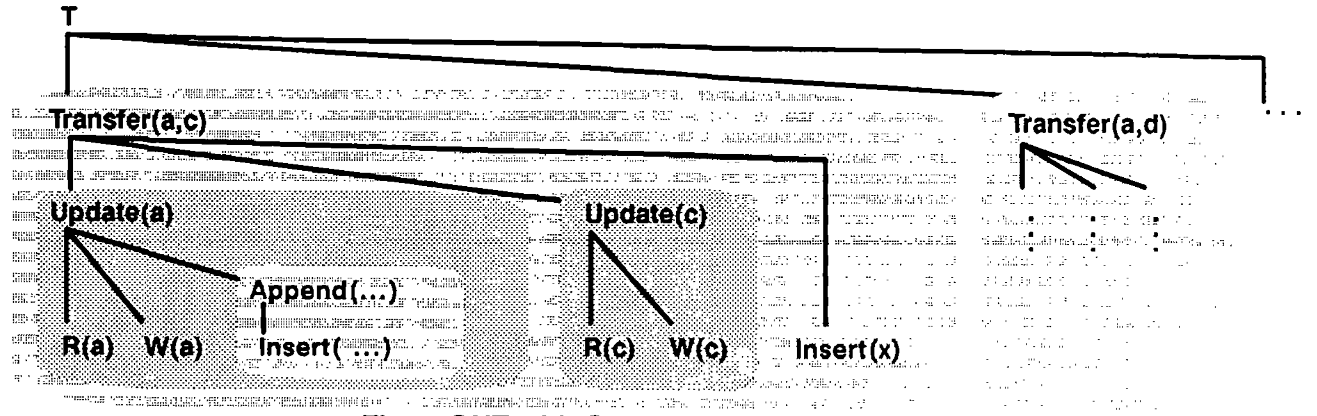


Fig.3: ONT with Open and Closed STs

4.3 Atomicity

By default, atomicity holds for transactions and STs at all nesting levels. For failures in closed STs, atomicity is achieved by restoring the old state of the objects that are modified within the ST. To roll back a completed open ST, this state–oriented undo method is not feasible, since the updates of the ST have already become visible to all compatible operations. Rather all completed open STs must be compensated by executing additional "counter–STs". In this sense, an ONT is never really rolled back but always completes successfully, possibly after having compensated its original effects.

Note that compensating a completed ST requires that both the forward ST and the compensating ST are atomic. Further note that, like in all nested transaction models, ST failures can be dealt with by undoing and restarting only the failed ST, which provides intra–transaction savepoints.

4.4 Persistence

In applications such as CASE, it is unacceptable that the work of a long-lived transaction is completely undone if the transaction eventually aborts (e.g., because of a crash). On the other hand, simply modeling a long design session as a sequence of shorter (top-level) transactions may not be an acceptable solution, because this would automatically make the (intermediate) results of each transaction visible to other transactions. The two requirements - long "spheres of control" with respect to isolation and shorter "spheres" with respect to persistence - can be reconciled by allowing the programmer of an ONT to attach a "persistent" attribute to individual STs.

The updates of a persistent ST are made persistent when the ST successfully completes; if the ST aborts before its completion then all its updates are undone. The programmer of an ONT can undo a completed persistent ST only by explicitly invoking a compensating ST. Compensating STs that are implicitly invoked by the system to undo a completed open ST affect only non-persistent STs.

Persistent STs (sometimes called "nested top-level actions" [11]) relax the atomicity of their parents since they survive the abort of a parent. In the example of Fig. 3, it may be reasonable to specify the *Append* ST as persistent. That is, an entry in the security journal is not removed if the entire transaction aborts. A persistent ST may have non-persistent as well as persistent descendants. The difference is that the non-persistent descendants are made persistent when their lowest persistent ancestor completes successfully, whereas persistent descendants are made persistent upon their own successful completion.

6. Conclusion

The MLT model and its generalization toward ONTs can be viewed as an evolutionary path from the classical transaction concept to more advanced transaction models. While being relatively conservative compared to other models, ONTs are yet a fairly powerful instrument that satisfies many of the requirements stated in the introduction.

MLT and ONT management is useful for a variety of potential applications, namely for

- extending data managers by an additional application-specific layer, such as building a filing and retrieval service for office documents on top of an existing DBMS [27],
- coping with the coexistence of global and local transactions in a federated multi-DBMS [17],
- building transaction management for object-oriented database systems such that the semantics of methods is exploited for enhanced concurrency [15, 18],
- supporting intra-transaction parallelism, by handling parallel subtransactions uniformly regardless of whether they belong to different transactions or to the same transaction, and
- exploiting the transaction support provided by advanced operating systems.

Acknowledgements

We have benefited from many discussions with colleagues working on the same or related problems. In particular, we would like to thank Catriel Beeri for the fruitful cooperation on MLTs.

References

[1] Beeri, C., Bernstein, P.A., Goodman, N., A Model for Concurrency in Nested Transactions Systems, JACM 36 (1989), 1

[2] Beeri, C., Schek, H.-J., Weikum, G., Multi-Level Transaction Management, Theoretical Art or Practical Need?, 1st Int. Conf. on Extending Database Technology, 1988, Springer, LNCS 303

[3] Davies, C.T., Data Processing Spheres of Control, IBM Systems Journal 17 (1978), 2

[4] Elmagarmid, A., Leu, Y., Litwin, W., Rusinkiewicz, M., A Multidatabase Transaction Model for InterBase, VLDB 1990

[5] Fekete, A., Lynch, N., Merritt, M., Weihl, W., Commutativity-Based Locking for Nested Transactions, Technical Report MIT/LCS/TM-370, MIT, Cambridge (Mass.), 1988, to appear in: Journal of Computer and System Sciences

[6] Garcia-Molina, H., Using Semantic Knowledge for Transaction Processing in a Distributed Database, TODS 8(1983), 2

[7] Garcia-Molina, H., Salem, K., Sagas, ACM SIGMOD Conf., 1987

[8] Gray, J., The Transaction Concept: Virtues and Limitations, VLDB Conf., 1981

[9] Korth, H.F., Levy, E., Silberschatz, A., Compensating Transactions: A New Recovery Paradigm, VLDB Conf., 1990

[10] Martin, B.E., Modeling Concurrent Activities with Nested Objects, Int. Conf. on Distr. Computing Systems, 1987

[11] Mohan, C., et al., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, IBM Research Report RJ6649, San Jose, 1989, to appear in: ACM TODS

[12] Moss, J.E.B., Nested Transactions: An Approach to Reliable Distributed Computing, MIT Press, 1985

[13] Moss, J.E.B., Griffeth, N.D., Graham, M.H., Abstraction in Recovery Management, ACM SIGMOD Conf., 1986

[14] O'Neil, P.E., The Escrow Transactional Method, TODS 11 (1986), 4

[15] Rakow, T.C., Gu, J., Neuhold, E.J., Serializability in Object-Oriented Database Systems, IEEE Data Eng. Conf., 1990

[16] Schek, H.-J., Paul, H.-B., Scholl, M.H., Weikum, G., The DASDBS Project: Objectives, Experiences, and Future Prospects, IEEE Transactions on Knowledge and Data Engineering 2 (1990), 1

[17] Schek, H.-J., Weikum, G., Schaad, W., A Multi-Level Transaction Approach to Federated DBMS Transaction Management, Int. Workshop on Interoperability in Multidatabase Systems, 1991

[18] Skarra, A.H., Zdonik, S.B., Concurrency Control and Object-Oriented Databases, in: W. Kim, F.H. Lochovsky (eds.), Object-Oriented Concepts, Databases, and Applications, ACM Press, 1989

[19] Traiger, I.L., Trends in System Aspects of Database Management, 2nd Int. Conf. on Databases, Wiley Heyden, 1983

[20] Wächter, H., Reuter, A., Basic Concepts and Implementation Strategies of the ConTract Model (in German), Informatik Forschung und Entwicklung 5 (1990), 4

[21] Walter, B., Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications, VLDB Conf., 1984

[22] Weihl, W.E., Commutativity-based Concurrency Control for Abstract Data Types, IEEE Trans. Comp. 37 (1988), 12

[23] Weikum, G., A Theoretical Foundation of Multi-Level Concurrency Control, ACM PODS Conf., 1986

[24] Weikum, G., Principles and Realization Strategies of Multi-Level Transaction Management, Technical Report DVSI-1987-T1, TH Darmstadt, 1987, to appear in: ACM TODS

[25] Weikum, G., Enhancing Concurrency in Layered Systems, 2nd Int. Workshop on High Performance Transaction Systems, 1987, Springer, LNCS 359

[26] Weikum, G., Hasse, C., Broessler, P., Muth, P., Multi-Level Recovery, ACM PODS Conf., 1990

[27] Weikum, G., Schek, H.-J., Architectural Issues of Transaction Management in Layered Systems, VLDB Conf., 1984

IEEE Computer Society
1730 Massachusetts Avenue, NW
Washington, DC 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398

Mr. David B. Lomet
9 Cherry Lane
Westford, MA 01886
USA