SEPTEMBER 1987 VOL. 10 NO. 3

a quarterly bulletin of the
Computer Society of the IEEE
technical committee on

Data Engineering

CONTENTS

Letter from the TC chairman 1
S. Jajodia

Data Engineering in Transition 2
R. Shuey

Letter from the Editor 3
S. Sarin

Concurrency Control and Recovery for Global Procedures in Federated Database Systems 5
R. Alonso, H. Garcia-Molina, K. Salem

An Update Mechanism for Multidatabase Systems 12
Y. Breibart, A. Silberschatz, G. Thompson

Superdatabases: Transactions Across Database Boundaries 19
C. Pu

An Optimistic Concurrency Control Algorithm for Heterogeneous Distributed Database Systems 26
A. Elmagarmid, Y. Leu

Pragmatics of Access Control in Mermaid 33
M. Templeton, E. Lund, P. Ward

A Federated System for Software Management 39
D. Heimbigner

Information Interchange between Self-Describing Databases 46
L. Mark, N. Roussopoulos

Data Retrieval in a Distributed Telemetry Ground Data System 53
C. Steinberg

Calls for Papers 60

TC Membership Application Form 64

SPECIAL ISSUE ON FEDERATED DATABASE SYSTEMS

Editor-in-Chief, Database Engineering

Dr. Won Kim
MCC
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3439

Associate Editors, Database Engineering

Dr. Haran Boral
MCC
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3469

Prof. Michael Carey
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
(608) 262-2252

Dr. C. Mohan
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
(408) 927-1733

Prof. Z. Meral Ozsoyoglu
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106
(216) 368-2818

Dr. Sunil Sarin
Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142
(617) 492-8860

Chairperson, TC

Dr. Sushil Jajodia
Naval Research Lab.
Washington, D.C. 20375-5000
(202) 767-3596

Vice-Chairperson, TC

Prof. Krithivasan Ramamrithan
Dept. of Computer
and Information Science
University of Massachusetts
Amherst, Mass. 01003
(413) 545-0196

Treasurer, TC

Prof. Leszek Lilien
Dept. of Electrical Engineering
and Computer Science
University of Illinois
Chicago, IL 60680
(312) 996-0827

Secretary, TC

Dr. Richard L. Shuey
2338 Rosendale Rd.
Schenectady, NY 12309
(518) 374-5684

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database 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 Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database 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.

Dear TC DE Members and Correspondents:

Since it has been a year when I last wrote you, it is time for me to provide an update of our activities over the past year.

The Technical Committee on Database Engineering has become the **TECHNICAL COMMITTEE ON DATA ENGINEERING.** The new name accurately reflects the evolving nature of the database technology. It encompasses not only the more traditional aspects of databases and knowledge bases but also many topics that allow a broader scope. It is appropriate that the accompanying editorial is written by Dick Shuey who has been active in the IEEE Communications Society for many years and is new to our TC.

As a result of the financial uncertainty, the publication of **Data Engineering** fell behind schedule in 1986. Thanks to the editor-in-chief, Won Kim, and the associate editors, it is being published once again in a timely manner. There have been some problems in putting together the December 1986 issue on the European ESPIRIT project. Guy Lohman has responded to our call and is organizing an alternative issue on query optimization that should be published in November. I might also add that we are eliminating the two charges that were initiated last September: the voluntary page charge for the papers and charge for the conference announcements.

Although we have been successful in getting financial support this year from the IEEE Computer Society for publishing **DE**, we still need a long term solution to our financial problem. Several alternatives were proposed and considered. The only satisfactory alternative appears to be establishing a subscription fee for **DE**. This will provide us with a continuing source of income and eliminate financial uncertainty.

I urge you to express your opinion on this issue. If you support this alternative, it is important that you write and tell me. Your past support has been greatly appreciated, but an overwhelming positive response from the membership is needed in order to influence the officers of the IEEE Computer Society.

Finally I would like to remind you that the Fourth Data Engineering Conference will be held February 2-4, 1988 in Los Angeles, California. John Carlis, the Program Chair, tells me that he has received over 210 submissions. It appears that we will have once again an excellent conference. I encourage all of you to attend and participate in this major conference.

Sushil Jajodia

Sushil Jajodia
August 14, 1987

DATA ENGINEERING IN TRANSITION

Richard L. Shuey

This is the first issue of the quarterly bulletin on DATABASE ENGINEERING that has the new name DATA ENGINEERING. The change in name, in conjunction with the earlier adoption of the name for the International Conference on Data Engineering and the subsequent change in name of the parent technical committee of the Computer Society, completes a transition. That transition highlights the evolutionary path that the field is taking. The primary roots of data engineering lie in the field of database technology. That fact accounts for the earlier titles emphasizing databases. Database technology remains one of the more mature disciplines of what we now perceive as the broader field of data engineering. Database technology and theory continue to provide a base from which this new field is evolving.

The overall objective of data engineering is to insure that in large computer and information systems, the data required by the individual components and their internal processes will be available when needed, in the proper form to be used, and appropriately controlled with respect to access and security. As it becomes more common for databases to be shared by multiple applications and multiple users, the technology important to attaining that objective will extend further beyond that of databases per se. Relevant technologies include those of: communication systems; security and cryptograph; artificial intelligence; the management and control of data both within and external to a database system; distributed system architecture; interfacing between unlike and independent information systems; integration of database technology with programming languages; the software for transaction and data systems; the coupling to existing information found in books, publications, technical handbooks, graphs, etc.; and, the coupling to data currently being created that is not in a convenient form for input to a computer. To illustrate this dependency, consider:

Communication technology is an old, mature but rapidly evolving field. That technology is applicable not only to communications within a computer but also to communications between computers in a laboratory environment, a local area such as an institution, a city, within nations, and internationally. The management, control, integrity, security, and interfacing of data must be addressed at many levels in the information system hierarchy, including the communications systems and databases involved. We do not have an adequately mature engineering discipline nor complete sets of techniques and mechanized tools to permit us to build and maintain with confidence the very large complex data systems that exist in today's society. Advances in software engineering methodology and the development of associated mechanized software development facilities (factories) can help. Much of the data that we need in an engineering design system exists in published articles, handbooks, graphs, etc. The format in which these data are presented is not consistent or standard. The scales of graphs may be different, the results of materials tests depend on how the material was produced and the testing environment, etc. Engineering data of this nature are needed as an input to computerized design systems and a satisfactory solution is not evident. A significant effort is needed to address this challenge. The definitions of the terms data, metadata, heuristic, information, intelligence, and transformations between these categories are not entirely clear, nor universally agreed to. Yet, the importance of the underlying concepts is agreed to by all. The concepts keep arising in data engineering and many other fields. Progress will require interdisciplinary efforts.

As most engineering fields, data engineering is objective oriented. We wish to design systems that make data available. We must be willing, in fact expect, to utilize any technology that will lead to better data systems. We must seek, solicit, involve, and welcome interdisciplinary participation in our work. Not only will we directly benefit, but our perspective on areas of mutual interest will in the long run benefit our collaborators. The challenge to the membership and officers of the Technical Committee on Data Engineering is clear. We must be aware of relevant work in other fields. We must make the participants in those other fields aware of our work in overlapping areas of potential mutual interest. Where sound mutual interest exists or develops, we must seek better coordination and even integration of effort. In a practical sense we should actively seek involvement by interdisciplinary professionals in our publications, meetings, and membership. This must be a two-way street. We will push for adequate coverage of viewpoints from theory to applications. The theories must be tested by application, and the applications must be evaluated to permit validation and extension of the theories.

Those involved directly with the operation of the Technical Committee and the International Data Engineering Conference have actively encouraged and promoted evolution along these lines. We have made progress, but not as rapidly as we would like. To the degree that name changes can be important, particularly in interacting with others, we believe that this name change is an important evolutionary step. In going further down the indicated path we shall need the support and participation of the membership as a whole. We seek your active and personal participation.

Letter from the Editor

This issue of IEEE Data Engineering (formerly Database Engineering) addresses the subject of Federated Database Systems. A federated database system consists of a collection of individual autonomous databases, which may be heterogeneous, that agree to share information and cooperate in some controlled way. Because of the proliferation of databases that need to share information and yet retain local control, methods for designing and implemented federated systems are becoming increasingly important. This need was anticipated in an early paper by Hammer and McLeod.[1] Further evidence of its importance is the recent growth of research in this field and the announcements by some DBMS vendors of distributed database products that include "gateways" to heterogenous systems. Because retrieval from heterogeneous or federated databases, and the related issue of schema integration,[2] has been an active research topic for several years, I have devoted this issue to topics that have received less coverage.

The first four papers in this issue are about concurrency control in federated systems. In the same way that federated data access must deal with heterogeneous data models, languages, and schemas, concurrency control in federated systems must deal with heterogeneity in the concurrency control algorithms used by the individual databases. Except for a recent paper by Gligor and Popescu-Zeletin,[3] the problem of heterogeneous concurrency control has not been addressed significantly in the literature. The inclusion in this issue of four papers on this subject is an attempt to remedy this.

The concurrency control solutions presented are based on different assumptions about how much information about the individual local concurrency controls is available to the global concurrency controller. Alonso, Garcia-Molina, and Salem assume no knowledge of local concurrency controls. Relying on global locking at the granularity of individual databases, they describe "altruistic locking" which allows early release of locks subject to some constraints that preserve serializability. Alonso et al. also address the problem of recovering from failures of global transactions, including compensating for local actions that have already committed, and present a mechanism, called a "saga," for further increasing concurrency when global serializability and consistency constraints that span multiple databases do not need to be preserved. The algorithm presented by Breitbart, Silberschatz, and Thompson also assumes no knowledge of local concurrency controls or of the data objects accessed, and ensures global serializability by controlling the order in which transactions are submitted to individual databases. This scheme avoids deadlock and also ensures that the interaction between local transactions (not seen by the global concurrency controller) and global transactions does not violate serializability.

---

[1] M. Hammer and D. McLeod. "On Database Management System Architecture," in *Infotech State of the Art Report vol. 8: Data Design*. Pergamon Infotech Limited. 1980. (Also available as MIT Laboratory for Computer Science Technical Memo TM-141.) This paper appears to have introduced the term "federated database."

[2] C. Batini. M. Lenzerini. and S.B. Navathe. "A Comparative Analysis of Methodologies for Database Schema Integration." *ACM Computing Surveys 18*. 4. December 1986.

[3] V. Gligor and R. Popescu-Zeletin. "Concurrency Control Issues in Distributed Heterogeneous Database Management Systems." in *Distributed Data Sharing Systems*. eds. F. Schreiber and W. Litwin. North-Holland. 1985. See also "Transaction Management in Distributed Heterogeneous Database Management Systems." *Information Systems 11*. 4. 1986.

The concurrency control papers by Pu and by Elmagarmid and Leu, in contrast, assume that the serialization order of transactions on a local database can be observed, and present "optimistic" algorithms that check the global transaction ordering after the fact and abort transactions that cannot be serialized. Elmagarmid and Leu make the additional assumption that the local read and write sets of global transactions are known, thereby reducing the abort rate. Pu's "superdatabases," on the other hand, support hierarchical composition of global transactions.

The remaining papers in this issue address other practical problems in federated database systems. Templeton, Lund, and Ward describe how access control is supported in the MERMAID front-end to existing heterogeneous databases. Heimbigner describes an experimental federated system, Keystone II, that connects workstations each of which is running an existing software development environment. While Keystone II is not a DBMS in the traditional sense, it addresses important object sharing issues that will arise when emerging "object-oriented" database technology is extended to distributed environments, such as deciding which objects referenced by a given complex object should be transmitted when the latter is imported by one node from another. Mark and Roussopoulos describe a set of data management tools and a data dictionary for information interchange (including interchange of meta-data), based on work done for NASA.

The last paper in this issue is an "unsolicited" one that is perhaps loosely related to the federated systems theme. Steinberg describes a proposed architecture for handling both conventional and real-time data requests in a telemetry data system for the Jet Propulsion Laboratory. While we rely for the most part on invited papers for the Bulletin, we encourage unsolicited submissions as well. An unsolicited paper will be forwarded to an appropriate editor based on the editor's interests and possible matches with themes of upcoming issues. We also encourage submissions on the themes of past issues, from researchers or practitioners whose work may have been overlooked the first time around.

I would like to thank the authors for contributing to what I hope the readers find an interesting issue.

Sunil Sarin
August 1987

CONCURRENCY CONTROL AND RECOVERY FOR GLOBAL PROCEDURES IN FEDERATED DATABASE SYSTEMS

Rafael Alonso
Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton N.J. 08544

1. Introduction

As the name suggests, a federated database is actually a collection of databases cooperating for mutual benefit. The particular quality that distinguishes federated databases from other distributed databases is the degree of autonomy of the members of the federation. For example, it is assumed that federation members may not be forced to perform activities for other members, that each determines which parts of its local database will be shared, and how they will be shared, that each maintains its own database schema, and that they may withdraw from the federation if they so choose [Heim85a].

The purpose of the federation is to increase the capabilities of its members, i.e., to permit transactions that deal with non-local data. The federation permits members controlled access to foreign databases. We will use a simple model of inter-database interaction: all interaction will be accomplished through the use of *global procedures.* A global procedure is initiated at some federation member (its home, or local, node), and can request other members to execute procedures (usually transactions) on its behalf. Other methods of cooperation are possible, e.g., a federation member may agree to "export" a particular portion of its local data to another member [Heim85a]. However, for this discussion we will consider only global procedures.

In general, a number of global procedures will be in progress simultaneously within the federation. The question we will address in this paper is how concurrent global procedures should be managed. Specifically, we are interested in concurrency control and recovery for global procedures. However, before we address these issues we first present simple models of the federation and its members which we can then use in our discussions.

Each member, or node, of the federation consists of a transaction and database manager. (We will refer to the local transaction/database manager of the ith member as LTM_i.) An *LTM* presents a transactional interface to the local database to users at its local node. It also presents a transactional interface (which may or may not be the same as the local interface) to a *global procedure manager* (*GPM*) residing at the local node.

The *GPM* is the interface between the local node and the rest of the federation. The *GPM*s of federation members are connected by some type of communications mechanism. A *GPM* receives requests for service from the *GPM*s of other members of the federation, translates those requests into a form that is palatable to its *LTM*, and forwards the requests to it.

In addition to handling requests from other nodes, each *GPM* provides a federation interface to applications at its local node. The local interface provided by GPM_i is a set of global procedures $P_{i1},..,P_{in_i}$, i.e., these procedures are available to local users at node i. This interface is the mechanism under which global procedures are initiated.

Each global procedure consists of a (possibly ordered) set of requests for service at other nodes. As described above, each request is translated by the target *GPM* into a transaction for its *LTM*, and the results are returned to the *GPM* making the request. The block diagram in Figure 1 describes an n-member federated system of the type we have just described.

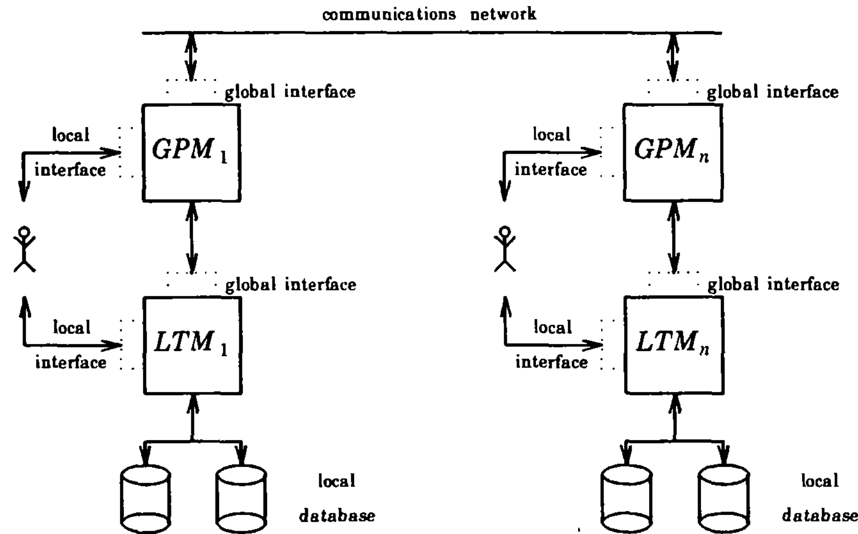


Figure 1 - Federated System Block Diagram

To reflect the autonomy of federation members, we make several assumptions in the context of the federation model:

- A *GPM* has no knowledge of the structure and organization of the local database, nor of the implementation of its *LTM*. The *GPM* knows only of the transactions available to it through the *LTM*'s global interface.
- Support for global procedures is implemented entirely within the *GPM*s at each node; no modifications to the *LTM*s are required.
- The *only* interaction between a *GPM* and its *LTM* is through that *LTM*'s global, transactional interface.

These assumptions have a number of important ramifications, some of which we discuss next:

- If a *GPM* submits a request for an update transaction to its *LTM* on behalf of a global procedure, it is not possible to lock or shield the updates once the local transaction has completed. The *GPM* may shield the updates from other *global procedures* by refusing to submit further requests to the *LTM* on behalf of those procedures until the first global procedure is finished. However, there is no way for the *GPM* to keep transactions submitted through the *LTM*'s local interface from seeing the updates. Note that if a *GPM* submits more than one transaction to its *LTM* on behalf of a single global procedure then there is no way to guarantee that that global procedure can be serialized with transactions submitted locally to the *LTM*.
- If concurrency controls are to be implemented by the *GPM*s for global procedures, then dependencies among global procedures must be maintained with a data granularity no smaller than an entire local database or federation node. In other words, if two global procedures have transactions submitted on their behalf to the same *LTM*, those transactions (and thus the global procedures they are a part of) must be assumed, from the point of view of the *GPM*s, to

conflict. Thus the lockable entities in a global concurrency control mechanism will be the nodes of the federation themselves.

- Once a transaction has been run by an *LTM* on behalf of some global procedure, updates made by that transaction cannot be rolled-back, or undone, by the *GPM*. The only recourse of the *GPM* in this case is to request the execution of a *compensating transaction* by its *LTM*. Thus a global procedure cannot be truly atomic in the transactional sense. We will discuss this issue further in Section 4.
- Local transactions submitted through the *LTM*'s local interface are not affected by global concurrency controls. Although locally and globally submitted transactions may conflict within the *LTM*, the *LTM* can deal with this conflict as it sees fit, e.g., it may abort either type of transaction at any time.

In the rest of this paper we will briefly survey two approaches to global concurrency control that operate within this framework, and then discuss recovery of global procedures. Given our space limitations, our objective will be to intuitively explain these approaches and not to provide details. The details of each of the concurrency controls, together with discussions of the recovery issues, are given in separate papers.

Altruistic locking [Sale87a] is an extension of two-phase locking. Like two-phase locking, altruistic locking results in serializable schedules. A global locking protocol, such as two-phase or altruistic locking, plus local concurrency controls that guarantee serializable schedules, can ensure the serializable execution of global procedures.[†] Altruistic locking can make this guarantee while allowing potentially greater concurrency than two-phase locking.

For many applications it is not necessary to serialize global procedures. A *saga* [Garc87a] is a procedure that can be broken up into a collection of smaller transactions which can be interleaved in any way with other transactions. The transactions in a saga are related to each other and should be executed as a (non-atomic) unit. Since global procedures in a federated database are collections of service requests (i.e., local transactions), they are natural as sagas if the application semantics do not require serial consistency for the entire procedure.

To make the discussion of sagas and altruistic locking clearer, we will consider the database facilities of a hypothetical car rental company which has a number of independently owned outlets in several cities across two states. Each outlet has its own local database which records reservations and the state of the local fleet. The outlets' local databases are joined into a single federated system, shown in Figure 2. In the figure, each node is identified by a single capital letter.

2. Sagas

Imagine a car rental customer making a business trip which takes him to cities A, B, C, and then D. At each city, he wishes to reserve a car from the local outlet for one week. P_{A1} is a global procedure (available at node A) which implements this by requesting reservations from each of the nodes. For the sake of this discussion, we will assume that P_{A1} makes requests of A, B, C and D in that order.

It is probably not necessary for P_{A1} to hold on to all of its resources until it completes. For instance, once P_{A1} successfully gets the reservation at node A, it could immediately allow other global procedures to make requests at A. However, we do not wish P_{A1} to be simply a collection of independent requests (local transactions). P_{A1} is a unit which should be successfully completed or not done at all. For example, if the reservation at C cannot be obtained then it is likely that the previously obtained reservations at A and B will have to be changed. Thus it seems natural to

---

† Global procedures can be serialized with local transactions only if they submit at most one transaction request to each node.

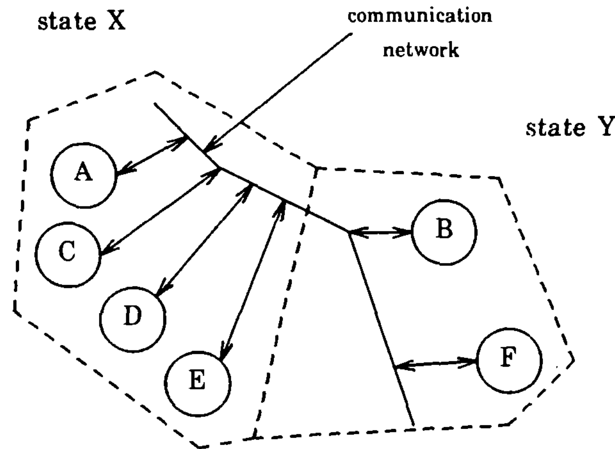


Figure 2 - Example Federated Database System

treat P_{A1} as a saga.

To amend partial executions of a saga, each transaction in the saga should be provided with a *compensating transaction*. A compensating transaction undoes, from a semantic point of view, its associated transaction. In our example, if a transaction in P_{A1} reserves a car at B, then its compensating transaction cancels the reservation at B. The system makes a *semantic atomicity* guarantee for the saga: if the saga is aborted, any transactions in the saga that have already completed will be compensated for. We discuss compensation further in Section 4.

By running P_{A1} as a saga rather than ensuring its total serializability with other global procedures, the system can obtain some potentially substantial performance benefits. In particular, it is not necessary to maintain any global locks to synchronize a (global) saga. The home node of the saga simply submits its transaction requests to the proper federation nodes.

3. Altruistic Locking

Sagas are not always the appropriate abstraction to use for global procedures. In some cases, it really is necessary to synchronize a procedure with other global procedures. For example, consider a procedure P_{A2} used to determine the total number of cars available at the outlets in state X. P_{A2} requests that nodes A, C, D, and E (i.e., all of the nodes in state X) run local transactions to determine the number of cars available at the local outlet. To guarantee an accurate count, P_{A2} should be synchronized with other global transactions that modify inventories of available cars. P_{A3}, a procedure that records the loan of cars from node A to node C, is an example of the type of global transaction that, if not serialized, could destroy the accuracy of P_{A2}'s count.

This problem can be avoided by using global two-phase locking. Before requesting service from a node, that node must be locked and the lock must be held until the end of the global procedure. A locked node cannot service requests for any other global procedures. (Recall that the locking granularity at the global level is the node. Locking a smaller granule (e.g., a relation) would require that the GPM have some knowledge of the structure of its local database.) However, the performance problems of global two-phase locking may be substantial. If P_{A2} manages to lock node A first, P_{A3} would have to wait for P_{A2} to query all four nodes in state X before the lock on A is released and it can attempt to continue. The situation gets worse as global procedures get larger and access more nodes.

Altruistic locking is a locking protocol that may ease this type of performance problem. Like two-phase locking, altruistic locking ensures that execution schedules are serializable. However, it provides a mechanism for global procedures to release locks before they finish, possibly freeing waiting global procedures to acquire the lock and continue processing.

Applied to global locking, the altruistic locking protocol works as follows. As with two-phase locking, a global procedure must lock a node before it can request work from that node. Once the global procedure's request has been processed, and *if the global procedure will request no further work from that node*, it can *release* its lock on the node. Releasing a lock is a special operation, peculiar to altruistic locking. Releasing a lock is like conditionally unlocking it. Other global procedures waiting to lock the released node may be able to do so, but only if they are able to abide by certain restrictions. Note that a global procedure is free to continue locking new nodes after it has released locks, i.e., locks and releases need not be two-phase.

The set of nodes that have been released by a global procedure constitute the *wake* of that procedure. If P_{A3} locks a node in P_{A2}'s wake, we say that P_{A3} is in the wake of P_{A2}. Under the simplest altruistic locking protocol each global procedure concurrent with P_{A2} must remain completely inside the wake of P_{A2}, or completely outside, until P_{A2} has finished. For example, P_{A3} must lock only nodes released by P_{A2}, or it must not lock any nodes released by P_{A2}.

While this may seem somewhat restrictive, consider our previous example in which P_{A3} was forced to wait (in the worst case) for P_{A2} to make requests of all four nodes in X. If altruistic locking were used, P_{A2} could release each node as soon as its local query at that node was successfully completed. Thus P_{A3} could lock A as soon as P_{A2} moved on to C, and could lock C as soon as P_{A2} moved on to D. One nice feature of altruistic locking is that it is certain to provide at least as much concurrency as two-phase locking, and possibly more. In other words, there is no situation in which a global procedure that would have been permitted to run under two-phase locking would be prohibited from running under the altruistic protocol. Of course, the actual performance advantages of altruistic locking would depend on the resource requirements of global procedures and the cost of global operations (such as requesting services or locks) in a particular application.

A more complicated version of the altruistic protocol permits a global procedure to "straddle" another's wake, i.e., to be partially in and partially out of the wake, in some circumstances. For example, consider a global procedure P_{A4} that records the loan of cars from A in state X to B in state Y. Imagine that P_{A2} (which counts the cars available in X) and P_{A4} run concurrently and that P_{A2} manages to lock A before P_{A4}. Under the simple protocol just described, once P_{A2} releases A, P_{A4} can access that node. However, B is outside the wake of P_{A2}. Since P_{A4} is already in the wake of P_{A2} (because it locked A), it will have to wait until P_{A2} finishes before it can lock B.

This is unfortunate because we know that P_{A4} could have been serialized after P_{A2} even if it locks B without waiting for P_{A2} to finish. We know this because P_{A2} never accesses B. The more complicated altruistic locking protocol can take advantage of information about which nodes a global procedure will visit during its lifetime. It lets global procedures make use of another special concurrency control operation called the *mark*. A global procedure marks a node to indicate that it will access that node sometime in the future.

Marking is an option, not a requirement. By marking nodes a global procedure is assisting other global procedures by informing them of its intended behavior. A global procedure that does mark must abide by several restrictions in order for the marks to be useful. A global procedure that chooses to mark may not lock a node without marking it first, and must stop marking nodes once it has issued a release operation.

As we have already hinted, a global procedure running in the wake of a marking global procedure need not always remain within the wake. It may lock a node outside of the wake provided that node has not been marked by the procedure in whose wake it is running. In our example, this

means that P_{A4} would be permitted to lock B after locking A. Since P_{A2} never needs access to B, it will not have marked B.

Of course, we cannot permit P_{A4} to simply lock and request a local transaction at B even if P_{A2} hasn't marked it. Imagine a global procedure P_{A5} that records the transfer of cars from B to E. Conceivably, P_{A5} could lock B after P_{A4} was finished and then lock E before P_{A2} got that far. This results in an unserializable schedule among the three global procedures P_{A2}, P_{A4} and P_{A5}.

To avoid this kind of situation, P_{A4} should indicate that any global procedure locking B after P_{A4} must be serialized after P_{A2}. The altruistic protocol prescribes a simple mechanism for accomplishing this. P_{A4} is required to release B *on behalf of* P_{A2} before it can lock B. In other words, P_{A4} never really leaves the wake of P_{A2}. Instead, it expands the wake to include the node that it wishes to access. The protocol also permits a global procedure outside another's wake to enter the wake under similar conditions. Details can be found in [Sale87a].

4. Recovery

Whether global procedures are scheduled serializably or are treated as sagas, some recovery mechanism will be needed. For this reason, the *GPM* at each node must have access to some stable storage mechanism on which it can maintain a record of its activity. Such a log would record the progress of global procedures initiated at the local node, plus the status of requests being executed by the local *GPM* on behalf of other nodes. Of course, requests being executed for a global procedure can fail for reasons other than power losses, e.g., the local transaction spawned by the request might deadlock in the *LTM*.

As we have already observed, true atomicity is not possible for global procedures. However, some useful guarantees can be made in case global procedures run into problems when executing. In general, a global procedure can be recovered *forwards* or *backwards* from the failure of a request. Forward recovery involves retrying the failed request. This is useful for transient failures like deadlocks of local transactions. The decision to retry might be made by the *GPM* at the site of the failure, or by the *GPM* at the node from which the request initiated.

Backward recovery is the abortion of the global procedure, undoing its effects. Backward recovery in a federated system poses special problems because of the autonomy of the federation members. As has already been mentioned, it is not possible, in general, to completely erase the effects of a global procedure. The means of aborting a global procedure is the execution of *compensating transactions* at the nodes visited by the global procedure. A compensating transaction undoes, from the view of the semantics of the local database, the effect of a previous transaction.

Global procedures cannot be said to have a commit point in the same sense as transactions do. Updates made on behalf of global procedures are committed, i.e., made available to others, as soon as the local transaction that implements the update on behalf of the global procedure has committed. However, at some point the initiating *GPM* must decide whether to externalize the results of the global procedure. Thus it is perhaps better to say that there are "externalization dependencies" among global procedures, rather than commit dependencies. Such dependencies might arise if a global procedure releases (or unlocks) a node that has been updated on its behalf.

Depending on the application, it may or may not be desirable to delay the externalization of a global procedure until those procedures on which it depends have been externalized. When a global procedure is aborted, other procedures dependent on it can be aborted as well (using compensating transactions). Note, however, that there is no way to keep locally-submitted local transactions from seeing updates made on behalf of a global procedures and externalizing them. In some applications, maintenance of externalization dependencies may be deemed unnecessary. For example, if a global procedure increments a flight's reservation count by one in an airline reservation system, it may be acceptable to allow other procedures to see the modified count without making them

dependent on the reservation procedure.

Whether or not dependencies are maintained, the use of compensating transactions allows the *GPM's* to make a *semantic atomicity* guarantee [Garc83a] for global procedures: either the procedure will be completely executed or compensation will be requested for the completed parts of a partially executed procedure. Global procedures which use altruistic (or two-phase) locking can have a stronger guarantee if they do not release nodes at which local update transactions have been executed on their behalf, i.e., if they do not allow other global procedures to become dependent on them. (Note that global procedures which use altruistic locking but do not release their updates can still achieve more concurrency than global procedures using two-phase locking.) If *P* is such a procedure, it can be ensured that other global procedures will not see *P*'s updates unless it has finished or compensation has been requested for the updates.

5. Conclusions

We have looked at two general ways of synchronizing global procedures in a federated database system, using a number of examples. Altruistic locking at the global level guarantees the serializability of global procedures, while allowing more concurrency than global two-phase locking.

If serializable execution of global procedures is not important to the application and the procedures can be broken up into a collection of transactions, then sagas can be used. Sagas provide semantic atomicity but do not serialize the execution of global procedures. Sagas require no global locking and permit more concurrency than mechanisms that treat global procedures as a single unit.

Because global procedures can affect local resources only through a transactional interface, it is not possible to make them completely atomic. However, compensating transactions can be used to provide a semantic atomicity guarantee for global procedures.

References

Garc83a.

Garcia-Molina, Hector, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems*, vol. 8, no. 2, pp. 186-213, June 1983.

Garc87a.

Garcia-Molina, Hector and Kenneth Salem, "Sagas," *Proc. ACM SIGMOD Annual Conference*, pp. 249-259, San Francisco, CA, May, 1987.

Heim85a.

Heimbigner, Dennis and Dennis McLeod, "A Federated Architecture for Information Management," *ACM Transactions on Office Information Systems*, vol. 3, no. 3, pp. 253-278, July, 1985.

Sale87a.

Salem, Kenneth, Hector Garcia-Molina, and Rafael Alonso, "Altruistic Locking: A Stratagy for Coping with Long-Lived Transactions," CS-TR-087-87, Dept. of Computer Science, Princeton University, April, 1987.

AN UPDATE MECHANISM FOR MULTIDATABASE SYSTEMS

Yuri Breitbart
Computer Science Dept.
University of Kentucky
Lexington, KY 40506

Avi Silberschatz
Computer Science Dept.
University of Texas
Austin, TX 78712

Glenn Thompson
Amoco Production Co.
Research Center
P.O. Box 3385
Tulsa, OK 74102

1. Introduction

A multidatabase system (MDBS) consists of one or more databases, possibly distributed, which are controlled by one or more database management systems (DBMSs). An MDBS creates the illusion of logical database integration without requiring physical integration of the databases.

Recently, a large amount of research has been conducted in the area of multidatabase systems (e.g. [LAND82], [LITW82], and [BREI85]). However, the problem of updating semantically related data located in preexisting databases has not been sufficiently addressed. In [PU86], the update problem is addressed under the assumption that the MDBS is aware of local transactions. Making the MDBS aware of local transactions requires changes to the local concurrency control mechanisms to enable the local DBMSs to report local transaction execution information to the MDBS, which uses this information for local and global transaction synchronization. Introducing such changes allows any two DBMSs to communicate with each other and, therefore, reduces the multidatabase concurrency control problem to the concurrency control problem in homogeneous distributed database management systems (DDBMS).

Another assumption that is frequently made is that retrieve-only multidatabase systems do not require a concurrency control mechanism, since the probability of inconsistent retrievals in the presence of local transactions is quite low [LAND82]. In [BREI87a], we show that while the probability of inconsistent retrieval may be low, it may still occur.

In this paper, we propose an update mechanism that allows the MDBS to update semantically related data items while retaining global database consistency in the presence of local transactions. This approach is being implemented in the Amoco Distributed Database System (ADDS) [BREI85], [BREI86].

The ADDS update model is based on the principles of retaining local database autonomy and disallowing changes in the local DBMS software to accommodate ADDS. Modifying the DBMSs to interact with the MDBS requires significant development effort when support for a new DBMS is added. These changes may also create difficult problems, both in maintaining current applications and in maintaining the DBMS software.

Since changes to the local database software are not permitted, the DBMSs treat the global subtransactions and the local transactions equally. Also, the local DBMSs should perform their operations without the knowledge of other DBMSs and the MDBS. Therefore, local autonomy requirements make the update problem in a multidatabase system significantly different from the update problem in a homogeneous DDBMS.

If a global database contains replicated data, the copies of the data should not be updated by local transactions. Consider, for example, a global database that contains global data item x which has a copy at site A and site B. If a local transaction is submitted at site A that changes the value of x, the global database becomes inconsistent, since the value of x is no longer the same at both sites.

In the ADDS system we allow local transactions to read local data items and write non-replicated data items. In [BREI87b], we prove that allowing these local transactions does not compromise global database consistency.

2. Multidatabase Concurrency Control Problems

There are several inherent difficulties in solving the multidatabase update problem in the presence of local transactions [GLIG85].

1. Generating and executing subtransactions based on the global transactions submitted to the MDBS.
2. Maintaining global transaction and subtransaction atomicity.
3. Preserving the relative execution order, determined by the MDBS, of the subtransactions at the local sites.
4. Detecting and recovering from or preventing global deadlocks.

To maintain global database consistency in the presence of local transactions, it is sufficient to ensure the same global transaction execution order at each site. In [GLIG85], this condition was stated only in regard to global transactions that have conflicting operations. However, even in the absence of conflicts among global transactions, the execution order of the subtransactions must be the same at each site. In [BREI87b], we define a model of the multidatabase update problem and prove that maintaining the execution order of the global transactions at each site ensures global database consistency.

Another problem in the design of an MDBS system is the avoidance of undetectable global deadlocks [GLIG85]. The next example illustrates a serializable execution of global and local transactions that generates an undetectable global deadlock.

Example 1

Consider a global database consisting of data items a and b at site 1 and c and d at site 2. Let T_1 and T_2 be global transactions and L_3 and L_4 be local transactions at sites 1 and 2, respectively.

T_1: $r_1(a)\ w_1(a)\ r_1(c)\ w_1(c)$ T_2: $r_2(b)\ w_2(b)\ r_2(d)\ w_2(d)$
L_3: $r_3(a)\ r_3(b)$ L_4: $r_4(c)\ r_4(d)$

Consider the following local schedules at sites 1 and 2, respectively.

S_1: $r_3(a)\ r_1(a)\ w_1(a)\ r_2(b)\ w_2(b)\ r_3(b)$
S_2: $r_4(c)\ r_1(c)\ w_1(c)\ r_2(d)\ w_2(d)\ r_4(d)$

Both schedules are serializable and their combined execution retains global database consistency. However, an undetectable deadlock may occur during execution as follows.

At site 1, T_1 waits for item a that is locked by L_3. L_3 waits for item b that is locked by T_2. T_2, in turn, is waiting for a message from the MDBS that it may proceed. However, the MDBS is waiting to receive a message from T_1 at site 2 that it has completed, since the MDBS is trying to synchronize the execution of T_1 and T_2 at the global level. Also, at site 2, T_1 waits for item c that is locked by L_4. L_4 waits for item d that is locked by T_2 and T_2 is waiting for a message from the MDBS that it may proceed at site 2. Figure 1 illustrates the deadlock situation described above. □

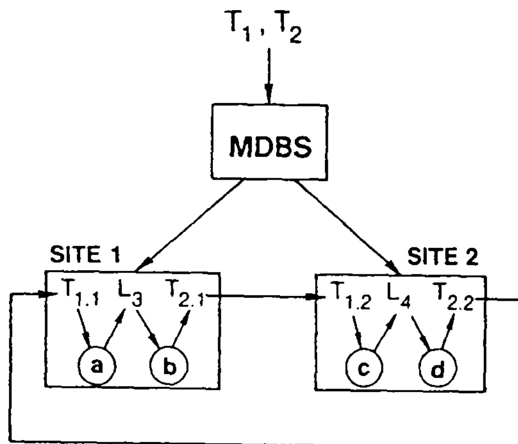


Figure 1. Undetectable Deadlock

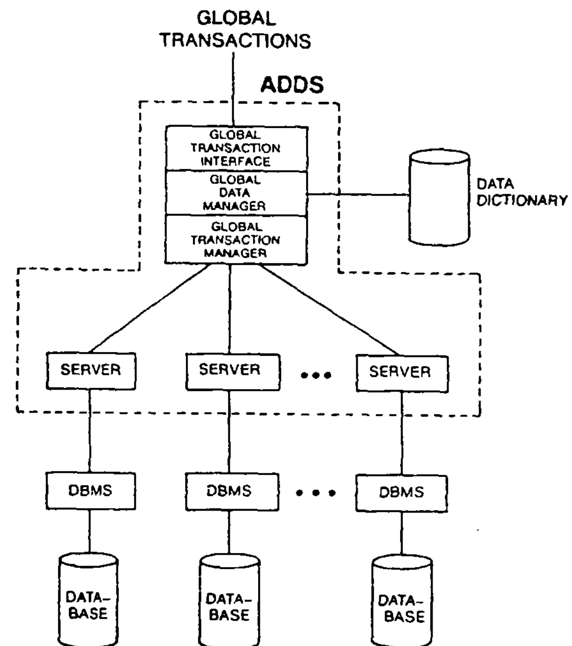


Figure 2. ADDS Architecture

The reason for the undetectable deadlock is that the MDBS is attempting to synchronize the execution of T_1 and T_2, in that order. However, the local DBMSs reverse the order unbeknownst to the MDBS. The reversal does not destroy global database consistency. However, it creates a deadlock that must be resolved by some method other than simply maintaining the execution order of the global transactions at the local sites.

3. General Architecture of ADDS

The ADDS system provides uniform access to preexisting heterogeneous distributed databases. ADDS uses a relational data model and an extended relational algebra query language to provide access to distributed data. The local database schemas are mapped into a relational global database schema as described in [BREI86] and the mappings are stored in the ADDS data dictionary. The data dictionary also contains the physical characteristics and location of the local data. The only communication between ADDS and the local DBMSs is in the form of query submission and data retrieval. ADDS requires that each of the local DBMSs utilize some sort of a concurrency control that maintains local database consistency.

Figure 2 illustrates the layered architecture of the ADDS system. Global transactions are considered to be processing programs. The global transaction interface (GTI) receives user transactions, ensures their syntactical correctness and generates a global execution plan.

The global data manager (GDM) uses the data dictionary to determine the location or locations of the data referenced by global transactions. The GDM is also responsible for managing all intermediate data that is received from the global transaction manager during transaction execution.

The global transaction manager (GTM) manages the execution of the global transactions. The GTM allocates a server to a global transaction to process read and write operations for data controlled by a single DBMS.

The servers translate global read and write operations into the languages of the local DBMSs. The servers also transfer retrieved data to the GTM. The GTM allocates one server to a global transaction for each of the sites referenced by the transaction. A server allocated to a transaction is not released until the transaction has completed execution at each site and the results of the transaction have been committed by the MDBS.

As global operations are received, the GTM sends the global operations to the appropriate servers. If a server is not allocated to the current global transaction for a particular site, the GTM allocates a server to the transaction and passes the global operation to the appropriate servers for execution.

When a global transaction completes execution, the GTM instructs the servers allocated to the transaction, to commit the updates to the local databases. ADDS uses a two-phase commit protocol in communication with the servers to commit the results of a transaction. ADDS does not require any specific commit protocol to be supported by the local DBMSs and assumes that any local DBMS is capable of committing the results of local transactions. If a global transaction wishes to abort, the GTM instructs the servers to rollback the updates to the local databases.

The current version of the ADDS system is implemented under the IBM VM/SP operating system. The local databases supported include IMS, SQL/DS,

INQUIRE, RIM, and FOCUS. Communication with the local DBMSs is accomplished using the SNA and ETHERNET networks. The current ADDS network nodes include geographically distributed mainframes containing complete ADDS systems, as well as, workstations (e.g., Sun and Apollo) containing only ADDS user interface software and connected by local area networks.

4. ADDS Update Algorithm

The notion of a site graph [BREI87b], [THOM87] is central to our discussion of the ADDS update algorithm. We create a site graph of a global transaction T by first determining the sites that contain copies of the global data items referenced by T and connecting them as nodes in a graph that has exactly one path between any two nodes. The nodes of the graph are connected by undirected edges labeled with the transaction name T. The edges of the graph do not carry any special meaning and are arbitrarily chosen by the MDBS.

Given a set of global transactions, if we combine a site graph for each of the transactions into a single graph, we obtain a site graph for the system of global transactions. The next example illustrates the notion of a site graph.

Example 2

Consider a global database that contains data item x at sites 1 and 2, y at sites 1 and 3, and z at sites 2 and 3. Global transactions T_1 and T_2 are defined in the following way.

$$T_1: r_1(x)\ w_1(y) \qquad T_2: r_2(y)\ w_2(z)$$

The GTM may generate one of the following sequences of local operations for each transaction.

$$T_1: r_1(x_1)\ w_1(y_1)\ w_1(y_3) \qquad T_2: r_2(y_3)\ w_2(z_2)\ w_2(z_3)$$

or

$$T_1: r_1(x_1)\ w_1(y_1)\ w_1(y_3) \qquad T_2: r_2(y_1)\ w_2(z_2)\ w_2(z_3)$$

The site graphs of T_1 and T_2 for each site selection are shown in Figures 3a and 3b, respectively. □

The existence of a cycle in the site graph of a system of global transactions may cause global database inconsistency during the execution of read and write operations of global and permitted local transactions. On the other hand, in the absence of cycles in the site graph, ADDS guarantees the correct execution of any mix of global transactions and permitted local transactions and also guarantees the absence of global deadlocks.

The technique used by ADDS to process read and write operations for a system of global transactions is described below. For all operations, the GDM uses the data dictionary to determine the sites that contain a copy of

the referenced data item. Upon receiving a read operation, the GTM selects a site that does not create a cycle in the site graph. The GTM site selection algorithm is described below.

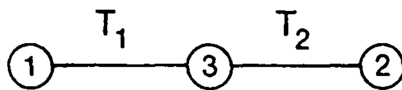


Figure 3a. Site Graph With No Cycles

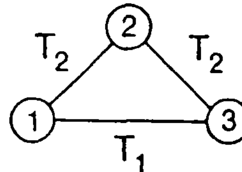


Figure 3b. Site Graph With a Cycle

If the transaction has already processed data at a site that contains a copy of the data item, the site is selected to perform the read operation. However, if the transaction has not processed any data at a site that contains a copy of the data item, all sites that contain a copy of the data item must be examined individually. If there exists a site that contains a copy of the data item and the addition of the site to the site graph does not create a cycle, the site is selected for the execution of the read operation. A new server for this site is allocated to the transaction. This server will then process all data items that are located at the specified site for the transaction. If no site containing the data item may be added to the site graph without creating a cycle, the transaction is rolled back and later restarted.

Upon receiving a write operation, the GTM adds all the sites that contain a copy of the data item to the site graph. If the addition of these sites to the site graph does not create a cycle, the write operation proceeds. If any of the sites that contain the data item do not have servers allocated to the transaction, the GTM allocates the required servers and sends the write operation to the servers for execution. If the addition of the sites that contain the data item creates a cycle in the site graph, the transaction is rolled back and later restarted.

After a transaction has committed or aborted, all edges labeled with the transaction are removed from the site graph. A correctness proof for the above algorithm appears in [BREI87b]. Careful analysis of the algorithm shows that it solves all of the multidatabase concurrency control problems mentioned in Section 2. The MDBS, by careful distribution of global operations to the local sites, ensures global database consistency without any additional information from the local DBMS concurrency control mechanisms.

The algorithm permits concurrent execution of global and local transactions. However, the level of concurrency for global transactions in this environment may be less than the level of concurrency for global transac-

tions in the absence of local transactions. In our view, the reduction in the level of concurrency is a small price to be paid for retaining local autonomy in a multidatabase system. The nature of the cost will be determined by a performance evaluation of the algorithm, which is currently being conducted.

References

[BREI85] Breitbart, Y. and L. Tieman. "ADDS - Heterogeneous Distributed Database System." Distributed Data Sharing Systems. Eds. F. Schreiber and W. Litwin. North Holland, 1985, 7-24.

[BREI86] Breitbart, Y., P. Olson, and G. Thompson. "Database Integration in a Distributed Heterogeneous Database System." Proceedings of the International Conference on Data Engineering, 1986, 301-310.

[BREI87a] Breitbart, Y., A. Silberschatz, and G. Thompson. "An Approach to the Update Problem in Multidatabase Systems." Amoco Production Company Research Technical Report, F87-C-11, Tulsa, OK, 1987.

[BREI87b] Breitbart, Y., A. Silberschatz, and G. Thompson. "Concurrency Control in a Heterogeneous Distributed Database System," 1987 (in preparation).

[GLIG85] Gligor, V. and R. Popescu-Zeletin. "Concurrency Control Issues in Distributed Heterogeneous Database Management Systems." Distributed Data Sharing Systems. Eds. F. Schreiber and W. Litwin. North-Holland, 1985, 43-56.

[LAND82] Landers, T. and R. Rosenberg. "An Overview of Multibase." Distributed Data Systems. Ed. H. Schneider. North-Holland, 1982, 153-184.

[LITW82] Litwin W., J. Boudenant, C. Esculier, A. Ferrier, A. Glorieux, J. La Chimia, K. Kabbaj, C. Moulinoux, P. Rolin, and C. Stangret. "SIRIUS Systems for Distributed Data Management." Distributed Data Bases. Ed. H. J. Schneider. North-Holland, 1982, 311-366.

[PU86] Pu, C. "Superdatabases for Composition of Heterogeneous Databases." Columbia University Technical Report No. CUCS-243-86, 1986.

[THOM87] Thompson, G. "Multidatabase Concurrency Control." (Ph.D. dissertation in preparation, Oklahoma State University, 1987.)

Superdatabases: Transactions Across Database Boundaries

Calton Pu
Department of Computer Science
Columbia University
New York, NY 10027

1 Introduction

For both efficiency and extensibility, integrated and consistent access to a set of heterogeneous databases is desirable. However, current commercial databases running on mainframes are, by and large, centralized systems. Physical distribution of data in distributed homogeneous databases has been demonstrated in several systems, such as Distributed INGRES [Ston79] and R* [LHM*84]. Nevertheless, the research on integrated heterogeneous databases has been limited to query-only systems such as MULTIBASE [LR82] and MERMAID [TBH*83].

In contrast to the relative success of research on query processing and optimization over heterogeneous databases, few results have been reported on consistent *update* across heterogeneous databases. Our answer to this challenge is the building of *superdatabases*. Unlike earlier works on uniform query access through a single language, our emphasis centers on consistent update across heterogeneous databases. A superdatabase is conceptually a hierarchical composition of element databases, which may be centralized, distributed, or other superdatabases. A supertransaction running in a superdatabase is composed of component transactions from the element databases in the same way standard nested transactions [Moss81] form a top-level transaction. Therefore, results from the component transactions remain invisible (uncommitted) until the supertransaction commit.

Update support in homogeneous databases relies on two sets of fundamental techniques: concurrency control and crash recovery. We propose the construction of a superdatabase through hierarchical composition of concurrency control and crash recovery. Starting from Reed [Reed78], many years of research on nested transactions have produced several particular implementations of nested transactions. Generalizing an earlier work [Pu86], which used hierarchical composition to implement a nested transaction mechanism within a single database, we apply hierarchical composition across database boundaries.

In Section 2 we summarize the general architecture of superdatabases. In Section 3 we describe some sufficient conditions to make element databases composable. In Section 4, we explain the design of a superdatabase capable of gluing the element databases together. Section 5 sketches an implementation plan. Finally, Section 6 concludes the paper.

2 Hierarchical Composition

The superdatabase composes element databases hierarchically. In figure 1, DB_j represent different element databases glued together by superdatabases. An atomic transaction spanning several element databases is called a *supertransaction*. When participating in a supertransaction, the local transaction on each element database is called a subtransaction. For simplicity of presentation, we make the standard assumption [GP85] that there is only one subtransaction per element database for each supertransaction.

We divide this hierarchical composition into two parts, the element databases at the leaves and the superdatabase as the internal nodes. In this section, we summarize the ideas. The conditions an element database must satisfy so the superdatabase can handle it are described in detail in section 3. The design of the the superdatabase to connect composable element databases is detailed in section 4.

An element database is said to be *composable* if it satisfies two requirements. The first is on crash recovery: the element database must understand some kind of agreement protocol, for example, two-phase

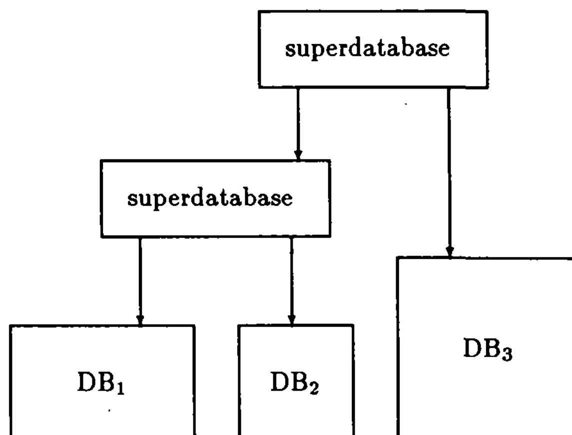


Figure 1: The Structure of Superdatabases

commit. Otherwise, one element database may decide to commit its component transaction, while another decides to abort. This necessary requirement is a consequence of distributed control, not heterogeneity.

The second requirement is on concurrency control: the element database should present an explicit serial ordering of its local transactions. All major concurrency control methods (two-phase locking, timestamps, and optimistic concurrency control) provide an easy way to capture the serial order they impose on the transactions. Furthermore, since any agreement protocol implies communication between participants, passing the explicit serial order of subtransactions (local to each element database) may piggyback on these messages, reducing the performance impact of the second requirement.

For consistent updates, these two are the only requirements we make on the element databases. An element database may be centralized, distributed, or as we shall see, another superdatabase. Unfortunately, in practice most centralized databases do not support any kind of agreement protocol. Similarly, most distributed databases do not supply the transaction serial order. Consequently, our results cannot be applied directly to existing databases without modification. Nevertheless, we believe that these relatively mild requirements, once identified, can be feasibly incorporated into current and future database systems. The pay-off is significant: extensibility and accommodation of heterogeneity.

We have three design goals for the superdatabase that glue the composable element databases together.

1. Composition of element databases with many kinds of crash recovery methods.

2. Composition of element databases with many kinds of concurrency control techniques.

3. Recursive composibility; i.e. the superdatabase must satisfy the requirements of an element database.

The realization that we need only an agreement protocol for crash recovery made the first goal easy. The key idea that achieved the second goal is to use the explicit serial ordering of transactions, the common denominator of best known concurrency control methods. The third goal was accomplished through careful design of the agreement protocol and explicit passing of the serial order.

3 Element Databases

The usual model of a distributed transaction contains a coordinator and a set of subtransactions. Each subtransaction maintains its local undo/redo information. At transaction commit time, the coordinator organizes some kind of agreement with subtransactions to reach a uniform decision. The two-phase commit protocol is the most commonly used because of its low message overhead.

The distributed database system R* [LHM*84] provides a tree-structured computation, which refines the above flat coordinator/subtransactions model. Subtransactions in R* are organized in a hierarchy, and the two-phase commit protocol is propagated down the tree structure. The transaction commits only if all subtransactions in the tree vote for commitment. Since heterogeneous databases are distributed by nature, it is necessary that each element database maintains the undo/redo information locally. In addition, it is necessary that each element database understands some kind of agreement protocol, such as the two-phase commit outlined above, three-phase commit, and the various flavors of Byzantine agreements.

On concurrency control, we assume the element databases maintain serializability of local transactions. The question is whether the superdatabase can maintain global serializability given local serializability. The answer is yes, if the superdatabase certifies that all local serial orders are compatible in a global serial order. One way to implement the superdatabase certification is to require that each element database provide the ordering of its local transactions to the superdatabase. Please note that this assumption provides a sufficient condition for composition of heterogeneous databases, but it is not necessary, since implicit serialization is possible under some circumstances (section 5). The serial order of each local transaction is represented by an *order-element*, or O-element for short. In Section 4, we shall describe the composition of O-elements for certification. Here, we only discuss how the concurrency control methods produce the O-elements.

First, we consider element databases with two-phase locking concurrency control. Eswaran et al. [EGLT76] showed that two-phase locking guarantees serializability of transactions because $SHRINK(T_i)$, the timestamp of transaction T_i's lock point, indicates T_i's place in the serialization. We take advantage of this fact and designate $SHRINK(T_i)$ as the O-element for element databases with two-phase locking.

The second most popular concurrency control method uses timestamps for serialization. Since transactions serialized by timestamps have their serialization order explicitly represented in their timestamps, these serve well as O-elements. Timestamp intervals [BEHR82] or multidimensional timestamps [LB86] can be passed as O-elements as well.

As another alternative, optimistic concurrency control methods also provide an explicit serialization order. Kung and Robinson [KR81] assign a serial transaction number after the write phase, which can be used directly as O-element. Ceri and Owicki [CO82] proposed a distributed algorithm in which a two-phase commit follows a successful validation. Taking a timestamp from a Lamport-style global clock [Lamp78] at that moment will capture the serial order of transactions.

There is no constraint on the format of the O-element. Each element database may have its own representation. We only require that two O-elements from the same element database be comparable, and that this comparison recover the serialization guaranteed by local concurrency control methods. More formally, let the serialization produced by the concurrency control method be represented by the binary relation *precede* (denoted by $\leq$). We require that O-element(T_1) $\leq$ O-element(T_2) if $T_1 \leq T_2$ in the local serialization.

In summary, for crash recovery any distributed database satisfies our requirements, namely, local redo/undo capability and some kind of agreement protocol. For concurrency control, we ask that the serialization order is made explicit through O-elements for the superdatabase described in section 4. Since explicit serialization is sufficient but not necessary, weaker requirements are possible.

4 Superdatabase Design

The superdatabase has two main components: crash recovery and concurrency control. Since it contains no user data, all stored in element databases, both components are only concerned with control information on subtransactions. Three main components form the superdatabase: the commit protocol for a supertransaction; the recovery from superdatabase crash; and the serialization of supertransactions. Due to space constraints, we summarize the main results in this paper, and interested readers should consult another paper [Pu87] for more details.

Given that some form of agreement is necessary (section 3), the question is whether it is sufficient for hierarchical commit. In R*, two-phase commit implements hierarchical commit. Two-phase commit

protocol obtains agreement on the outcome of the transaction independently of recovery information to undo/redo updates. So does all the other agreement protocols, such as three-phase commit and Byzantine agreements. The important thing is that for each element database, the superdatabase must understand and use the appropriate protocol.

Since the superdatabase is the coordinator for the element databases during commit, it must record the transaction on stable storage. Otherwise, a crash during the window of vulnerability would hold resources in the element databases indefinitely. Of the known recovery methods, logging is the best for superdatabase recovery. Since no before-images or after-images need to be saved, versions are of little utility. Conceptually, the superdatabase log is separate from the element database logs, just as the superdatabase itself. In actual implementation, the superdatabase log may be physically interleaved with an element database log, as long as the recovery algorithms can separate them later.

For each transaction, the superdatabase saves the following information on the log: participant subtransactions; parent superdatabase, if any; and the transaction state (prepared, committed, or aborted). The transaction state is written to the log during the agreement protocol. If a transaction was in the active state when the superdatabase crashed, the superdatabase simply waits for (re)transmission of two-phase commit from the parent. In case it is the root, it (re)starts the two-phase commit. If a transaction was in the prepared state when the superdatabase crashed, the superdatabase inquires the parent about the outcome of the transaction. If the transaction has been committed, the results are retransmitted to the subtransactions.

Compared to crash recovery, superdatabase concurrency control is more elaborate. The main problem that the superdatabase has to detect is when subtransactions from different element databases were serialized in different ways. In the following example, this happens when a second transaction T_2 with the same subtransactions produces the ordering: O-element($T_{1.1}$) $\leq$ O-element($T_{2.1}$) and O-element($T_{2.2}$) $\leq$ O-element($T_{1.2}$).

```
BeginTransaction(Top-level, T1)
    cobegin
        DB1.BeginTransaction(T1, T1.1) ... actions ... CommitTransaction(T1.1)
        DB2.BeginTransaction(T1, T1.2) ... actions ... CommitTransaction(T1.2)
    coend
CommitTransaction
```

To prevent this kind of disagreement from happening, we define an order-vector (O-vector) as the concatenation of all O-elements of the supertransaction. In the example, O-vector(T_1) is (O-element($T_{1.1}$), O-element($T_{1.2}$)). The order induced on O-vectors by the O-elements is defined strictly: O-vector(T_1) $\leq$ O-vector(T_2) if and only if for all element database j, O-element($T_{1.j}$) $\leq$ O-element($T_{2.j}$). If a supertransaction is not running on all element databases, we use a wild-card O-element, denoted by $*$ (star), to fill in for the missing element databases. Since its order does not matter, by definition, O-element(any) $\leq *$, and, $* \leq$ O-element(any).

From this definition, if O-vector(T_1) $\leq$ O-vector(T_2) then all subtransactions are serialized in the same order, ordering the supertransactions. Therefore, we can serialize the supertransactions by checking the O-elements of a committing supertransaction against the history of all committed supertransactions. If the new O-vector can find a place in the total order, it may commit.

The comparison with all committed supertransactions may be expensive, both in terms of storage and processing. Fortunately, it is not necessary to compare the O-vector with all committed supertransactions. Since a transaction trying to commit cannot be serialized in the ancient history, it is sufficient to certify the transaction with a reasonably "recent history" of committed supertransactions. To determine the beginning of active history, the superdatabase asks the element databases to send in the most ancient active transaction id and subtracts the duration of the longest possible supertransaction. In practice, an estimated time-out period based on the applications probably suffices.

From the composition point of view, the key observation is that the certification based on O-vectors is independent of particular concurrency control methods used by the element databases. Therefore, a

superdatabase can compose two-phase locking, timestamps, and optimistic concurrency control methods. As long as we can make the serialization in element databases explicit, the superdatabase can certify the serializability of supertransactions. More importantly, the certification gives the superdatabase itself an explicit serial order (the O-vector) allowing it to be recursively composed as an element database. Thus we have found a way to hierarchically compose database concurrency control, maintaining serializability at each level.

The certification method is optimistic, in the sense that it allows the element databases to run to completion and then certifies the serial ordering. In particular, the O-vector is constructed only after the subtransactions have attempted to commit. Since some concurrency control techniques (such as time-interval based and optimistic) decide the transaction ordering only at the transaction commit time, it is difficult for the superdatabase to impose an ordering during the execution of the optimistic subtransactions. In other words, the superdatabase has to be as optimistic as its element databases.

5 Run-time Overhead and Performance

The main piece of information the element databases give to the superdatabase is the O-element. With some concurrency control methods, such as timestamps, the production of the O-element is trivial. If the element database is centralized, then the cost of taking a timestamp is also low. However, if the element database is a distributed database with internal concurrency control, then a global clock will be necessary to capture the serial order. Fortunately, the maintenance of a global clock is independent of the number of transactions, and therefore can be amortized.

Another run-time overhead is the message containing the O-element. Since we have demonstrated the necessity of an agreement protocol for recovery purposes, at least one message must be exchanged between the superdatabase and each element database at commit time. The certification occurs only at commit time, so the subtransaction serial order information can piggyback on the commit vote message. Therefore, the superdatabase does not introduce any extra message overhead during transaction processing.

Although in principle a superdatabase must check the serializability of all subtransactions, there are important cases that permit optimization. For example, if all element databases use strict two-phase locking, the lock points of the subtransactions are synchronized by the commit protocol, and no certification will be necessary among them. However, the certification algorithm must be used between the group of strict two-phase locking databases and others such as general two-phase locking, timestamps, and optimistic methods.

The superdatabase design using O-vectors in section 4 receives the explicit serialization order from the element databases. Unlike other approaches [GP85] which pass information about local objects, we use the local concurrency control to condense synchronization information and reduce message flow. We have shown that the local total ordering is sufficient for global serialization. However, less information results in less concurrency. Unrelated supertransactions may cause unnecessary aborts due to the their appearance of conflicting serialization from different element databases. Fortunately we can use our knowledge of the concurrency control methods themselves to increase transaction concurrency in the superdatabase. Two examples are two-phase locking and timestamps.

To avoid unnecessary aborts among element databases using general two-phase locking, we only need to synchronize the lock points of participating component transactions explicitly. For example, a supertransaction can use two-phase agreement to synchronize the lock points. This synchronization makes the supertransaction two-phase globally, so all element databases now agree on the serialization of the component transactions. However, we still have to certify the group of two-phase locking component transactions with others synchronized through different concurrency control methods.

Timestamp-based element databases also can provide the superdatabase with additional information. For example, time-interval based concurrency control methods allow the superdatabase to serialize some transactions that would have been aborted in the minimal design.

6 Conclusion

We have described the design of superdatabases and the algorithms used to compose consistent databases out of both homogeneous and heterogeneous elements. There are four good characteristics in the superdatabase approach to building heterogeneous databases.

First, superdatabases guarantee the atomicity of global updates across the element databases. This atomicity includes both reliability atomicity through an agreement protocol, such as two-phase commit, and concurrency atomicity through the certification of serialization provided by the element databases.

Second, the design of superdatabase is adaptable to a variety of crash recovery methods and concurrency control techniques used in the element databases. We have established the necessity for an agreement protocol for supertransaction commit. However, the protocol is independent of particular crash recovery methods used to undo and redo local transactions in the element databases. We have also shown that as long as the element databases use concurrency control methods which easily supply an explicit serial order of their transactions, they can be included under the superdatabase.

Third, databases built with superdatabases are extensible by construction. Element databases may be added or deleted without changing the superdatabase. In additon, many interesting applications can take advantage of the extensibility. For example, a replicated database can be constructed by connecting two identical element databases with a superdatabase. Another example is that given a database X, satisfying the requirements of section 3 for crash recovery and concurrency control, a superdatabase delivers the distributed version of X.

Fourth, transactions local to element databases run independently of the superdatabase, which intervenes only when needed for synchronization or recovery of supertransactions across different element databases. In other words, the additional overhead introduced by the indirection through superdatabase is paid only by the direct users of its services. The only interference happens when a component transaction of a supertransaction conflicts with a local transaction.

Even though we described the serialization of supertransactions using O-vectors, the hierarchical approach admits other methods that explore the properties of particular concurrency control methods. For example, using an agreement to synchronize lock points of two-phase locking elements databases and distributing global timestamps to timestamp-based element databases are techniques that may improve the concurrency in the superdatabase.

Global deadlock detection and resolution remains a research challenge, since it is immune to hierarchical approaches. Observing that the time-out mechanism is inherent in distributed systems, we expect it to be useful in avoiding deadlocks.

Many years of research on heterogeneous databases have achieved impressive and substantial progress, especially in query language translation and view integration. We hope the combination of our results with previous work on heterogeneous databases will produce superdatabases which are consistent, adaptable, and extensible.

References

[BEHR82] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser.
Dynamic timestamp allocation for transactions in database systems.
In H. J. Schneider, editor, *Distributed Data Bases*, North-Holland, 1982.

[CO82] S. Ceri and S. Owicki.
On the use of optimistic methods for concurrency control in distributed databases.
In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–129, Lawrence Berkeley Laboratory, University of California, Berkeley, February 1982.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The notions of consistency and predicate locks in a database system.

Communications of ACM, 19(11):624–633, November 1976.

[GP85] V. Gligor and R. Popescu-Zeletin.
Concurrency control issues in distributed heterogeneous database management systems.
In F.A. Schreiber and W. Litwin, editors, *Distributed Data Sharing Systems*, pages 43–56, North Holland Publishing Company, 1985.
Proceedings of the International Symposium on Distributed Data Sharing Systems.

[KR81] H. T. Kung and John T. Robinson.
On optimistic methods for concurrency control.
Transactions on Database Systems, 6(2):213–226, June 1981.

[Lamp78] L. Lamport.
Time, clocks and ordering of events in a distributed system.
Communications of ACM, 21(7):558–565, July 1978.

[LB86] P.J. Leu and B. Bhargava.
Multidimensional timestamp protocols for concurrency control.
In *Proceedings of the Second International Conference on Data Engineering*, pages 482–489, Los Angeles, February 1986.

[LHM*84] B. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost.
Computation and communication in R*: a distributed database manager.
ACM Transactions on Computer Systems, 2(1):24–38, February 1984.

[LR82] T. Landers and R.L. Rosenberg.
An overview of MULTIBASE.
In H.J. Schneider, editor, *Distributed Data Bases*, North Holland Publishing Company, September 1982.
Proceedings of the Second International Symposium on Distributed Data Bases.

[Moss81] J.E.B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, Massachusetts Institute of Technology, April 1981.

[Pu86] Calton Pu.
Replication and Nested Transactions in the Eden Distributed System.
PhD thesis, Department of Computer Science, University of Washington, 1986.

[Pu87] C. Pu.
Superdatabases for Composition of Heterogeneous Databases.
Technical Report CUCS-243-86, Department of Computer Science, Columbia University, June 1987.

[Reed78] D.P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, Massachusetts Institute of Technology, September 1978.

[Ston79] M. Stonebraker.
Concurrency control and consistency of multiple copies of data in Distributed INGRES.
IEEE Transactions on Software Engineering, SE-5(3):188–194, May 1979.

[TBH*83] M. Templeton, D. Brill, A. Hwang, I. Kameny, and E. Lund.
An overview of the MERMAID system – a frontend to heterogeneous databases.
In *Proceedings of EASCON 1983*, pages 387–402, IEEE/Computer Society, 1983.

An Optimistic Concurrency Control Algorithm for Heterogeneous Distributed Database Systems

Ahmed K. Elmagarmid and Yungho Leu
Computer Engineering Program
121 Electrical Engineering East Bldg.
Pennsylvania States University
University Park, PA 16802
(814) 863-1047

1. Introduction

A heterogeneous distributed database management system (HDDBS) is a software layer which interconnects existing DBMSs to facilitate the access of data across DBMSs. The DBMSs may differ in data models, data definition and manipulation languages, transaction management (including concurrency control and commit protocols), and internal data structures [GLIG86].

This paper presents a global concurrency control algorithm and a global commit protocol to support consistent updates in heterogeneous database systems. This paper is organized as follows. A transaction processing model of HDDBS is given in section 2. In section 3, basic assumptions and global serializability are discussed. The concurrency control algorithm and the commit protocol are given in section 4.

2. Transaction Processing Model for Heterogeneous Distributed Database management systems

In this section, a model for the proposed concurrency control algorithm is presented. As shown in figure 1, the transaction management model for heterogeneous distributed DBMSs consists of global and local transaction management levels[GLIG86]. A global transaction is performed in two modules, a Global Data Manager (GDM) and a Global Transaction Manager (GTM). Global transactions reference data at more than one site in the system. A global transaction when submitted by the user is decomposed by the GDM into a set of subtransactions which access data in the local site on behalf of the global transaction. The set of subtransactions in turn are transmitted to the appropriate local sites to be executed. When the results of subtransactions come back to the GDM, the GDM then integrates the results and presents the final result to the user. The functions of GDM include (1) global data model analysis, (2) query decomposition, (3) query translation, (4) execution plan generation and (5) result integration. The purpose of the GTM is to control the execution of subtransactions to help ensure the consistency of the database. To be more specific, the GTM must extend the functions of the local transaction managers (LTMs) to maintain the global consistency of the database. The GTM ensures the global consistency of the database by maintaining the serializability of the execution of global transactions even when

global transactions are executed concurrently. The other function of the GTM is to provide failure atomicity. Failure atomicity means that a global transaction behaves as though it executes to completion or not at all. That is, if the system crashes in the middle of executing a transaction, intermediate results will not be left in the database.

In figure 1, T_L represents a local transaction executed in a local site. T_i^k represents the subtransaction of a global transaction T_i at site k. For simplicity, we assume that a global transaction may have at most one subtransaction per site.

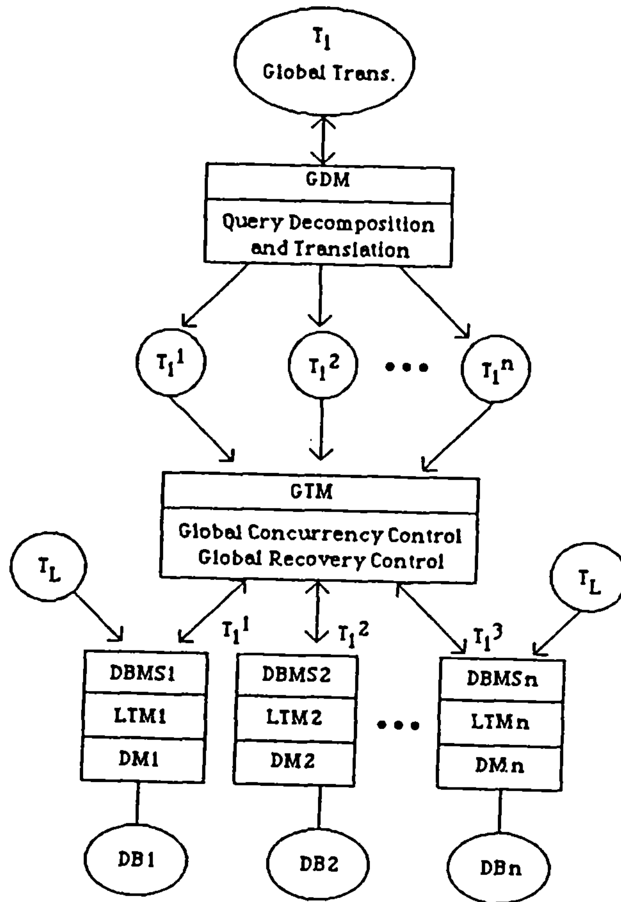


Figure 1. The Transaction Model of HDDBS

3. Basic Assumptions and Global Serializability

Transactions consist of a sequence of primitive operations which can be either read or write. Two primitive operations are said to be conflicting if (1) they belong to different transactions;(2) both access the same database item; and (3) at least one of them is a write operation. Three types of transactions are present in HDDBS. They are global transactions, subtransactions, and local transactions. A global transaction consists of one or more subtransactions. Subtransactions access

identifier of a subtransaction.

For two-phase locking protocols, the time at which a subtransaction releases its first lock can be used as the subtransaction serial order. The subtransaction timestamp and the time when a subtransaction is validated can both be used as serial orders for algorithms based on timestamp ordering and optimistic approach respectively. A detailed discussion of how serial orders of transactions can be derived from existing concurrency control algorithms is given in [PU86].

4.2 Optimistic Approach to Global Concurrency Control

When all of the subtransactions of a global transaction have been executed, the GTM starts validating this global transaction against a set of recently committed global transactions. If the transaction that is attempting to commit is serializable with all of the recently committed global transactions, then this global transaction is committed, otherwise it is aborted.

For the purpose of validation, the GTM must keep two data structures for the recently committed global transactions, the Serial Order Array (SOA) and the Read-Write Set Array(RWSA). The SOA contains serial orders of subtransactions of the recently committed global transactions. For example, if there are three local sites and three committed global transactions, then a possible SOA is shown in figure 2. Where τ_{13} is the serial order of T_1^3 (subtransaction of T_1 at site 3), and so on. The '*' means unspecified, this happens when the global transaction has no subtransaction at the corresponding site. The RWSA stores all the read set and write set of the subtransactions of all committed transactions at all sites. For the previous example, the RWSA is shown in figure 3.

	site 1	site 2	site 3
T_1	τ_{11}	*	τ_{13}
T_2	τ_{21}	τ_{22}	*
T_3	τ_{31}	τ_{32}	τ_{33}

Figure 2. Serial Order Array

	site 1	site 2	site 3
T_1	read-set$_{11}$ write-set$_{11}$	read-set$_{12}$ write-set$_{12}$	read-set$_{13}$ write-set$_{13}$
T_2	• •	• •	• •
T_3	• •	• •	• •

Figure 3. The Read-Write Set Array

When a global transaction has no subtransaction at a site, the corresponding read set and write set are said to be empty. When the GTM receives a global transaction, it derives the set of sites, which is called the site set, involved in the execution of the global transaction. The read sets and write sets for the subtransactions are provided by GDM. The algorithm consists of two modules (figures 4 and 5), the validation module and the commit module. In the following pseudo code for the validation module, T is the global transaction that is attempting to commit, t is a recently committed global transaction, s represents a

Validation module::

```
BEGIN
     serializable := TRUE;
    << For all t∈ {recently committed global transaction } DO
       BEGIN
         conflict_set := ∅;
         FOR all s∈ site_set DO
              IF((RWSA(t,s) conflicts with read_set(T^s)) OR
             (RWSA(t,s) conflicts with write_set(T^s)))
                  THEN conflict_set := conflict_set U {s};
         IF NOT ( all the serial orders of the conflict set
                   of t  precede the serial orders of the
                   subtransactions of T or vice versa)
             THEN
                 BEGIN
                   serializable := FALSE;
                   GOTO L
                 END
       L : END; >>
END.
```

Figure 4. The Validation Module

Commit module::

```
BEGIN
    FOR   all s∈site-set DO
    send the corresponding subtransaction to s;
    Invoke  a time-out mechanism;
    wait  until   (STUB  processes   of  all  site-set  send  the
responses)
    IF ( some node response with 'reject' flag)
       THEN
         broadcast 'restart' message to all site-set nodes
       ELSE    (* it is 'accept' *)
          BEGIN
             call validation module;
             IF serializable = TRUE
                THEN
                  BEGIN
                    broadcast 'commit' to all site-set nodes
               << update  the SOA and RWSA array for the
              committed global transaction >>
END
                ELSE
                   broadcast 'restart' to all site-set nodes;
          END;
END.
```

Time-out exception handler::

```
BEGIN
     broadcast 'abort' to all site-set nodes;
END;
```

Figure 5. The Commit Module

local site, and T^s represents the subtransaction of T on site s. The conflict-set is obtained from the site-set. The site set contains all the site ids where the subtransactions of the committed transaction (t) and the subtransactions of the attempting to commit transaction (T) conflict. In the above algorithm, '<<', and '>>' denote a critical section.

On receiving a global transaction, the GTM submits the subtransactions to the corresponding sites, and invokes a time-out mechanism. When the GTM can not receive all the response messages of the involved nodes in time, it will assume that some node has failed, so the corresponding global transaction is aborted. The global transaction is restarted when any subtransaction of it can not be serializably executed or when the global transaction has not been executed in a globally serializable way.

To make the above-mentioned approach applicable, the recently committed set of global transactions must be kept as small as possible. One way to achieve this is to record the time when the global transaction committed. And for every active global transaction, record the time when it is submitted. Periodically compare the commit time of global transactions with the submit time of active global transactions. The global transaction whose commit time is smaller than the submit time of all active global transactions can be purged from the recently committed set.

References

[BERN81] Bernstein, P.A. and Goodman, N.
"Concurrency control in distributed database systems"
ACM Computing Surveys, Vol. 13, No. 2, pp. 185-221 June 1981

[GLIG84] Gligor, V. and Luckenbaugh, G.
"Interconnecting Heterogeneous Database Management Systems", IEEE Computer, Vol. 17, No. 1, pp.33-43, Jan, 1984

[GLIG86] Gligor, V. and Popescu-Zeletin, R.
"Transaction management in distributed heterogeneous database management systems"
Information systems Vol. 11, No. 4 pp. 287-297, 1986, Pergamon Press

[HELA86] Helal, A. and Elmagarmid, A.
"Heterogeneous Database Systems"
Technical Report TR-86-004, Department of Electrical Engineering , Pennsylvania State University, 1986

[PU86] Pu, C.
"Superdatabases for composition of heterogeneous databases"
Technical report No. CUCS-243-86, Dept. Of Computer Science, Columbia University, 1987

Pragmatics of Access Control in Mermaid

Marjorie Templeton, Eric Lund, Pat Ward
UNISYS, System Development Group
2400 Colorado Ave, Santa Monica, CA, 90406
213-829-7511

1. INTRODUCTION

Access control is an important component of any DBMS (Database Management System). Users must be authorized to access the database and specific permissions may be defined for each user such as whether they may update data, modify the schema, access the entire database or only some subset, access only at specific times or from specific terminals. Definition of an access policy and maintenance of the policy becomes complex in distributed systems and even more complex in heterogeneous distributed systems because multiple system administrators and operating systems are involved.

In this paper we will discuss some general considerations for access control and then how it has been implemented in Mermaid, a distributed database front-end system.

2. OVERVIEW OF MERMAID

We will first give a brief introduction to Mermaid. For more details, see [TEMP87].

Mermaid provides the end user with the capability to access data in multiple databases through a common query language (SQL), using a single terminal, and through a common view of the databases. The user does not need to know where the data is located or when his query requires combining data from multiple databases. There is a central DD/D (Data Dictionary /Directory) that contains information about the data and its location.

The major processes in the system are:

- User Interface: The user interface appears to the user to be a DBMS because it provides a set of commands similar to those provided by most DBMSs. This includes support for query libraries, query editors, debugging, help, synonym replacement, spelling correction, report manipulation, and options for customizing the system. The user interface is used to prepare an SQL query (either a retrieve or an update) and submit it for processing.

- Distributor: The distributor process contains the optimizer and the controller. The query optimizer first accesses the

DD/D to determine which site or sites contain the data. If the query can be processed by a single site, it is sent to the controller. If multiple sites are required, the optimizer plans the lowest cost method to process the query. Query fragments and data transfer commands will be sent to the DBMS interface processes and finally a report will be assembled in one database and then sent to the report generator. The controller within the distributor initiates processes, transmits and receives messages from remote processes, and performs error checking and recovery.

- DD/D Interface: All information about schemata, databases, users, access rights, host computers, and the network is contained in a DD/D that is stored in a database and accessed through a special DBMS interface.

- DBMS Interface: There is one DBMS interface process for each "target" database to be accessed. It contains code that is specific to the DBMS, the operating system, and the network protocols on the host.

- Report Generator: The report generator formats the report.

3. ACCESS CONTROL IN MERMAID

Our goal in the Mermaid system is to develop a basic system that provides at least the level of access control offered by commercial centralized DBMSs while developing a secure version. The secure version will have less functionality and poorer performance, so we plan to maintain two versions. Providing basic access control in a distributed, heterogeneous system is much more complex than providing the same level of access control in a centralized system. There are more processors involved, more system admininstrators involved, and more levels of checking.

Administrative complexity arises due to the necessity of providing local control over local databases. In a tightly coupled distributed system, it is possible to have central system administration. However, in a federated system such as Mermaid, there are data and system administrators for Mermaid and for each underlying database. Access through the Mermaid system is a right that is granted to individual users of Mermaid. Access to the Mermaid system does not give the user automatic access to the databases accessed by Mermaid, and access to the underlying databases does not give a user access to Mermaid.

3.1 USER VIEWS

A Mermaid database is a "virtual" database that is a view into one or more underlying target databases. The full federated database includes all relations and fields in the federated view

which may actually be less than all of the data that is available in the underlying database. When developing the Mermaid federated database, the schemas of the underlying databases are supplied to the builder and are stored as "local schemas" in the DD/D. These local schemas may actually be views. It is easiest to develop the federated database if all Mermaid users use the same view into the underlying database and then further protection can be built using a Mermaid view. That way, Mermaid does not have to deal with different local schemas in the same underlying database. If this can't be done, then each view of the underlying database needs to be defined as a different target database even though the views are actually in the same underlying database.

User views may be defined for specific users or groups of users above the federated schema. Instead of opening the federated database, the user opens a view. For example, the federated database name may be "navyall" while views are named "pacific_ships", "submarines", and "battle_bgroups". The navyall database contains information on all types of ships in all oceans. It includes information on the ship's location, plans, weapons carried, membership in battle groups, maintenance history, and ports visited. The "pacific_ships" would include all information but only about ships in one location. The "submarines" would include all information about a specific type of ship. The "battle_groups" would include a subset of the relations.

Several federated databases may be defined over different groups of databases and subsets of the databases. Any single database may belong to many federated databases. This capability is another way to control access. For example, some group of users may need access to several databases containing information about commercial shipping while another group of users is interested in military activities. Some databases may contain information about both and therefore different subsets of the databases may participate in both federated databases. It is a matter of judgement whether the "commercial" and "military" views should be implemented as two separate federated databases with their own views or as two views of the same federated database. If the two sets of users are quite distinct, the relations accessed are largely disjoint, and the target databases that participate are partially disjoint, then it may simplify administration to consider them separate federated databases. If the access controls on the two databases are quite different, then it will make the system more secure to treat them as different federated databases.

3.2 INSTALLING A NEW USER

The first step in installing a new user is to determine which federated databases or views the user will be allowed to access. He then needs to obtain a login id on each computer that contains

means that end user organizations will not see the code. If source code is ever provided, the encryption routines must be stripped out or provided with special protection to protect the seed values to the random number generator and the manifest constants for the enigma machine.

In some environments, even the storage of encrypted passwords is not acceptable. We have a flag in the DD/D that indicates that no password is stored and that we must request one from the user when logging into a target database computer. This makes the system less transparent, but it is one of the necessary tradeoffs between ease of use and security.

3.5 SECURITY OFFICER MONITORING

Mermaid keeps two types of trails: the audit trail and the journal trail.

The audit trail is intended for the security administrator. It covers all logins to Mermaid as well as rejected attempts. A record is also written with significant events such as the logins to remote computers and the queries asked. The audit trail is protected by Unix as writable by everyone and readable only by the security administrator. It is stored in a directory in the Mermaid system space. This provides some protection, but it could be erased by a malicious user.

The journal file is written for each run of Mermaid. It could be used for detailed security checking, but its primary purpose is for system debugging and performance monitoring.

4. REFERENCES

[KAHN67] David Kahn, "The Codebreakers", The Macmillan Company, 1967.

[KNUT81] Donald Knuth, "The Art of Computer Programming, Seminumerical Algorithms", Volume 2, 1981.

[TEMP87] M.Templeton, D.Brill, A.Chen, S.Dao, E.Lund, R.MacGregor, P.Ward, "Mermaid - A Front-end to Distributed Heterogeneous Databases", Proceedings of the IEEE, May 1987.

A FEDERATED SYSTEM FOR SOFTWARE MANAGEMENT

Dennis Heimbigner

Computer Science Dept.
University of Colorado
Boulder, CO 80309-0430

1. Introduction

The federated approach to data sharing [HAMM80, HEIM85] was originally set in the context of traditional databases. A federated database was conceived as one which had no global schema. The centralization inherent in traditional distributed databases was replaced by autonomy plus cooperation. Autonomy referred to the ability of component systems to decide which information to share (export), which external information to use (import) and to dynamically change those decisions. The architecture described in [HEIM85] envisioned multiple autonomous workstations, each with its own database and local schema. These databases also had export schemas and import schemas to control the interdatabase sharing.

The federation architecture appears to be useful in contexts other than databases. In particular, it appears to be a useful organizing principle for distributed software management systems[1]. Such systems are rapidly moving off of single mainframes and onto networks of personal workstations. In a mainframe, it is easy to provide implicit sharing of various software objects (programs, libraries, tools, and so on) though a common file system. In a network of workstations without any distributed file manager, implicit sharing is difficult and a larger burden is placed on programmers to establish explicit patterns of sharing by using, for example, file copying and mail messages.

Keystone II[2] is a prototype system developed at the University of Colorado to explore some of the issues involved in federated software management. It was constructed by grafting new features onto an existing centralized software manager called Odin [CLEM86]. Keystone II is not intended to be provide solutions to all the issues of federation, but rather it is designed to emphasize three issues:

Transparency:
: Access to shared objects is as transparent to user programs as possible. In particular, the location of objects is transparent to Odin.

Consistency:
: Keystone II maintains a loose form of consistency between the exporter's copy of an object and the importer's copy. Importers keep local copies that are updated automatically when the exporter's copy is changed.

Closure:
: Importing a shared object may require importing additional supporting objects (e.g., libraries or included files). The set of supporting objects is called a *closure*. A major research issue for Keystone II is the exploration of the problems posed by closure.

Two other issues that are important for federations are autonomy and avoidance of global schemas. These elements are present in Keystone II, but they are not primary issues in the research.

Keystone II is implemented in C under Sun Unix 3.3. It currently will import objects and keep them up to date. It can support closures for certain objects, it maintains an import database, and allows "de-import". Keystone II must be considered a prototype, but even at that level, it shows the relative merits and demerits of a federated approach to software management.

2. Odin

To understand Keystone II, it is necessary to understand the software management portion of Keystone II, which is represented by the Odin system [CLEM86]. Odin is built on top of Unix and manages a collection of objects, which are Unix files. In Odin, objects are either *atomic* or *derived*. An atomic object is one which is provided to Odin by a programmer. In effect, it is any object that is not derived. Typically atomic objects are source files constructed by a programmer using an editor.

A derived object is one that is automatically produced from other objects by the application of a *tool*. A tool in Odin is a program that takes objects as input and produces an object[3] as output. The output object is said to be derived from the input objects. The input objects may be atomic or derived.

[1] The term "distributed" is intended to modify the term "system" as opposed to the term "software".

[2] Keystone (I) [CLEM85] was an earlier, simpler, attempt to do this, but it lacked transparency and inter-machine consistency.

[3] In practice, a tool can produce multiple outputs as pieces of a single *composite* object.

In Odin, there is a system of types and supertypes forming a type hierarchy (a DAG). All objects (both atomic and derived) known to Odin are typed. The type of derived objects is determined by the tool that produces them. Odin requires that each tool produce a unique type, so tools and types are interchangeable concepts in Odin. Since Unix files technically are untyped, Odin uses file name extensions as a means of determining the type of atomic objects.

The main activity of Odin is to maintain and extend two relations: *derives* and *source*[4]. The *derives* relation shows how objects of one type can be derived from objects of other types. A tuple of the form

$derives([T_1,T_2,...T_n],T_0,Tool_0)$

exists for every derived type T_0. It indicates that objects of type T_0 can be derived by applying $Tool_0$ to a set of objects of types T_1 through T_n.

The *source* relation corresponds to the derives relation but describes which existing objects have actually been derived from other objects according to the rules defined in the *derives* relation. There is an M to 1 mapping from tuples in the *source* relation to tuples in the *derives* relation. Suppose that object O_i is of type T_i, Then the *source* relation might contain the tuple

$source([O_1,...O_n],O_0)$

if object O_0 had in fact been produced by applying the appropriate tool to O_1 through O_n. This tuple in *source* corresponds to the previous tuple shown for *derives*. Notice that the tool need not be respecified in the source relation since it is deducible from the type of object O_0.

As described above, Odin has essentially the same functionality as the Unix Make program [FELD79], where the dependency lines in a Makefile correspond to tuples in the *source* relation and the rules in a Makefile (such as ".c.o" rules) correspond to tuples in the *derives* relation. In Make, the programmer must explicitly specify the whole *source* relation in the form of a complete Makefile. Odin differs in being able to automatically extend the *source* relation by a process called "type inference". Thus if I ask for the object of type "exe" (standing for a Unix executable program file) that is derivable from, say, "test.c", Odin can search its *derives* relation to find a sequence of types leading from "c" to "exe" and then apply the corresponding tools to extend the *source* relation to construct that derived object of type "exe"[5]. Notice that this will infer an intermediate object of type "o" (Unix relocatable file) and place it in the *source* relation automatically. In Odin, this kind of command is written as "test.c : exe". This command may also be said to be the name of an object since it specifies how it is constructed from existing objects ("test.c" in this case).

3. Keystone II Architecture

Figure 1 shows the functional architecture of the Keystone II system. It is assumed that programmers on each workstation interact with Odin. They use Odin to manipulate objects located at their workstation (local objects) as well as remote objects that have been imported. These imported objects appear to be just additional local atomic objects as far as Odin is concerned. It is important to notice that the imported object may actually be derived on the exporting system. It will appear as an atomic object locally, however.

As the figure shows, each exporter has an export server process running at all times. Depending on what it is doing, the export server may need to invoke a server Odin on its node to complete some of its activities.

Operations on the federation are invoked as tools under Odin. These tools run as Unix processes under Odin. The figure shows Odin on node 1 executing the import tool. The import tool typically talks over a network to the export server on some other node. In the process of importing an object, the export server is in turn calling a server Odin to perform some function (see next section).

Keystone II uses a number of private databases. "ODIN/IMP" is a directory containing the local copies of imported objects. The import database records information about the imported objects. The sentinel database is used by the server Odin to maintain consistency. These databases are described in more detail in the next section.

4. Detailed Operation of Keystone II

The programmer using Keystone can do all of the normal Odin operations, plus the operation of importing some remote object. The import activity is actually embedded into Odin as just another set of tools.

Rather than describe the general format of the import command, we will use the following example command as the basis for presentation:

db : import +host=ehost +object=program.c:exe +dest=local.exe

Odin requires an initial object from which to derive other objects, and so we introduce an empty object, "db", to fill that role. In Odin, parameters are specified using a "+" notation. Thus, this command specified three parameters to the

[4] These two relations are actually stored as directed graphs. The type derivation graph (TDG) stores the *derives* relations and the object derivation graph (ODG) stores the *source* relation.

[5] Make actually can do a limited form of type inference, namely one step inference. Odin, however is substantially more powerful since it can infer arbitrarily long derivation sequences.

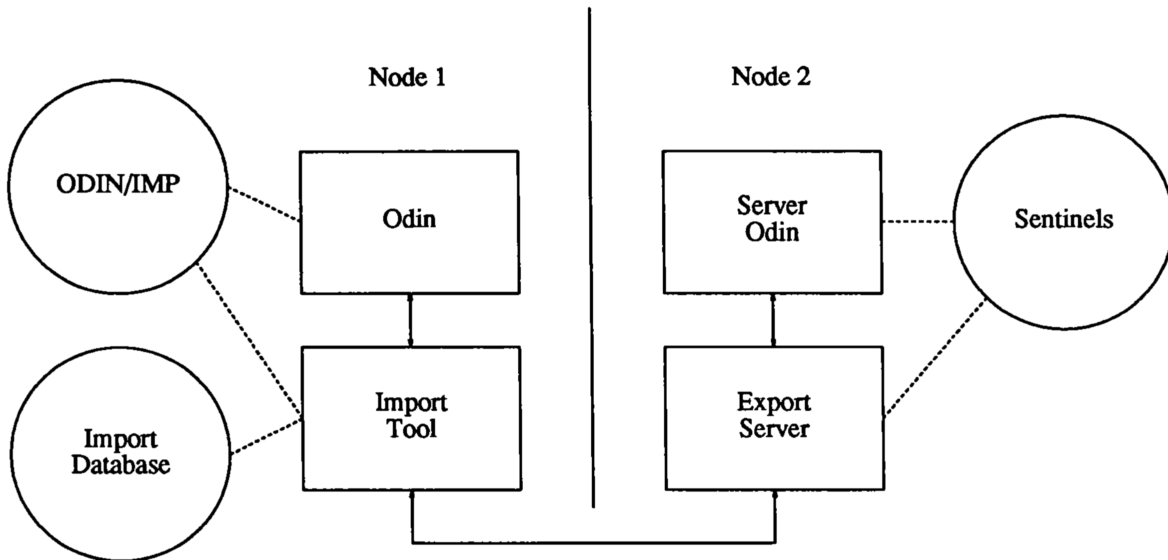


Figure 1. Keystone Architecture.

command: (1) a host name ("ehost"), (2) an object name ("program.c:exe"), and (3) a local destination name ("local.exe"). The host specifies the remote machine from which the object is to be imported. The object name is in the context of that remote host's file system and Odin *source* relation. The destination name specifies the location of that imported object in the local file system and its local name in that file system. The type of the final object is "import", but it is an empty object. The whole purpose of this command is to force the execution of the import tool solely for its side effects (i.e., setting up the import) and not for any output it might produce (except error messages).

When Odin receives this command, it invokes the import tool with the specified parameters. Notice that the file name extension of the destination name should match the type of the remote object ("exe" in this example). This allows Odin to treat the object correctly if it is used in further local derivations.

When the import tool is invoked, it picks a unique name in a special directory on the local host. The special directory is called "ODIN/IMP", and so the import tool might generate the file name "ODIN/IMP/ta03835". This file is where the local copy of the remote file will physically reside in this file. Normally, the programmer does not need to be aware of this file. The local file name (the "+dest" parameter) is created as a symbolic link[6] to the local copy. Thus from the user's point of view, the local copy appears to reside at a specified place in his file system even though it physically resides in the "ODIN/IMP" directory.

Once the local files are taken care of, the import tool establishes a network connection to the export server on the remote host machine ("ehost", in this case). The import tool sends three pieces of information to the export server: (1) the importer's identity ("ihost" in this case). (2) the remote object name, and (3) the name of the local copy file (e. g., "ODIN/IMP/ta03835"). Eventually, the export server returns a message indicating success or failure, and the import tool terminates its operation, which in turn completes the original import command. Assuming everything succeeds, the programmer can then begin to use the imported file in subsequent Odin commands.

The export server on each machine has a number of duties. It must arrange for the physical instantiation of the object to be exported, it must copy the object to the importer, and it must set up a mechanism for ensuring that the imported copy is kept consistent with the exporter's copy.

The first and second of these steps (instantiation and initial copy) are handled by using a server Odin plus special tools on the exporter to do all the work. As a result of the above example import command, the export server on "ehost" would start up a server Odin and give it the following command:

program.c : exe : export +host=ihost +dest=ODIN/IMP/ta03835

[6] A symbolic link is a file which contains the name of another file. References to the symbolic link file are transformed by Unix into a reference to the file named in the symbolic link.

This command requests Odin to create the derived object "program.c : exe," which is the object that is being exported. This object is then derived to an object of type "export", which is again a ruse for invoking the export tool for its side effects. The export tool copies its input object (program.c:exe) over the network to the importing host ("ihost") and stores the copy into the local file "ODIN/IMP/ta03835" on the importer. The copy is actually performed using the Unix "rcp" (remote copy) command. At the completion of this Odin command, the exported object will have been instantiated and a copy placed with the importer.

The third step of the export server's operation is to ensure a limited form of consistency between copies on the exporter and the importer. To ensure consistency, changes to the exported object (which must occur at the exporter) must automatically cause all imported copies to be changed to match the export copy. Consistency enforcement is not atomic. Rather, it is a looser form in which the exported objects are re-copied to the importers one at a time. Thus there may be a short delay before all imported copies are consistent with the export copy.

Keystone II implements this function by using a feature of Odin called sentinels. An Odin sentinel is a command that has the property that its corresponding derived object is always kept current. It is specified by giving an Odin command to access a specific Odin object.

Whenever an object is imported, the export server establishes (on the exporter) a sentinel such as the following:

program.c : exe : rcp +host=ihost +dest=ODIN/IMP/ta03835

Whenever the object "program.e:exe" changes, this whole sentinel command is immediately re-evaluated by Odin. The effect is to copy the new value of the object to the importer of that object using "rcp". This ensures consistency between the local exporter's copy and the importer's copy of the object.

Eventually, an importer may decide that it no longer needs access to a shared object. So Keystone II provides a facility to support the "de-import" of an object. Simply deleting the symbolic link is inadequate since the local copy will still exist, the import database will not be consistent, and the exporter will not know of the deletion. Keystone II deals with this by providing a command:

db: import +remove=local.exe

This command removes the symbolic link, reclaims the space in "ODIN/IMP", notes the change in the import database, and sends a message to the export server to update its information.

5. Import Closure

For transparent import of single files, the above mechanism is quite adequate. Unfortunately, it is rare for a file to be so isolated that it alone needs to be imported. For example, importing the C source file "test.c" for the purposes of compilation will probably require importing any **#include** files referenced by that C source file. In other words, "test.c" depends on its included files. Equivalently, these included files support "test.c".

It is time consuming and error prone to have the programmer import a file and all the files upon which it depends, so Keystone II attempts to determine the set of supporting files and automatically import them along with the original file. This collection of supporting files is termed a *closure*. Unfortunately, the concept of a closure is tricky to define. The notion of "depends" hinges on the operations to be performed on the imported object. For example, if "test.c" is imported for the purpose of printing it, then its included files may be irrelevant. If it is to be compiled, then they are quite relevant. If it is to be compiled and loaded, then the closure may also need to specify appropriate libraries.

Keystone II defines closures by requiring the importer to state what types will be derived (by the importer) from the local copy of the imported object. This is equivalent to specifying the tools that will be applied to the object by the importer (because the type implies the tool). To handle closure, the import tool is redefined to take an additional parameter that specifies the list of types that the importer expects to derive from the imported object. For example, when importing "test.c", the programmer might specify "+deriv=[o,exe]". It is important to realize that these derivations will be performed at the importer, not at the exporter. But the exporter needs to know this list so it can figure out what other exported objects would be needed by the importer to actually do those derivations. In other words, the exporter constructs a closure based on information provided by the importer about what derivations the importer will use.

In order to avoid building the closure construction into Odin, there is assumed to exist a collection of closure construction tools (one per Odin type) that actually calculate (at the exporter) the list of files that must be included in the closure. These tools must know about the possible derivations from their type and be prepared to calculate the closure appropriate to any given set of derivations. These tools must also be recursive since forming the closure is a transitive operation. That is, included files, for example, may in turn reference other included files.

It is interesting to note that closure was not explicitly treated in the original formulation of federated databases [HEIM85]. It turns out that the issue was implicitly recognized but it was avoided by requiring a user to import all supporting types before importing any type. Thus, the user was forced to import the closure manually.

6. Closure Issues

In practice, calculating and importing a closure is a difficult operation. In particular, it may not be possible (at least in Unix) to even calculate the closure. Suppose for example that a host wants to import a Unix relocatable object file (i.e., ".o") and apply the "exe" derivation to it. In Unix, the construction of an executable file requires the programmer to specify a set of libraries needed to complete external references in the relocatable file. These libraries are specified on the loader command line and are not defined in the relocatable file itself. Thus, it is impossible to calculate the closure of a relocatable file given only the contents of that file. Somehow, an external specification of the required libraries must be provided. Since there is no easy way in Keystone II to specify such external specifications, importing of non-closeable files is disallowed.

Even when a closure can be calculated, it is still difficult to properly import all of the files in the closure. The importer will have specified the local name for the basic imported file, but there will be no names specified for the other files in the closure and the references to the supporting files (i.e., the path names) may not be correct within the file system of the importer.

Keystone II attempts to solve this problem by changing the references to the supporting files so that they are correct within the importer. This is accomplished by the following sequence of actions:

(1) The export server calculates the closure and returns it to the import tool.

(2) The import tool assigns unique local names in "ODIN/IMP" to each of the files in the closure and sends this list of local names to the export server. It also records the closure and local names in an import database (see figure 1). If a file has already been imported for some other closure, then it is not re-imported.

(3) the export server applies a per-type renaming tool to each file in the closure to produce a new version that properly references the new local name for all supporting files. Notice that this means that the exporter's copy of a file is no longer identical to the importer's local copy.

(4) The export server establishes appropriate sentinels to make sure that the renaming is performed every time a new copy of a file is sent to the importer.

The renaming tools are almost identical with the closure constructor tools except that instead of extracting names of supporting files, they modify references in those files. In fact, our current extraction and rename tools use the same text with small compilation flags to determine which tool is to be created.

7. Related Work

Within research on software environments, the range of problems to be solved is sufficiently broad that few environments manage to address the issue of distribution. None of the environments which do consider the issue take an explicitly federated approach. Rather, they are either based on a (logically) centralized data repository (i.e., a classical distributed database), or they rely on some existing distributed file system.

DSEE [LEBL84, LEBL85a, LEBL85a] is a system developed by Apollo Computers that relies on a distributed database to achieve multi-machine operation. Cedar [DONA85, SCHM82, TEIT85] at Xerox Parc uses an existing distributed file system in its operation. The differences between classical distributed databases and federated databases have been discussed in [HEIM85] and need not be repeated here. However, a comparison of federated systems and typical distributed file systems is relevant to Keystone II. In particular, Sun Microsystems Network File System (NFS) [SUN86] is representative of typical distributed file systems and is a reasonable alternative to a federated approach in an network of workstations. A comparison of the features of a federated approach to the features of NFS may be of interest.

NFS (network file system) was developed at Sun Microsystems, but is rapidly becoming a standard for network file systems. NFS allows a workstation node in a network to act as a server by exporting a subtree of its file system. There is, in effect, an export database on each server describing the subtrees that are exported. A client node can graft an exported subtree onto its local file system using the Unix "mount" facility. The server is actually unaware of which clients have imported one of its subtree because NFS uses a so-called stateless protocol. Access to such a subtree is (almost[7]) transparent to programs executing on the client. Client access (reads and writes) to files in the imported subtree are all performed on the server so that clients can update remote files. The export database, which is a text file on the server, is persistent over server crashes, but the mounts performed by a client disappear whenever a client crashes. This means that client must re-import all of its subtrees at every reboot. Because the Unix mount facility has security implications, only highly privileged users (i.e. the super-user) can import and mount remote file systems. The practical effect of this restriction is that the pattern of file sharing between servers and clients is relatively static.

Keystone II also provides an (almost[8]) transparent interface to programs. Unlike NFS, the imports established by Keystone II are persistent over crashes since they are recorded in the file system rather than, as in the Unix mount facility, in the computer's memory. Also, unlike NFS, anyone (subject to normal security constraints) can import a file from another node in the network. Special privileges are not required, which means that the patterns of sharing may be more

[7] The stateless protocol breaks the transparency.

[8] It is transparent to the extent that Unix symbolic links are transparent to programs.

dynamic than with NFS. Keystone II uses local copies for imported files so client updates are not as transparent as with NFS. Read access, however, can be faster because the copy is local. Finally, because exported files are supported by Odin, they may be computed rather than static. That is, an NFS remote file is just an ordinary data file, whereas a Keystone II remote file can be the result of a complex series of derivations. Further, Keystone II will keep the exporter's copy of the derived file current and automatically transfer that copy to importers to maintain consistency. Finally, it is interesting to see how both systems handle the closure problem. Keystone II has specific mechanisms for closures. NFS sidesteps the issue by assuming that the imported subtree includes the closure or that the required files are available in fixed places in all file systems (such as <stdio.h>).

8. Unresolved Issues

Keystone II currently does not support an export database such as was proposed for federated databases. Introducing this database into Keystone II obviously would require extending Keystone II with some new tools to maintain this database. Providing such a database would make it possible for Keystone II to provide better access control to exported objects and to allow the exporter to revoke access to already exported objects.

As originally conceived, one of the motivations for the federated approach was to allow dynamic changes in the structure of the federation. Such changes might include failures of various kinds as well as changes in the export interfaces. Robustness in the face of such changes is a difficult to achieve and Keystone II has as yet no facilities for dealing with it. It is possible, however, to outline some of the problems and potential solutions.

Transferring ownership of an exported object from one node to another is the basic operation underlying changes in the federation. The simplest approach is to transfer the object and the relevant sentinels to the new exporter and then notify each importer of the change. One example of the use of such a transfer mechanism is in configuration control. It is likely that shared software objects are periodically placed under configuration control. To do this, such an object must be moved from a developer node to a configuration control node.

Another way that federation changes may occur is through network failures. A network of workstations can fail in a variety of ways: nodes may fail or the network may become partitioned. Further, such failures may last for varying periods of time. The principal effect of most failures in a federation is to prevent access to some subset of the available exported data for some period of time.

In Keystone II, inability to access already imported objects for short periods is not a serious problem since the importers have local copies. In fact, under the current implementation, exporters can fail and recover any number of times without the importers being aware of it. Attempts to import new objects from a failed exporter will, of course, be delayed until the exporter recovers.

Long term failures are more serious because the current architecture prevents updates to exported objects except at the exporter. Thus objects exported by failed nodes are unmodifiable. One approach to dealing with this is to extend the basic transfer mechanism. This involves choosing (by some form of election) some other node to serve as exporter for the objects previously exported by the failed node. Once such a node is chosen, the appropriate objects (and their closures) need to be transferred to that elected node. The basic transfer mechanism can be used with the modification that the source of the object to be transferred is one of the local copies on an importer rather than the original exported object.

Update of shared objects is always a problem and Keystone II does not yet address this issue. At the moment, only the exporter can effectively modify a shared object and have it copied to the importers. Eventually, Keystone II must allow importers to make such modifications and have them properly distributed.

Heterogeneity is another issue that is not addressed in Keystone II. Keystone is capable of operating on various machines (Suns, Vaxes, Pyramids, etc.) as long as they all use Berkeley Unix 4.3, they all use Odin. and only text files are exported. More diverse forms of heterogeneity are beyond Keystone's current capability.

9. Conclusions

Keystone II demonstrates that the federated approach is a viable approach to distributed software management. It provides transparency and inter-machine consistency. Closure presents special problems in the software context and Keystone II provides a partial solution. It is clear that many other issues still need to be addressed: update, robustness and heterogeneity to name three. Further research is needed to clarify these issues and to provide more general solutions to all of the problems faced by Keystone II.

Acknowledgements I would like to acknowledge the help of Henry Vellandi in constructing a working version of Keystone II. I would like to also acknowledge the support of Lee Osterweil for suggesting that a federated Odin might be interesting. The help of Geoff Clemm, the author of Odin, was also indispensable.

References

[CLEM85] G. M. Clemm, D. M. Heimbigner, L. J. Osterweil, and L. G. Williams, "KEYSTONE: A Federated Software Environment," *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, Harwichport, Massachusetts, June 1985, pp. 80-88.

[CLEM86] G. M. Clemm, "The Odin System: An Object Manager for Extensible Software Environments," University of Colorado Ph. D. thesis.
Also available as Technical Report CU-CS-314-86, University of Colorado, Department of Computer Science, February 1986.

[DONA85] J. Donahue, "Cedar: An environment for 'experimental' programming," in *Integrated project support environments*, J. McDermid (Ed), 1985, pages 1-9.

[FELD79] S. I. Feldman "Make - A Program for Maintaining Computer Programs," *Software Practice and Experience* volume 9, pp. 255-265, 1979.

[HAMM80] M. Hammer and D. McLeod, "On Database Management System Architecture" in *Infotech State of the Art Report: Data Design*, published by Pergamon Infotech Limited, Volume 8, pages 177-202, 1980.

[HEIM85] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Mangement," *ACM Transactions on Office Information Systems*, volume 3, no. 3, pp. 253-278, 1985.

[LEBL84] D. B. Leblang and R. P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, published as *Software Engineering Notes* volume 9, no. 3, pp. 104-112, 1984.

[LEBL85a] D. B. Leblang and G. D. McLean, Jr., "DSEE: Overview and configuration management," in *Integrated project support environments*, J. McDermid (Ed), 1985, pages 10-31.

[LEBL85b] D. B. Leblang and G. D. McLean, Jr., "Configuration Management for Large-Scale Software Development Efforts," *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, Harwichport, Massachusetts, June 1985, pp. 122-127.

[MCDE85] J. McDermid (Ed.), *Integrated project support environments*, published by Peter Peregrinus Ltd, 1985.

[SCHM82] E. E. Schmidt, *Controlling Large Software Development in a Distributed Environment*, Ph.D. Thesis, EECS Department, University of California, Berkeley, 1982 and Technical Report CSL-82-7, Xerox PARC, 1982.

[SUN86] *Networking on the Sun Workstation*, Part no. 800-1324-03, Revision B of 17 February 1986, Sun Microsystems Inc.

[TEIT85] W. Teitleman, "A Tour Through Cedar," *IEEE Transactions on Software Engineering* volume SE-11, no. 3, pp. 285-302, 1985.

Information Interchange between Self-Describing Databases

Leo Mark & Nick Roussopoulos
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Within the framework of a Self-Describing Database System we describe a set of Data Management Tools and a Data Dictionary supporting Information Interchange.
The concepts are based on our experience from a project on standardized information interchange in NASA[1].

1. Introduction

Although electronic data transfer is technologically feasible today, information interchange is not. Information interchange cannot be achieved by transferring bit-strings using standard commercial protocols that support only the data interchange. High level information interchange protocols must be developed and used to allow the utilization of the technological miracles called communication networks. Information interchange can only be achieved if the sender and receiver have the same high level semantically rich description of the data being interchanged. We therefore believe, that protocols for information interchange must support the interchange of metadata as well as the data itself.

The approach to information interchange described in this paper is much less ambitious than those followed in distributed, heterogeneous, and multi- databases where a complete global schema is available on-line to all users. In our approach, the global schema consists of a table of data format identifiers. All electronically transmitted data, including metadata, is prefixed by a data format identifier and can be interpreted automatically by first requesting and interpreting the corresponding data format. A standardized protocol for this kind of information interchange has been defined by NASA's Consultative Committee for Space Data Systems over the past 2 years, and an organizational entity, the Control Authority, is being established to control and maintain the registration and use of data formats. The focus of this paper is on the software tools and the data dictionary needed to support the standardized protocol. Prototypes of the software tools and the data dictionary are currently being built as a proof of concept.

The concepts of data models that can describe themselves and document their own evolution are central to information interchange. The definition of these concepts are summarized below:
Self-describing models are those which allow users with some basic knowledge about the model to browse and find out all the information needed to use the database. The basic required knowledge is minimal, dependent on the model itself, and independent of the database content. For example, in a self- describing relational model, the basic knowledge required by the user includes the table structure, the way the column (attribute) names are used, the relational algebra or another query language, etc.
Self-documenting models are those which can document the evolution of the database during operation. In a self-documenting data model a user needs no additional knowledge to understand data derivations. The semantics of data derivations can be obtained from the semantics of the operand data objects and the used operators. This means, for example, that in a self-documenting relational model the user has no problem understanding the meaning of any derived relation once he understands the meaning of the base relations and the used operators.

Integrating data, schema and meta-schema (i.e. the schema description) into a uniform structure is fundamental in a self-describing model because this allows the users to browse through the schema first to find out about the database and then proceed to access the data. Understanding both the operations permitted by the data language and the interpretation of the obtained results is instrumental in a self-documenting

1: This work was supported by NASA under contract no. NAS5-29265.
Positions herein are those of the authors and do not necessarily reflect the official NASA position.

model, because without this understanding the system cannot document the database evolution.

The idea of integrating data, schema, and meta-schema has been pursued in several papers. The Extended Relational Model, RM/T [Codd 79], expands the relational algebra with operations on the catalog to support database reorganization. More recently, the ISO-Report on conceptual schema concepts [Griethuysen 82], points out the need for explicit representation and capabilities to change metadata. An excellent paper, entitled "Scientific Information = Data + Metadata," [McCarthy 84], supports our position that we can only interchange information by interchanging both data and metadata - without the metadata we cannot interpret the data. Finally, Information Resource Dictionary Systems, IRDS, have been the subject of a considerable amount of research [Dolk 87], [ANSI X3H4 part 1-4]. Our own work in this area include [Burns 86], [Mark 85a, 85b, 86], [Roussopoulos 83].

The rest of the paper is organized as follows. In section 2 we briefly present the Architecture of a Self-Describing Database System. In section 3 we present the concept of Standard Format Data Units - SFDUs, developed at NASA as part of a project on standardized information interchange. In sections 4 and 5 we present, within the framework of a Self-Describing Database System, the software tools and the data dictionary contents required to support information interchange.

2. Architecture of a Self-Describing Database System

The architecture of a Self-Describing Database System is illustrated in figure 1, [Mark 85]. This architecture has recently been adopted by the ANSI/SPARC as the basis for a new Reference Model for database management systems, and it is the basis for current work in the ISO. For a detailed description of the Reference Model, see [Burns 86].

A Self-Describing Database System is unique in that it provides an **active and integrated** data dictionary as part of the database management system; and, as we shall see, this data dictionary plays an important role in information interchange.

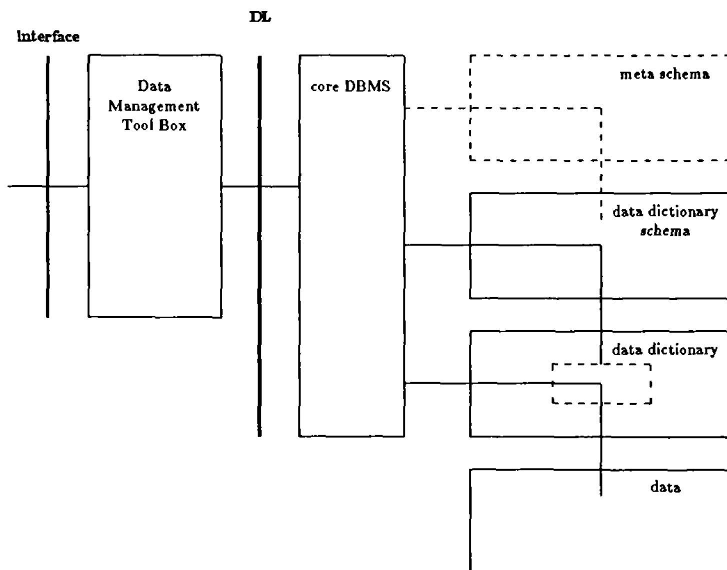


Figure 1. Architecture of a Self-Describing Database System

The **core DBMS** supports the well-known **point-of-view dimension** of data description, consisting of internal, conceptual, and external schemata. In addition, it supports and enforces the **intension-extension dimension** of data description. The intension-extension dimension has four levels of data description. **Application data** are stored as **data**. The **application schemata**, describing and controlling the use of the application data, are stored in the **data dictionary**. The **rules for defining, managing, and controlling the use of the application schemata** are stored in the **data dictionary schema**. A **fundamental** set of rules for defining schemata, i.e. a description of the data models supported by the Self-Describing Database System, is defined in the **meta-schema**. The set of rules in the meta-schema will allow the management strategies

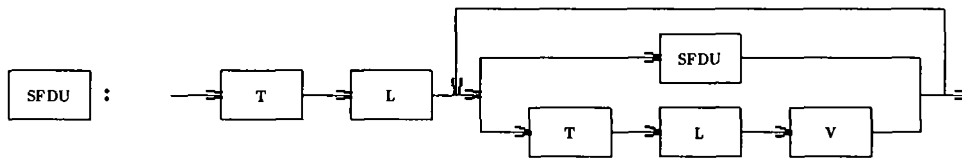
represented in the data dictionary schema to evolve in accordance with changing data management policies. Each level of data description in the intension-extension dimension is the extension of the level above it, and the intension for the level below it. The meta-schema is self-describing, i.e. it is one of the schemata it describes.

The **core DBMS** can be thought of as a DBMS stripped to the bones. It supports the Data Language, DL, which is the only language used to retrieve and change data and data descriptions at any level in the intension-extension dimension. The DL provides a set of primitive operations on any data element or data description element at any level in the intension-extension dimension of data description. Any compound operations needed must be implemented as a tool in the **Data Management Tool Box** using the primitive operations of the DL. Data Management Tools are plug-compatible with the core DBMS through the DL.

As we shall see, all the tools needed to support information interchange between Self-Describing Database Systems will be presented as Data Management Tools.

3. Standard Format Data Units - SFDUs

An SFDU is self-describing, i.e. it contains both data and information about the data format needed to automatically interpret the data. An SFDU has a recursive type-length-value, TLV, encoding with the following syntax:



The Type Field, **T**, has a fixed length and is represented in restricted ASCII. It identifies the data format definition needed to interpret the data contained in the Value Field, **V**.

The Length Field, **L**, is a fixed length restricted ASCII integer or a binary, depending on the Type Field, **V**. It represents the length of the Value Field, **V**, in octets.

The Value Field, **V**, can be in any desired code or representation that can be expressed using a data format definition language, **DDL**, supported by the Self-Describing Database System.

If a network is not exclusively used for transporting data structured as SFDUs, then a special Type Field is needed to indicate that something is an SFDU. For a detailed discussion, see [CCSDS 86].

4. Data Management Tools for Information Interchange

The following Data Management Tools are needed for information interchange:

- data format browser
- data format editor
- SFDU editor
- SFDU sender
- SFDU receiver
- SFDU interpreter
- data format interpreter

Each tool has a simple function, but several tools may of course be combined to perform more complex functions once each of them is fully understood.

Data Format Browser:
A user must be able to **browse** a set of **data formats** for information of interest. Traditional data retrieval languages allow users, that know what they are looking for and have sufficient knowledge about the structure of the database to ask for it, to retrieve data in a structured manner. In contrast, a database browser allow users, that do not know exactly what they are looking for and how to ask for it, to build up a sufficient level of knowledge about the data and the data formats of the database to retrieve data from it.

Since the primary purpose of data formats is to locate, define, and control data, a database browser must give its users easy access to data formats as well as data, and it must present both in the same way. In addition to the metadata strictly needed by the database system there should be metadata aimed at supporting the database browser. These include lists of aliases and synonyms for names, textual descriptions, subject indices, descriptions of derivations, etc. One of the unique characteristics of a Self-Describing Database System is that it represents data and data formats in exactly the same way; therefore the browser can be used to browse both data and data formats and to jump between them using one simple set of concepts at both levels of data description.

Data Format Editor:
Having located some data of interest, a user must be able to **request** data in a data format which fits the application best. To meet the user's request a set of operations allowing the user to specify and store a desired data format using existing data formats must be available to the user. In addition, a set of operations for defining data formats from scratch and storing them must be available.
To eliminate errors and provide a high level of support in the definition or derivation of data formats, the above operations are best supported by a **syntax directed editor for each DDL** used.
The specific set of operations on data formats will of course depend on the DDLs used. Some examples of operations on data formats that are defined in terms of the DDL for the relational data model are: **union** of two formats or parts of two formats; **intersection** of two formats or parts of two formats; **join** of some of the relations in a format or of relations in different formats; **projection** of some relations in a format; and, **selection** of a subset of the data of some relation in a format.
Two important function of the data format editors are the **registration** of new data formats and the **control of versions** of data formats. If new data formats are not registered, then data represented according to the data formats cannot be interpreted.

SFDU Editor:
To eliminate errors and provide a high level of support in the definition of SFDUs, a **syntax directed editor for SFDU assembly** should be provided. This editor will aid the user in correctly defining and storing SFDUs by controlling that SFDU assembly follows the syntax defined above.
The syntax directed SFDU editor will also support operations on SFDUs to construct new SFDUs.

SFDU Sender:
The **SFDU sender** encodes SFDUs produced by an SFDU editor in a representation that conforms to the ISO-OSI network Presentation Layer Protocol, used and sends the SFDU. It is relatively unimportant which language we use for the concrete encoding of an SFDU, ISO-8211 [ISO 86b], GDIL [Billingsley 86], etc., as long as the language supports a precise representation of all elements of the SFDU. For a description of the ISO-OSI Reference Model see [ISO 81].
The SFDU sender may be combined with and accept input directly from the SFDU editors, or it may accept stored SFDUs identified by the user when the SFDU sender is called.
SFDUs need not neccesarily be send over an ISO-OSI network, it could be send via e-mail, on a discette, on a tape, etc. In this case there is no need for encoding the SFDU.

SFDU Receiver:
The **SFDU receiver** autonomously receives SFDUs from the ISO-OSI network represented in accordance with the Presentation Layer Protocol. It performs the inverse decoding function of the SFDU sender and stores the received SFDU in the database. As an alternative, the SFDU receiver may be combined with and produce input directly to the SFDU interpreter to be described next.
We note, that the data contents created at the SFDU originator's end of the system by an application process is delivered in a semantically unaltered form to the peer level of the recipient system regardless of the transmission and storage means. What happens in the communication path is not of concern to the SFDU domain.

SFDU Interpreter:
All sites in the open system exchanging information in SFDUs must be equipped with software to read and interpret SFDUs. The function performed by the **SFDU interpreter** is very simple and basically consists in separating SFDUs into their Type Fields and Value Fields.

Data Format Interpreter:
For each DDL used at a given site there must be software to read and interpret the data format definitions and the data represented in accordance with the data formats.
For each DDL used at a given site, the data dictionary schema defines to the extent possible all syntactic rules for defining data formats using the DDL. This implies that the data dictionary schema becomes an active part of the interpreter for data formats and that any data format can be stored in the extension of the data dictionary schema. If the description of the DDL is complete, then the core DBMS takes on the role of data format and data interpreter for the DDL.
The **data format interpreter** interprets data from a Value Field using the data format identified by the corresponding Type Field. The data format interpreter may be combined with and receive input directly from the SFDU interpreter described above.
There are basically two possible situations discussed in details in the following.
In the first situation, the data format definition is already stored in an interpreted form in the data dictionary at this site. Using this data format definition, the data format interpreter and the core DBMS can interpret and store the data contents as data under the data format definition.
In the second situation, the data format definition is not currently stored at this site. The data format interpreter requests the data format definition identified by the Type Field. The data format definition will be received in a separate SFDU and will be interpreted using the data dictionary schema and the data format definition will be stored in the data dictionary. This brings us back to the first situation again, and the interpretation of the original data contents can continue.
We note that **exactly** the same software is needed by the SFDU recipient to interpret data format definitions and data. The underlying assumption is that we have used the **same set of concepts for formatting data and for formatting data formats.** This assumption is equivalent to the basic assumption used for the intension- extension dimension of data description in a Self-Describing Database System.

One of the problems that usually requires a considerable amount of attention in information interchange systems is the conversion of primitive types between different hardware, operating systems, database systems, and programming languages. This problem has a local solution if we include an explicit description of the required precision in the data format definition.

<u>Example</u>

```
meterological_info=
    time: yy.mm.dd.hh.mm.ss: 0..100,1..12,1.boxht1,0..23,0.boxht9,0.boxht9
    location=
        longitude: hh.mm.ss: 0.boxht9,0.boxht9,0.boxht9
        latitude: degree n|s: 0..90,n|s
        altitude: feet: dddddd;
    temperature: °F: ddddd.d
    humidity: %: 0..100
    pressure: mb: 400..1600;
```

This data format definition describes the units of measure and the required precision of a physical representation of the data.
The DDL is very close to a relation definition in for example SQL [ISO 86a]. What is intensionally left out from the definition is a specification of the domains, i.e. the primitive data types. With the required precision specified we need an algorithm that can pick an appropriate set of domains for representing the data in this particular system. This means, that we need only n data conversion algorithms for n systems, whereas the order of $n^2/2$ algorithms are needed to convert between n systems in the general case.

5. Data Dictionary for Information Interchange

To support information interchange between Self-Describing Database Systems using SFDUs, the following requirements must be satisfied.

The data dictionary data must define data structures for storing received, but not interpreted SFDUs. The SFDUs must be stored in such a way that they can be retrieved for interpretation at a later time. This is best done by stripping off and storing the Type Field separately and using it as the search key when the corresponding Value Field is retrieved for interpretation. This organization also supports the storage of SFDUs that are assembled, but not yet sent. And, it allows for more permanent storage of not interpreted SFDUs.

The data dictionary schema must to the extent possible describe the syntax and construction rules for SFDUs. This will allow the data dictionary schema to be used by the SFDU interpreter, and it will allow the components of an SFDU to be stored when the SFDU is interpreted. It will also allow the data dictionary schema to be used be the syntax directed SFDU assembler to ensure correctly assembled SFDUs. It will allow the data dictionary schema to be used to control operations that create SFDUs from other SFDUs. It will finally allow the syntax and construction rules for SFDUs to evolve; this would not be possible without reprogramming if the syntax and construction rules were not explicitly represented in the data dictionary schema.

The data dictionary schema must to the extent possible describe the syntax of any standardized DDL used at this site. This will allow the data dictionary schema to be used by the data format interpreter for interpreting data formats, and it will allow data format definitions to be stored as data dictionary data. This will also allow browsing of data formats stored as data dictionary data, and it will allow the data dictionary schema to control all operations on data formats, such as copying, modifications, and operations reusing data formats to create new data formats. With the data format stored as data dictionary data the data format interpreter will be able to interpret the value part of an SFDU and store it as interpreted data.

The data dictionary schema must completely control the assignment of Format Identifiers for new data formats. The correct assignment of Format Identifiers must be guaranteed for each data format definition; this process can be automated. Format Identifiers can never be changed or reused, and the assignment of them should not be under user control. Format Identifiers must be strings without embedded encoded meaning; this can only be guaranteed if their assignment is not under user control.

The data dictionary schema must completely model the relationship between Format Identifiers and physical addresses in the communication net.

The data dictionary schema should - in addition to the above - automatically enforce any additional policies to the extent possible. With an explicitly represented model for the definition, management and use of SFDUs the data dictionary will be flexible wrt. changes in policies. Without an explicit representation the system will not be able to accomodate to changes without reprogramming. If any formal rules for controlling the proliferation of data format definitions are defined, then these should be included in the data dictionary schema.

No commercially available data dictionary system or database system has a data model which is sufficiently powerful to define a data dictionary schema fulfilling the above requirements. The data models are especially weak when it comes to describing constraints and operations on the stored data formats and SFDUs. If a commercially available database system is used for the implementation, it is therefore desirable that the data dictionary schema be **encapsulated** in a software module that provides a set of **primitive operations** on data formats and SFDUs. This could be implemented using triggers and event-procedures. These primitive operations should be the **only** operations that can access and change the data formats and SFDUs stored in the data dictionary directly. Other **compound operations**, such as the Data Management Tools, should access and change data formats and SFDUs stored in the data dictionary only through calls of the primitive operations.

References

ANSI/SPARC X3H4 part 1 - 4
American National Standards Institute: (Draft Proposed) "American National Standard information resource dictionary system: part 1 - Core standard, part 2 - Entity-level security, part 3 - Application program interface, part 4 - Support of standard data models," ANSI X3H4, American National Standards

Institute, New York 85.

Billingsley 86
Billingsley, F.: "Draft Specification for a General Data Interchange Language,", May 8, 1986. Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109.

Burns 86
Burns, T.; Fong, E.; Jefferson, D.; Knox, R.; Mark, L.; Reedy, C.; Reich, L.; Roussopoulos, N.; Truszkowski, W.: "Reference Model for DBMS Standardization," ACM SIGMOD Records, March 1986.

CCSDS 86
Consultative Committee for Space Data Systems,: "Standard Formatted Data Units - Structure and Construction Rules," Red Book, Issue-1, September 1986. CCSDS Secretariat, Communications Division (Code-TS), NASA, Wash., DC 20546.

Dolk 87
Dolk, D.R. and Kirsch II, A.: "A Relational Information Resource Dictionary System," Communications of the ACM, Volume 30, number 1, January 1987.

Griethuysen 82
Griethuysen, J.J. van (ed.): "Concepts and Terminology for the Conceptual Schema and Information Base." ISO/TC97/SC5/WG3 - N695, 1982.

ISO 81
International Standards Organization: "Information Processing Systems - Open Systems Interconnections - Basic Reference Model." Computer Networks 5, 1981.

ISO 86a
International Standards Organization TC97/SC21 N1479: (Working Draft) "Database Language SQL2," October 1986.

ISO 86b
International Standards Organization TC97/SC15 N198: (Working Draft) "ISO-8211 Specification for a Data Descriptive File for Information Interchange," National Bureau of Standards [Docket No. 50719-5119], 1986.

Mark 85a
Mark, L.: "Self-Describing Database Systems - Formalization and Realization," TR-1484 Computer Science Department, University of Maryland, U.S.A., 1985.

Mark 85b
Mark, L. and Roussopoulos, N.: "The New Database Architecture Framework - A Progress Report," in (Ed.) Sernadas, A., Bubenko, J., and Olive, A., "Theoretical and Formal Aspects of Information Systems," North-Holland 1985.

Mark 86
Mark, L. and Roussopoulos, N.: "Metadata Management," IEEE Computer Magazine Special Issue on "Future Directions in Database Systems - Architectures for Information Engineering", December 1986.

McCarthy 84
McCarthy, J.L.: "Scientific Information = Data + Meta-data," Technical Report, Lawrence Berkeley Laboratory, University of California, Berkeley, CA 94720.

Roussopoulos 83
Roussopoulos, N. and Mark, L.: "A Self-Describing Meta-Schema for the RM/T Data Model," In Proc. IEEE Workshop on Languages for Automation, IEEE Computer Society Press, 1983.

DATA RETRIEVAL IN A DISTRIBUTED TELEMETRY GROUND DATA SYSTEM

Carol J. Steinberg[1]
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109

1.0 INTRODUCTION

Flexible management and manipulation of large volumes of telemetry data for the Jet Propulsion Laboratory's unmanned deep-space missions has motivated the design of the Space Flight Operations Center (SFOC) distributed ground data system [EBER85] [JPL85]. SFOC supports real-time and non-real-time spacecraft telemetry data management, monitoring, and analysis as depicted in Figure 1. A Central Data Management (CDM) node lies at the heart of the system for managing the telemetry data. Application programs residing at SFOC workstation and function-dedicated computer nodes (herein referred to collectively as workstations) are provided with a set of services to access the telemetry data under CDM control. This paper proposes a concept for a runtime data retrieval subsystem for SFOC that is referred to as the Session Manager.

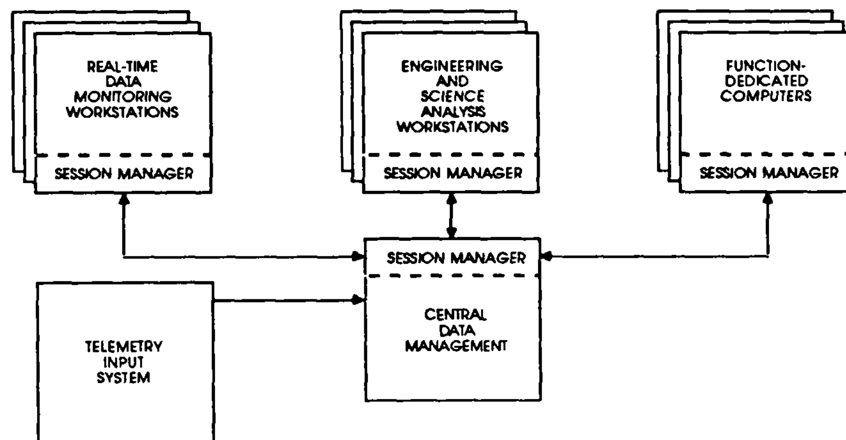


FIGURE 1. SFOC LOGICAL CONFIGURATION FOR THE DATA MANAGEMENT NODE AND CLIENT NODES

The Session Manager offers workstations a set of programmer services enabling them to establish a session with CDM, submit data requests to CDM, and process the data results returned from CDM. The Session Manager must gather the data satisfying the submitted data request from the storage structures at CDM and distribute that data to the requesting workstation application program. This scenario is not unlike what many commercial data management products offer their customers. But the Session Manager is set apart by key features mandated by the SFOC environment:

1) Support of several distinct data retrieval control mechanisms to accommodate real-time as well as non-real-time data retrieval and delivery;

2) Ability to access a variety of storage media via a variety of access strategies in a manner that is transparent to the user;

---

[1] Author's current employer:
Philips Ultrasound International
2722 South Fairview Street, Santa Ana, California 92704

3) A programmer's runtime interface modeled after those used for commercial relational database management systems [BLI85] but generalized to any form of record by record data retrieval;

4) Data retrieval by predefined commands identified by name and parameter list, and whose definitions reside at CDM;

5) Design that is both flexible and extensible to insure against locking the SFOC project into a fixed set of vendors and commercial products, and to allow for painless growth as SFOC matures over its projected 15 year life span.

2.0 SESSION MANAGER ARCHITECTURE

Figure 2 represents the functional layers for access of data at CDM. The Central Data Store represents the on-line data stored at CDM. The architecture of CDM includes a mix of storage media and data access strategies to accommodate the large data volumes and data usage variations. Consequently, several functional components must exist for extracting this data from CDM. These include File Transfer, Forms Interface, and Session Manager components. The Session Manager itself is composed of two layers: the Central Session Manager and the Session Manager Runtime Services. The Session Manager Runtime Services along with CDM Distributed Utilities provide a collection of non-procedural services and tools to the workstation environment.

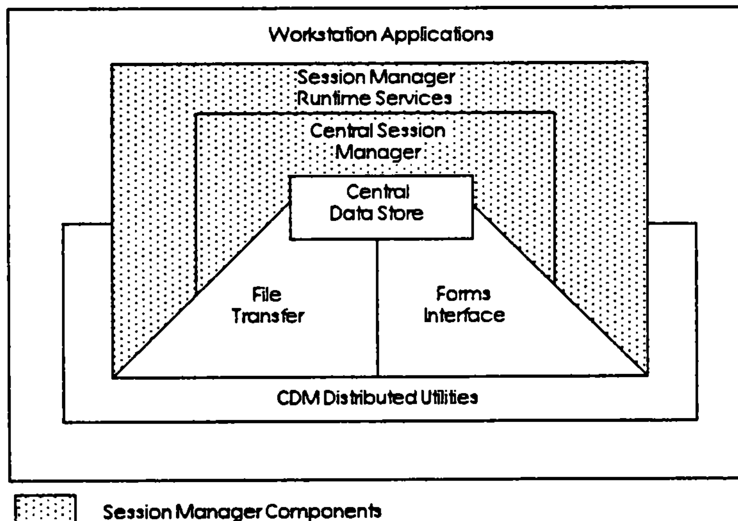


FIGURE 2. CENTRAL DATA STORE ACCESS LAYERS

The Session Manager Runtime Services (SMRS) is the programmer's "user interface" to CDM. The SMRS accepts session directives from the client application and conveys them to the Central Session Manager (CSM). Consequently, the CSM regulates the processing of data requests for a given CDM session. Given a data request, the CSM accesses the Central Data Store for data satisfying the client's data request and exports the data to SMRS for subsequent presentation to the requesting client.

The division of function in the Session Manager is a natural boundary for physical distribution of function across SFOC logical nodes. The SMRS resides on the workstations while CSM is resident at the CDM node. The SMRS and CSM communicate with one another via network services and a set of well-defined control and data messages.

An invocation of CSM (CSM server) exists for every active CDM session. An application may have any number of active CDM sessions. Figure 3 illustrates the interaction of workstation applications, SMRS and CSM, where application A has two active CDM sessions, and application B has one open session. One CSM server is always listening for a new workstation client.

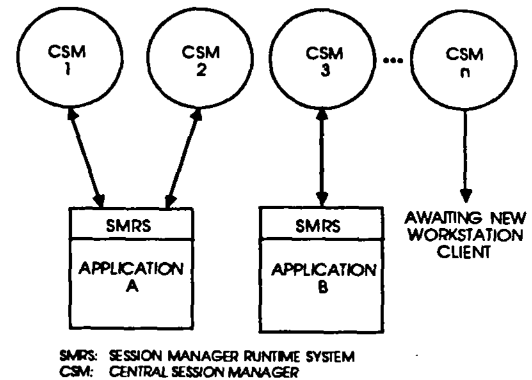


FIGURE 3. APPLICATION INTERACTION WITH SESSION MANAGER

3.0 CENTRAL SESSION MANAGER OVERVIEW

CSM coordinates data retrieval from CDM for workstation clients via received SMRS control and data request directives. CSM responds accordingly with status and data result messages. CSM's main responsibilities include 1) ensuring that the user establishing the session is a legal user, 2) validating data requests, 3) controlling data request execution, 4) accessing storage structures, and 5) exporting data results and status information to the SMRS.

The CSM is composed of a set of distinctive states, some of which are tightly coupled to the messages received from SMRS. Figure 4 represents the state transitions in fairly simple terms.

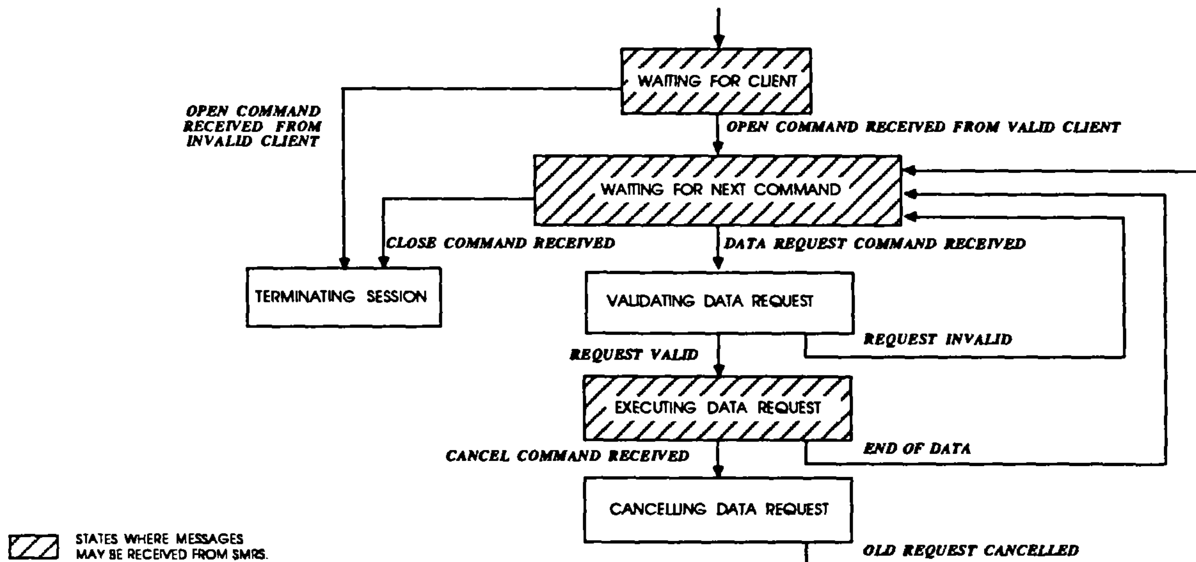


FIGURE 4. STATE TRANSITION DIAGRAM FOR ONE CENTRAL SESSION MANAGER SERVER

After a valid client has initiated a session with CDM, the CSM awaits a command message from the SMRS. The *WAITING FOR NEXT COMMAND* state is the resting state to which the CSM always arrives until a close command is received. If a data request is received, it is validated and the CSM proceeds to execute the data request. In the course of executing, the data are accessed from the Central Data Store, then packaged and exported to the SMRS via network services. (Network services handle the buffering of exported data for both client and server.) When all the data have been exported for a given data request, the CSM waits for the next command message. Should the client wish to abort a data request while it is executing, a cancel command may be sent to stop execution. Again, the CSM returns to the *WAITING FOR NEXT COMMAND* state. It is from this state that a new data request may be accepted for execution, or from which the session may be terminated.

3.1 EXECUTION OF DATA REQUESTS

The state *EXECUTING DATA REQUEST* can be considered a task that involves interaction of two functional layers: 1) a control layer for managing the execution of distinct classes of data requests, and 2) a data access layer for managing data retrieval from distinct storage structures. Figure 5 illustrates how the layers are configured. A "processing path" exists for each class of data request. The CSM follows one processing path per submitted data request. A processing path has at its disposal all of the available data access methods, using one per submitted data request.[2] The important point here is that the CSM may be expanded to handle new classes of data requests as support is required. Similarly, new storage structures and their associated access methods may be supported as CDM hardware and software tools evolve.

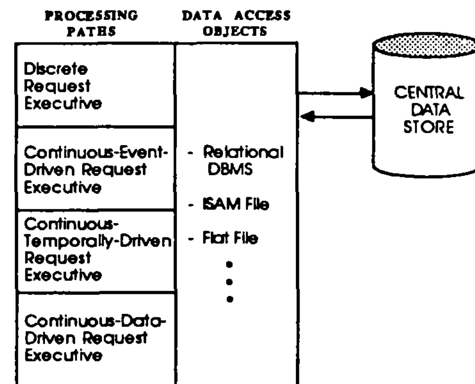


FIGURE 5. FUNCTIONAL LAYERS FOR DATA REQUEST EXECUTION

3.1.1 Data Request Classes

The two main data request classes are discrete and continuous. The discrete requests are those in which a data request is received, validated, and executed immediately. Only current data in the Central Data Store at the time the request is processed are available for distribution. When all data satisfying the request have been accessed by the CSM, the request is complete. For discrete requests, transition from the *EXECUTING DATA REQUEST* state to the *WAITING FOR NEXT COMMAND* state is on the end-of-data condition or upon receipt of a cancel command.

Continuous requests are those in which a data request is received, validated, and executed until a cancel command is received, at which point the CSM returns to the *WAITING FOR NEXT COMMAND* state. There are three distinct control mechanisms for continuous requests: 1) event-driven, 2) temporally-driven, and 3) data-driven. Continuous requests are targeted for data received in real-time where data exportation is triggered by one of the three control mechanisms listed. For instance, workstations may be alerted of real-time telemetry data that is outside of acceptable limits via an event-driven request control mechanism. This same control mechanism could be used to distribute maps for decoding incoming telemetry frames as maps become available or effective.

Based on this discussion, four data request classes, hence four processing paths have been identified: 1) discrete, 2) continuous-event-driven, 3) continuous-temporally-driven, and 4) continuous-data-driven. Each one is represented in Figure 5 as a distinct element in the control layer for managing the execution of data request classes.

---

[2] In the SFOC environment, data is expected to be partitioned across storage structures in a manner that is tailored to usage such that a candidate data request need never span a set of storage structures. Hence, the Session Manager has not been designed with a "merge" capability across storage structures. That a "processing path" has at its disposal all data access objects means that it can accommodate a data request operating on any one data access object known to it.

3.1.2 Data Access Objects

The Central Data Store is composed of a variety of storage structures which are themselves composed of a mix of media and data access strategies. Each storage structure has an associated set of operations for managing it. The storage structure along with its operations can be termed a "data access object". Examples of data access objects include relational DBMS, optical disk files, ISAM disk files, flat disk files, and tape.

3.2 REQUEST CONTROL CATALOG

The Request Control Catalog is the heartbeat of the Session Manager. It is through the Request Control Catalog that the CSM knows how to process a data request submitted by a workstation application. Data requests are predefined at CDM and are identified by name. The Request Control Catalog captures the following information about the data request: 1) a data request follows one processing path, e.g., discrete request, 2) a data request accesses one storage structure, e.g., relational DBMS, 3) a data request has zero or more substitutable parameters used as selection criteria for the request, and 4) a data request has a set of one or more data-items composing a target list of results to be returned by the request.

4.0 SESSION MANAGER RUNTIME SYSTEM OVERVIEW

The SMRS facilitates development of application programs which use CDM data resources. SMRS drives a CDM session via control and data request messages sent to the CSM. The SMRS must, in turn, respond to result and status messages returned by the CSM.

SMRS is composed of a set of functions for CDM session control, query, and result processing. Table 1 lists the fundamental functions of the SMRS in a synchronous environment. Note that this interface is a subset of function calls typically available in runtime systems of commercial relational databases. But in this case, the interface is applied to all program delivered data, independent of central data storage structure. Moreover, its modular nature lends itself nicely to expansion to accommodate expected new features; for example, asynchronous data request and delivery.

CLASS	FUNCTION	DESCRIPTION
Control	OPEN	Establish a CDM session.
	CLOSE	Close a CDM session.
	CANCEL	Cancel the current data request.
Query	REQUEST	Submit a predefined data request to CDM for runtime execution.
Process	BIND	Associate a program variable to a retrieved target list element.
	FETCH	Fetch a retrieved result record.
	DESC	Get datatype information about a target list element for the currently active data request.

Table 1. Basic Set of Session Manager Runtime System Functions

The application programmer must be aware of the data request's name, its parameters (data selection criteria), and the target list of data-items returned with each FETCH. Recall that this information is maintained in the Request Control Catalog referenced by the CSM.

The SMRS functions are independently invocable. Nonetheless, there is an inherent ordering to the functions as depicted in the state transition diagram of Figure 6. All SMRS directive messages sent to CSM are acknowledged via status and data messages returned to SMRS. CSM status messages indicate receipt of a valid SMRS directive, error conditions, CDM target list specification, end of data conditions, etc.

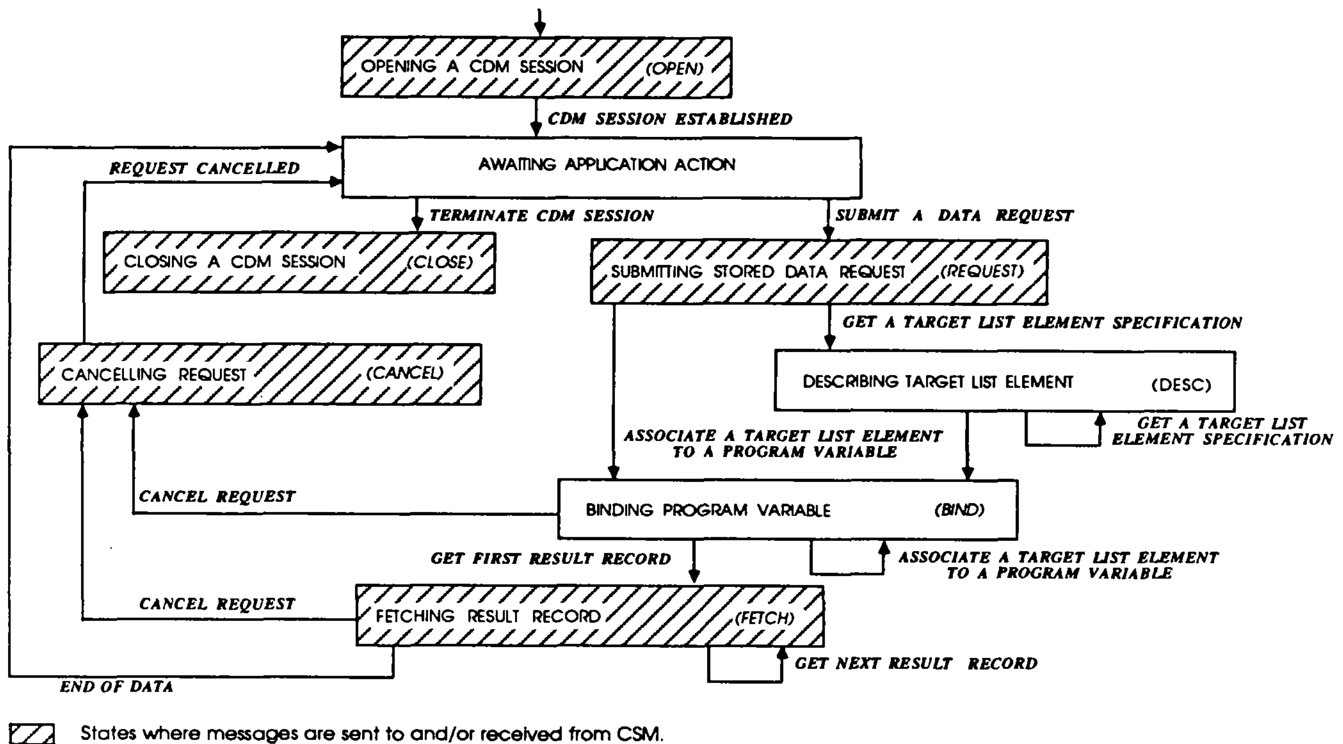


FIGURE 6. STATE TRANSITION DIAGRAM FOR SESSION MANAGER RUNTIME SERVICES FUNCTION INVOCATION

For a given CDM session, the functions communicate with one another via a shared SMRS Control Block which maintains information about that session's current runtime state. Hence, several data requests may be processed in a serial fashion in any CDM session. Furthermore, several SMRS Control Blocks, hence several CDM sessions, may coexist in a given application. This allows for submission of multiple data requests to CDM for parallel processing.

5.0 SESSION MANAGER AS AN EVOLVING SYSTEM

The Session Manager can potentially support other kinds of features. What has been described in this paper has been specific for sessions run in a synchronous mode. (A small portion of this has been prototyped for discrete requests.) It is conceivable that users may want to submit data requests for later execution, similar to batch runs, with data results returned to the same or even different workstation. In future versions, it may be desirable to loosen control over data request structure and instead of predefining all data requests at CDM, allow users to submit requests composed on-the-fly from a query language.

SMRS and CSM design is modular to facilitate expansion. As already pointed out, CSM design has been equipped to accept new processing paths for new data request classes. Also, as new data storage tools are made available to CDM, new data access objects may be added to the pool of objects available to the existing processing paths for data retrieval. The SMRS is similarly expandable. As new features are added to the Session Manager, for instance, data requests submitted for later execution, new functions may be easily added to the SMRS' user interface to accommodate these features. Finally, communications between the SMRS and CSM occur within well defined states. Additional message classes may be easily accommodated as functionality requires. Session Manager is an evolving subsystem and should evolve throughout the lifetime of SFOC.

ACKNOWLEDGEMENTS

I would like to thank Joan Van Cleef for preparation of the diagrams and manuscript.

The work described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under National Aeronautics and Space Administration contract.

REFERENCES

[BLI85] *Fortran Programmer's Guide to IDMLIB, The Intelligent Database Machine*, Britton-Lee, Incorporated, November 1985

[EBER85] Ebersole, M.M., *The Space Flight Operations Center Development Project*, Journal of the British Interplanetary Society, Vol. 38, pp. 472-478, 1985

[JPL85] *JPL Future End-to-End Information System Architectural Design*, JPL Publication D-952, April 1985

PRELIMINARY CALL FOR PAPERS

1988 ACM-SIGMOD

International Conference on Management of Data
June 1-3, 1988 Chicago, Illinois

Organizing Committee
General Chairpersons:

Dina Bitton
University of Illinois at Chicago

Peter Scheuermann
Northwestern University

Tutorials:	Clement Yu, University of Illinois at Chicago
Panels:	Meral Ozsoyoglu, Case Western Reserve University
Treasurer:	Udai Gupta, AT&T, Naperville
Local Arrangements:	Leszek Lilien, University of Illinois at Chicago
Registration:	Michael Carey, University of Wisconsin, Madison
Publicity:	Edward Omiecinski, Georgia Institute of Technology
European Coordinator:	Witold Litwin, INRIA, France

The 1988 ACM-SIGMOD Conference will feature two concurrent tracks: *Concepts & Techniques* and *Applications & Implementations*, each representing a vertical slice of Database Technology. *Concepts & Techniques* will focus on theoretical and algorithmic aspects of data and knowledge management encompassing data models, query languages, deductive databases, database design, access methods, query optimization, concurrency control and recovery. *Applications & Implementations* will focus on characterization of application profiles, requirement analysis, tools for logical and physical design, architectures for parallel and distributed database systems, and experimental performance studies.

Our goal is to have a balanced conference representing both the real users's needs, problems and experiences and possible solutions at the algorithm and system level.

Papers are solicited for both tracks. Four copies of each paper should be mailed to the appropriate Program Committee chairman.

Program Committee Chairpersons:

Concepts & Techniques Track

Per-Ake Larson
Department of Computer Sciences
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1
palarson%waterloo@relay.cs.net

Applications & Implementations Track

Haran Boral
MCC
3500 West Balcones Center Drive
Austin, Texas 78759
boral@mcc.com

Important Dates

Papers and Panels Submission Deadlines:	December 4, 1987
Notification of Acceptance:	February 12, 1988

Call for Papers and Participation

Second International Conference on Expert Database Systems

George Mason University

April 25-27, 1988

The Sheraton Premiere at Tysons Corner
Tysons Corner, Virginia

Sponsored by:

George Mason University

In Cooperation With:

American Association for Artificial Intelligence
Association for Computing Machinery -- SIGART and SIGMOD
IEEE Computer Society -- T. C. on Data Base Engineering

Conference General Chairman
Edgar H. Sibley
George Mason University

Program Chairman
Larry Kerschberg
George Mason University

Program Committee

Robert Abarbanel, *IntelliCorp*
Hideo Aiso, *Keio University*
Antonio Albano, *U. of Pisa*
Stephen J. Andriole, *GMU*
Robert Balzer, *USC - ISI*
Francois Bancilhon, *Altair, France*
Alex Borgida, *Rutgers University*
Don Batory, *U. of Texas*
Michael L. Brodie, *CCA*
Janis Bubenko, *U. of Stockholm*
Peter Buneman, *U. of Pennsylvania*
Stefano Ceri, *Politecnico di Milano*
Umesh Dayal, *CCA*
Mark Fox, *Carnegie-Mellon University*
Antonio L. Furtado, *Rio Sci. Center, IBM Brasil*
Herve Gallaire, *ECRC, Munich, FRG*
Barbara Hayes-Roth, *Stanford University*
Yannis Ioannidis, *Univ. of Wisconsin*
Sushil Jajodia *Naval Res Lab*
Matthias Jarke, *U. of Passau, FRG*
Jonathan King, *Teknowledge, Inc.*
Roger King, *U. of Colorado*
Robert Meersman, *Tilburg University*
Tim H. Merrett, *McGill University*
Matthew Morgenstern, *SRI International*
John Mylopoulos, *U. of Toronto*
Sham Navathe, *U. of Florida*
Erich Neuhold, *GMD, FRG*
Setsuo Ohsuga, *U. of Tokyo*
D. Stott Parker, Jr., *UCLA*
Alain Pirotte, *Philips Lab, Belgium*
W. Don Potter, *U. of Georgia*
Larry Reeker, *BDM Corporation*
Nick Roussopoulos, *U. of Maryland*
Erik Sandewall, *Linkoping University*
Timos Sellis, *U. of Maryland*
John Miles Smith, *CCA*
Reid Smith, *Schlumberger-Doll Research*
Arne Solvberg, *U. of Norway*
John Sowa, *IBM SRI*
Jacob Stein, *ServioLogic*
Michael Stonebraker, *UC - Berkeley*
Adrian Walker, *IBM T.J. Watson Center*
Gio Wiederhold, *Stanford University*
Andrew B. Whinston, *Purdue University*
Eugene Wong, *UC - Berkeley*
Carlo Zaniolo, *MCC*

Tutorial and Panel Coordinator
Lucian Russell, *Computer Sciences Corp.*

Conference Coordinator
Nancy D. Joyner, *GMU*

Exhibits Coordinator
Diane Entner, E-Systems

Publicity Chairman
Jorge Diaz-Herrera, *GMU*

Conference Objectives

The International Conference on Expert Database Systems has established itself as a *leading edge* forum that explores the theoretical and practical issues in making database systems more intelligent and supportive of Artificial Intelligence (AI) applications. Expert Database Systems represent the **confluence** of R&D activities in Artificial Intelligence, Database Management, Logic, Information Retrieval, and Fuzzy Systems Theory. It is precisely this **synergism** among disciplines which makes the Conference both stimulating and unique.

Topics of Interest

The Program Committee invites original theoretical and application papers (of approximately 5000 words) addressing (but not limited to) the following areas:

Theory of Knowledge Bases (including knowledge representation, knowledge models, knowledge indexing and transformation, knowledge servers, and formal semantics of knowledge/data bases).

Object-Oriented Systems (including object-oriented data models, query languages, transaction management, version control, and modeling applications for enterprises, CAD/CAM, VLSI, material properties databases, knowledge-based software engineering environments, etc.).

Reasoning on Knowledge/Data Bases (including reasoning under uncertainty, sensor fusion, non-monotonic reasoning, analogical reasoning, deductive databases, logic-based query languages, semantic query optimization and constraint-directed reasoning).

Knowledge Management (including methodologies for knowledge acquisition, the knowledge engineering process, constraint and rule management, knowledge-based requirements gathering and specification, and knowledge administration).

Distributed Knowledge/Data Bases (including loosely- and tightly-coupled architectures, intelligent query decomposition and processing, federated architectures, distributed problem-solving, and blackboard techniques for distributed control).

Intelligent Database Interfaces (including expert system -- database communication, knowledge gateways, knowledgeable user agents and browsers).

Natural Language Interaction (including question-answering, extended responses, cooperative behavior, explanation and justification).

Please send five copies of papers by October 14, 1987 to

Professor Larry Kerschberg
Dept. of Information Systems and Systems Eng.
George Mason University
4400 University Drive
Fairfax, Virginia 22030, USA

Important Dates

Submission Deadline:	October 14, 1987
Acceptance Notification:	December 15, 1987
Camera-Ready Version :	February 1, 1988
Conference Dates:	April 25-27, 1988

PRELIMINARY PROGRAM

International Conference on Data and Knowledge Systems for Engineering and Manufacturing

October 19-20, 1987

The Hartford Graduate Center
Hartford, Connecticut

THE CONFERENCE

This conference will bring together leading researchers and practicing engineers for an intensive series of presentations and discussions on systems, methods, and techniques for improving the productivity of manufacturing and engineering processes. It will provide a forum for interactions between university researchers and design and manufacturing engineers on intelligent data and knowledge systems for manufacturing automation. It will also offer an opportunity for the discussion and promotion of research and development cooperations in manufacturing automation between universities, industries, and government agencies.

CONFERENCE PROGRAM

The conference program includes paper sessions, panel discussions, and plenary sessions. Paper sessions address:

- Distributed Architectures for Manufacturing and Engineering
- Data Objects and Models
- Case Studies
- CAD/CAM Databases
- Expert and Knowledge-based Systems
- Intelligent User Interface Systems

Panel sessions include:

- The DOD Engineering Information Systems; **Chair:** *Robert Winner, Institute for Defense Analyses*
- Heterogeneous Database Support for Manufacturing and Engineering; **Chair:** *Sandra Heiler, Computer Corporation of America*
- Joint R&D Between Government, Industry, and University; **Chair:** *Howard Bloom, National Bureau of Standards*

The plenary sessions include the keynote address and an invited speaker.

Keynote Address:
Michael J. Wozny, National Science Foundation

Invited Speaker:
Robert Booth, General Motors

ORGANIZATION

The conference will be held at the Hartford Graduate Center, 275 Windsor Street, Hartford, Connecticut. The conference hotel is the Holiday Inn, Hartford - Downtown, 50 Morgan Street, Hartford Connecticut, (203) 549-2400. Reservations should be made by September 30, 1987.

For a copy of the Advance Program contact:

David L. Spooner
Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York 12180

Phone: (518) 276-6890

STEERING COMMITTEE

Fred Maryanski, Conference Chairman
T. C. Ting, Program Chairman
Tim Martyn, Local Arrangements
David Spooner, Publicity
Dong-Guk Shin, Registration and Treasurer
Peter Luh, Speakers

PROGRAM COMMITTEE

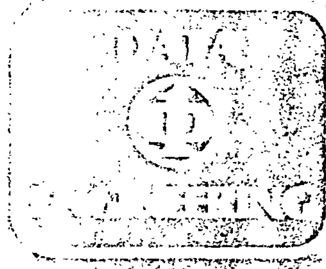
David Anderson	Frank Manola
Howard Bloom	Edith Martin
Stefano Ceri	George Nagy
T. C. Chen	Nicholas Roussopoulos
J. D. Dorng	Jack Smith
Richard Garrett	Michael Stonebraker
Tadao Ichikawa	Stanley Su
James Jordan	Herbert Weber
Erlang Jungert	Gio Wiederhold
Tok Wang Ling	Robert Winner
Vincet Lum	Stanley Zdonik

SPONSORED BY

The University of Connecticut
The Hartford Graduate Center
Rensselaer Polytechnic Institute
National Science Foundation (requested)
Institute for Defense Analyses

IN COOPERATION WITH

SIGMOD, ACM
IEEE Computer Society



COMMITTEE

Steering Committee:
C. V. Ramamoorthy, University of California, Berkeley
P. Bruce Berra, Syracuse University
Gio Wiederhold, Stanford University

General Chairperson:
Benjamin W. Wah, University of Illinois

Program Chairperson:
John Carlis, University of Minnesota

Program Co-Chairpersons:
Tadeo Ichikawa, Hiroshima University, Japan
Sushil Jajodia, Naval Research Laboratory
Iris Kameny, Rand Corporation
Roger King, University of Colorado
Witold Litwin, INRIA, Le Chesmay, France
Z. Meral Ozsoyoglu, Case Western University
Joseph Urban, University of S W Louisiana

Tutorials:
Amit P. Sheth, Honeywell Corporation

Industrial and Inter-Society Coordinator:
Dick Shuey, Consultant
2330 Rosendale Rd., Schenectady, NY 12309

Awards:
K. H. Kim, University of California, Irvine

Publicity:
Jie-Yong Juang, Northwestern University

International Coordination:
G. Schlageter, Fern Universitat, Hagen, FRG

Treasurers:
Kate Baumgartner, University of Illinois
Aldo Castillo, Cray Research

Local Arrangements:
Walter Bond, SDC
Homideh Afsarmanesh, Cal State, Dominquez Hills

Committee Members (Tentative):

Mohan Ahuja
A. K. Arora
J. L. Baer
Farokh B. Bastani
Don Batory
Kate Baumgartner
G. Belford
Bharat Bhargava
Richard Braegger
C. Robert Carlson
Nick Cercone
David Du
Earl Ecklund
Ramez El-Masri
Domenico Ferrari
Hector Garcia-Molina
Georges Gardarin
Robert Gerber
Sakti P. Ghosh
Georg Gottlob
Lee Hollaar
Yang-Chang Hong
David K. Hsiao
H. Ishikawa
Hemant K. Jain
Won Kim
Dan Kogan
Walter Kohler
Robert Korfhage
Tosiyasu L. Kunii
Winfried Lamersdorf
James A. Larson
Matt LaSaine
W-H. Francis Leung
Guo-Jie Li
Victor O.K. Li
Yao-Nan Lien
Leszek Lilien
Witold Litwin
Jane W.S. Liu
Ming T. (Mike) Liu
Raymond A. Liuzzi
Vincent Lum
Yuen-Wah Eva Ma
Mamoru Maekawa
Sal March
Gordon McCalla
J. Eliot Moss
Tadeo Murata
Philip M. Neches
Erich J. Neuhold
G. M. Nijssen
Ole Oren
Gultekin Ozsoyoglu
C. Parent
J. F. Paris
Krithi Ramamrithan
David Reiner
Gruia-Catalin Roman
Domenico Sacca
Giovanni Maria Sacco
Vikram Saletore
Sharon Salveter
Phillip Sheu
Edgar Sibley
John F. Sowa
David Spooner
David Stemple
M. Stonebraker
Stanley Su
Denji Tajima
Marjorie Templeton
A. M. Tjoa
Mas Tsuchiya
Yosihisa Udagawa
Susan D. Urban
Patrick Valduriez
Yann Viemont
Kyu-Young Whang
Chao-Chih Yang
S. Bing Yao
Clement Yu

SCOPE

Data Engineering is concerned with the semantics and structuring of data in information system design, development, management, and use. It encompasses both traditional applications and issues, and emerging ones. The purpose of this conference is to provide a forum for the sharing of practical experiences and research advances from an engineering point of view among those interested in automated data and knowledge management. Our expectation is that this sharing will enable future information systems to be more efficient and effective, and future research to be more relevant and timely.
We are particularly soliciting industrial contributions and participation. We know it is vital that there be a dialogue between practitioners and researchers. We look forward to reports of experiments, evaluations, and problems in information system design and implementation. Such reports will be processed, scheduled, and published in a distinct track.

TOPICS OF INTEREST

We invite you to submit papers on topics including but not limited to these:

- Applications
- Autonomous, distributed systems
- Data engineering tools
- Data management methodologies
- Data security and integrity
- Design of knowledge-based systems
- Distribution of data and knowledge
- Expert systems
- Architectures for database and knowledgebase systems
- Logical and physical database design
- Performance evaluation
- Statistical databases

PAPER SUBMISSIONS

Each paper's length should be limited to 8 proceedings pages, which is about 5000 words, or 25 double-spaced typed pages. Four copies of completed papers should be mailed before June 15, 1987 to:
John V. Carlis, Computer Science Department, University of Minnesota, 207 Church Street SE, Minneapolis, MN 55455, (612) 625-6092; carlis @umn-cs.

TUTORIALS

The days before and after the conference will be devoted to tutorials. Proposals for tutorials on Data Engineering topics, especially advanced topics, are welcome.
Send proposals by June 15, 1987 to:
Amit P. Sheth, Honeywell Computer Sciences Center, 1000 Boone Avenue North, Golden Valley, MN 55427, (612) 541-6899.

CONFERENCE TIMETABLE AND INFORMATION:

Papers due:	June 15, 1987
Tutorial proposals due	June 15, 1987
Acceptance letters sent:	September 15, 1987
Camera-ready copy due:	November 15, 1987
Tutorials:	February 1 and 5, 1988
Conference:	February 2-4, 1988

For further information contact the General Chairperson: **Benjamin W. Wah,** Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801, (217) 333-3516; wah%uicsld @uiuc.arpa.

AWARDS, STUDENT PAPERS AND SUBSEQUENT PUBLICATION

Awards will be given to the best paper and to the best student paper (denoted as such when submitted and authored solely by students). The latter will receive the K. S. Fu award, honoring one of the early supporters of the conference. Up to three grants of $500 each to help defray travel costs of student authors. Outstanding papers will be considered for publication in the IEEE Computer Society publications: Computer, Expert, Software, and Transactions on Software Engineering. For more information contact the general chairman.

EPILOG

Several hundred people have been involved in the data engineering conferences as committee members, reviewers, authors, and attendees. We have benefited by being involved, and extend an invitation to you to participate.

THE COMPUTER SOCIETY OF THE IEEE

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. IEEE

IEEE COMPUTER SOCIETY

TECHNICAL COMMITTEE APPLICATION

IEEE COMPUTER SOCIETY

INSTRUCTIONS: PLEASE PRINT IN INK OR TYPE (ONE CHARACTER PER BOX. INFORMATION WRITTEN OUTSIDE OF BOXES WILL NOT BE RECORDED). BECAUSE OF PACKAGE DELIVERY SERVICES, STREET ADDRESSES ARE PREFERRED RATHER THAN, OR IN ADDITION TO, POST OFFICE BOX NUMBERS. INTERNATIONAL MEMBERS ARE REQUESTED TO MAKE BEST USE OF AVAILABLE SPACE FOR LONG ADDRESSES.

LAST NAME | FIRST NAME | INITIAL | DR/MR/MRS/MS/MISS/PROF ETC

COMPANY UNIVERSITY AGENCY NAME | DEPARTMENT MAIL STOP/BUILDING/P O BOX APARTMENT ETC

STREET ADDRESS (OR POST OFFICE BOX) | DATE MONTH DAY YEAR

CHECK ONE
- ☐ NEW APPLICATION
- ☐ INFORMATION UPDATE

CITY | STATE | ZIP CODE

COUNTRY | OFFICE PHONE | HOME PHONE (Optional)

ELECTRONIC MAIL NETWORK | ELECTRONIC MAIL ADDRESS (Mailbox)

TELEX NUMBER (Optional) | IEEE MEMBER AFFILIATE NO | I am a Computer Society Member ☐ YES ☐ NO

DEFINITIONS:

TC MEMBER—Actively participates in TC activities, receives newsletters and all TC communications. Active participation means that you do or are willing to do something for the TC such as review papers, help organize workshops, conferences, etc., participate in standards development, help with TC operations, etc. Person can be a member in up to 4 TCs.

TC CORRESPONDENT—Does not participate actively in TC activities, receives newsletters and other TC communications. Person can be a correspondent in up to 4 more TCs.

PLEASE INDICATE YOUR TC ACTIVITY INTERESTS

5 4 3 2 1

HIGH INTEREST ... NO INTEREST

Technical Interests/Specialties: ________

____

____

____

____

____

____

TECHNICAL COMMITTEE CODES (MEMBER M, CORRESPONDENT C)		TC Codes	Conference Organizing Committee	Workshop Organizing Committee	Conference/Workshop Session Chairman	Help with Newsletter	Write Survey/Technical Papers	Review Papers	Standards Development	Educational Activities	Help with TC Operations	Other (Specify):
☐	Computational Medicine (01)	01										
☐	Computer Architecture (02)	02										
☐	Computer Communications (03)	03										
☐	Computer Elements (04)	04										
☐	Computer Graphics (05)	05										
☐	Computer Languages (06)	06										
☐	Computer Packaging (07)	07										
☐	Computers in Education (08)	08										
☐	Computing and the Handicapped (09)	09										
☐	Data Base Engineering (10)	10										
☐	Design Automation (11)	11										
☐	Distributed Processing (12)	12										
☐	Fault-Tolerant Computing (13)	13										
☐	Mass Storage Systems & Technology (14)	14										
☐	Mathematical Foundations of Computing (15)	15										
☐	Microprocessors & Microcomputers (16)	16										
☐	Microprogramming (17)	17										
☐	Multiple-Valued Logic (18)	18										
☐	Oceanic Engineering & Technology (19)	19										
☐	Office Automation (20)	20										
☐	Operating Systems (21)	21										
☐	Optical Processing (22)	22										
☐	Pattern Analysis & Machine Intelligence (23)	23										
☐	Personal Computing (24)	24										
☐	Real Time Systems (25)	25										
☐	Robotics (26)	26										
☐	Security and Privacy (27)	27										
☐	Simulation (28)	28										
☐	Software Engineering (29)	29										
☐	Test Technology (30)	30										
☐	VLSI (31)	31										
☐	Computer and Display Ergonomics (32)	32										
☐	Supercomputing Applications (33)	33										

please return this form to the following address

IEEE COMPUTER SOCIETY
1730 Massachusetts Ave NW
Washington DC 20036-1903

We look forward to having you with us!