IEEE Computer Society

Letters

Special Issue on Workflow and Extended Transaction Systems

Conference Notices

Important Membership Message- Repeated from March Issue.

To Current Members of the Technical Committee on Data Engineering:

There is both good news and bad news in this letter. The good news is that we are well on our way to being able to distribute the Bulletin electronically. This low cost method of distribution should ensure the long term economic survival of the Bulletin. It should permit us to continue to provide the bulletin to all members free of charge. The bad news is that if you do not re-enroll as a member of the Technical Committee, then the June 1993 hardcopy issue of the Bulletin is the last copy of the Bulletin that you will receive.

Our annual revenue, which comes almost entirely from sponsoring the Data Engineering Conference, is not sufficient to cover the costs of printing and distributing four issues of the Bulletin a year. Four issues is, we agree, the minimum publication schedule that is reasonable in order to bring you timely information on technical and professional subjects of interest in data engineering. Long term survival then requires that we limit free distribution to those that can receive the Bulletin electronically. We are working on arranging hardcopy distribution via paid subscription for those that are not reachable electronically or who simply prefer receiving hardcopy. The annual subscription fee for four issues is expected to be in the $10 to $15 range.

Please be aware that failure to enroll means that you will not remain a TC member, and hence that you will no longer receive the Bulletin.

Electronic Enrollment Procedure

Electronic enrollment is the preferred method of becoming a member of the Technical Committee on Data Engineering. To enroll electronically, please use the following procedure:

1. Send e-mail to TCData@crl.dec.com and include in the subject header the word "ENROLL".
2. You will then be sent via an e-mail reply, an electronic membership form that will request:

   Name, IEEE membership no., postal address, e-mail address

   In addition to the above information, you will be asked a number of questions on issues that will affect how the Bulletin's distribution will be handled. These questions will include whether you are interested in subscribing to the hardcopy version of the bulletin. This information will enable us to plan our print runs and determine the parameters of electronic distribution.
3. You should then e-mail the electronic application form to TCData@crl.dec.com, with the word "APPLICATION" in the subject header.

Alternative Enrollment Procedure

In the event that you cannot reach the above electronic mail address, you may enroll in the Technical Committee on Data Engineering by sending postal mail to David Lomet, whose address is given on the inside cover of this issue. Please clearly state that you want to enroll as a member of the TC. A hardcopy application will then be sent to you requesting the same information as described above.

Please note that if you are unreachable via electronic mail, that you will not receive a free copy of the Bulletin. Only hardcopy will be available to you which can be obtained only via paid subscription.

Rakesh Agrawal, David Lomet
TCDE Chair, Editor-in-Chief

Letter from the Editor-in-Chief

The current issue of the Bulletin is the first issue put together by an editor that I appointed, so I take special satisfaction in seeing this issue appear. Mei Hsu, who is active in the workflow field, served as the editor for this issue on "Workflow and Extended Tranaction Systems". The systems described here are destined to become increasingly important as our technical community tackles the task of providing applications that must retain state and satisfy constraints over extended periods. Mei has done a fine job both in terms of selecting authors and sheparding the papers through to final publication. Of importance to me also is the inclusion of a number of papers from industrial organizations. As I have noted before, I believe the research community should know more about industrial efforts.

Editorial changes continue as associate editors appointed when Won Kim was Editor-in-Chief begin to retire. Ahmed Elmagarmid had editorial responsibility for his last issue in December. Yannis Ioannidis was the issue editor of his last issue in March. I want to thank both of them for accepting editorial responsibilities during this transition phase for the Bulletin. Yannis has my special thanks for handling the first electronic edition. Both Ahmed and Yannis have now officially retired as editors.

It is with great pleasure that I appoint and welcome Goetz Graefe of Portland State University as a new Associate Editor of the Bulletin. Goetz is a graduate of the University of Wisconsin and has done outstanding work on query processing, join algorithms and object-oriented databases. I look forward to working with Goetz, who will be handling the December 1994 issue of the Bulletin.

The current issue of the Bulletin is the second that is entirely in electronic form. While it is clear that more "ironing out" of editorial procedures is required, the process has functioned successfully. I intend to specify author instructions more completely so as to provide an increasingly uniform style to the appearance of Bulletin articles. Additional difficulties continue to be encountered in handling postscript figures. Postscript from a diversity of sources is much less standard than one might hope. We continue to grapple with this problem. For this issue, Mei Hsu reworked several of the figures so that I could view and print them, greatly facilitating publication of the issue.

Progress on electronic distribution has not been as rapid as I would have liked. Trial electronic distribution will begin soon. It is not yet clear whether full electronic distribution of the September issue will occur as was originally intended. If not, hardcopy distribution will continue, **but** only to members that have re-enrolled following the enrollment procedure described in the preceding membership message. **So, once again, I urge you to re-enroll in the Technical Committee so that we can keep sending you the Data Engineering Bulletin, with its timely articles capturing the state of our field. If you do not re-enroll, this is your last issue.**

David Lomet
Editor-in-Chief

Letter from the Special Issue Editor

"After several decades of data processing, we have learned that we have not won the battle of modeling and automating complex enterprises," assert Medina-Mora, Wong, and Flores, in their article on ActionWorkflow in this Special Issue of Data Engineering. Their article, together with that by McCarthy and Sarin on Xerox's InConcert system, and that by Hsu, Obermarck and Vuurboom on business process modeling and execution, provide a perspective on where the industry stands on its offerings in workflow automation.

Workflow systems have their origin in the Office Automation effort in the 1970s, and also in the imaging-based systems in the 1980s, both of which were aimed at automating office paper processing. Today, the ambition of workflow systems is to elevate the abstractions of Information Technology (IT) to a level such that these abstractions can be directly understood and manipulated by business users to optimize their business processes, and to achieve competitive advantages.

Research in relaxed or extended transaction models, on the other hand, has been motivated by a need to access heterogeneous information sources and applications in a business transaction. The goal has been to preserve integrity properties of such transactions without violating autonomy of the information sources. The properties to be preserved by such transactions are often expressed as *constraints* among the sub-units of work in concurrent or related transactions. The ACID property of the basic transaction model has provided initial inspiration for modeling these constraints, and transaction managers (which enforce the ACID property) are extended to provide for enforcement of these newly evolving constraints.

The article by Weikum, Deacon, Schaad and Schek provides a lucid characterization of the "semantic serializability" property, a correctness criterion for concurrent execution of multi-step transactions. Rostagi, Mehrotra, Korth and Silberschatz proposes to "transcend" serializability, directly using consistency constraints specified by application administrators. Barga and Pu discusses "epsilon serializablity" for relaxed transaction management.

The ACTA (meta)model described by Chrysanthis and Ramamirtham abstracts the life cycle of a transaction as a sequence of "significant" events, and its properties as dependencies among the events. Klein and Upton has used a specification-based methodology in their implementation of an "extensible" transaction manager, where the coordination protocols are generated from dependencies. Georgakopoulos et al of GTE Lab introduce a facility enabling classes of extended transactions to be specified in an ACTA-like meta model, and managed by a "programmable transaction manager".

The next four papers have added aspects of "context data", or "data flow", to their models. Bukhres, Elmagarmid and Kuhn present two workflow application languages that embody the Flex Transaction Model. Tomlinson et al from MCC give an introduction on how Carnot generates task graph (workflow) scripts and enforces inter-task dependencies. Sheth and Rusinkiewicz offer their perspectives on correctness criteria of workflow, and Dayal and Shan articulate their plan in adding operation flow automation to the Pegasus project.

This collection of 13 articles bring forward a wide range of views on extended transactions and workflow systems. It also demonstrates the existence of different levels of concerns. Research in extended transactions, by generalizing the basic ACID transaction model, provides many promising ideas on how *coordination requirements* on business processes may be specified and enforced. Actual deployment of workflow application systems will be needed for us to know how these ideas can be effectively used to enhance the state of the art in business process modeling and workflow automation.

I wish to thank all the authors for their generous support (and their patience with the tight 4-page limit). Special thanks also go to my colleague Ron Obermarck for his help and encouragement.

Meichun Hsu
Activity Management Group, Digital Equipment Corp., Palo Alto, CA

Open Nested Transactions in Federated Database Systems

Gerhard Weikum *Andrew Deacon* *Werner Schaad* *Hans Schek*
Department of Computer Science
ETH Zurich
CH-8092 Zurich, Switzerland
E-mail: weikum@inf.ethz.ch

1 Models of Federated Transaction Management

1.1 State of the Art

The increasing demand for interoperability of existing information systems requires support for global transactions in federations of largely autonomous databases. In the past few years, substantial research has been carried out on this problem of federated transaction management. The following general model of a federated database system has evolved from this research [4]. A federated database system is composed of a number of pre-existing local databases (LDBs), which are managed by the same or by different database systems. An LDB is a collection of data and pre-existing applications. The applications run local transactions (LTs) that access only the LDB. The federated system serves to support new applications that span multiple LDBs. These applications run global transactions (GTs) that consist of multiple global subtransactions (GSTs) each of which accesses exactly one LDB.

There are two approaches to federated transaction management. The first approach is to rely on some common properties of the underlying LDBs. It has been shown that global serializability and atomicity, i.e., ACID guarantees for both GTs and LTs, are self-guaranteed if all LDBs ensure commit-order serializability [5, 16] and are able and willing to participate in a global two-phase commit. Note, however, that some of today's widely used DBMSs do not necessarily ensure commit-order serializability, since they make use of multiversion concurrency control protocols. The second approach is to build an additional layer on top of the LDBs, to ensure global serializability and atomicity. The goal of this approach is to improve the degree of local execution autonomy. Unfortunately, virtually all of the previously proposed approaches in this category incur severe performance problems. Our approach belongs to this category, but it aims to improve performance by exploiting application semantics.

1.2 Our Approach

We are investigating an approach to federated transaction management based on 1) exploiting semantics of high-level operations explicitly exported by the LDBs, and 2) embedding invocations of these operations into open nested transactions within a requestor-server architecture. The approach adopted is to restrict the way local systems cooperate with each other in a loosely coupled federation. We assume a federated system model that is based on a high-level requestor-server model [13]. For autonomy reasons, local systems may often restrict the operations that global transactions are allowed to invoke. For example, no airline, bank, or rental car agency allows other corporations to access and manipulate its data by means of SQL statements (not even with restricted access permissions and/or

views). Rather an LDB "exports" a well-defined set of high-level operations that may be invoked by global transactions. This notion of exported high-level operations corresponds to the "steps" of nested sagas [10] and ConTracts [21] (cf. also [6]). Many real-life examples of multidatabase interoperability seem to be based on such a high-level requestor-server model [1, 11, 12, 20]. These include, for example, networks of travel agencies, the international interbank clearing system SWIFT, etc.

We are investigating both the foundations and implementation issues of dealing with GTs and their co-existence with LTs within such a framework. Our approach is based on open nested transactions [13, 24], which allows us to ensure semantics-based global serializability and atomicity, and, if desired, to relax these properties in a controlled manner [24].

The rest of this synopsis discusses our semantics-based correctness model and gives a brief overview of a prototype that is being built within the COSMOS project at ETH Zurich. More details on the foundations of our approach and some background material can be found in [7, 18, 19, 22, 24]; more details on the implementation issues can be found in [15, 17, 22, 23].

2 Semantics-based Global Serializability

We assume that a compatibility specification for the exported operations of an LDB, usually in the form of a table, is provided by the administration staff of the LDB. Compatibility of two operations means that the ordering of the two operations in a schedule is insignificant from an application point of view. The compatibility specification could be based on state-independent commutativity or on return-value commutativity. A compatibility specification may even go one step further by explicitly declaring two non-commutative operations as compatible if the different effects of the two possible execution orders are considered negligible from an application point of view (e.g., a pair of Deposit and Withdraw operations on accounts without overdraft protection but with a penalty of say $ 10 which is charged when a Withdraw operation results in a negative balance).

We assume that GTs invoke only exported operations. An invoked high-level operation is dynamically expanded into a sequence of native DML statements of the underlying local DBMS (e.g., SQL statements). LTs may also invoke the exported high-level operations, or may directly execute native DML statements. Both LTs and GTs can be considered trees of operation invocations, with edges representing the caller-callee relationship between transactions and operations at different levels. For GTs, the children of the transaction root can also be viewed as subtransactions; so we are actually using a nested transaction model. Subtransactions may be committed (i.e., their effects are made visible to other transactions) independently of the commit of the transaction root; therefore, this model is called the *open nested transaction* model [13, 3, 24].

In this model, a schedule is a set of transaction trees together with an execution order $<$ of the invoked operations (i.e., the nodes of the trees). A schedule is called semantically serializable if it is equivalent to a serial execution of the transaction roots in the following sense [2]: a serial execution can be stepwise constructed from the original execution by

- *Rule 1 (compatibility of operations):* exchanging the order of two $<$-ordered and (with respect to $<$) adjacent subtrees the roots of which are compatible operation invocations and belong to different transaction trees, and
- *Rule 2 (reduction of isolated subtrees):* reducing an isolated subtree to its root, where a subtree is called isolated if all its descendants are serial and the entire subtree is $<$-ordered with respect to all other operations (i.e., not interleaved with other subtrees).

As an example, consider the schedule of Figure 1. The executions of native DML statements are modeled as (combined) Read/Write operations here; the execution order is from left to right. The

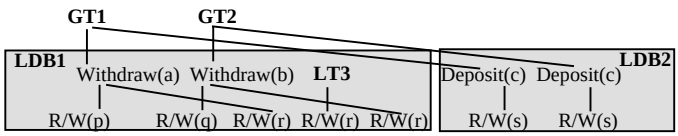


Figure 1: Semantically serializable example schedule

schedule is semantically serializable; it is equivalent to the serial execution GT1 < LT3 < GT2, which can be proven by applying the two rules and assuming that Deposit operations on the same account are compatible.

Our model of federated transactions subsumes sagas [9] as a special case. Assume that there were no LTs and that all exported high-level operations of an LDB are compatible with each other. In this case, GTs correspond to sagas, and local serializability alone is sufficient to guarantee correctness.

3 Semantics-based Global Atomicity

GTs are aborted by invoking additional compensating operations. We assume that an LDB exports a compensating operation for each exported high-level operation, and that there are well-defined rules for deriving the actual input parameters of the compensating operation from the actual input parameters and the return values of the corresponding "forward" operation.

Compensation achieves "semantic atomicity" in the following sense. Within a schedule, the operation invocation f is compensated by the operation invocation f^{-1} if the return values of all subsequent operation invocations are the same as if neither f nor f^{-1} had ever executed. Similar to relaxing commutativity towards compatibility, this definition of a compensating operation may be relaxed as follows (cf. also [14]). A subsequent operation invocation may have different return values if the observable differences are acceptable from an application point of view. For example, the cancelation of an order need not undo the possible restocking of inventory that was triggered by the order.

To reason about the interference of compensating operations and forward operations in concurrent executions, our model is extended as follows. Schedules are augmented by making all recovery-related operations explicit operations. Compensating operations, including their children at the DML level, and the additional undo operations of the LTs are added to the corresponding transaction trees [3, 19, 22]. In addition to the compatibility and reduction rules, we introduce the following rule:

- *Rule 3 (compensation of operations):* If the compensating operation f^{-1} of a forward operation f immediately follows f in a schedule, and if both f and f^{-1} are isolated, then both f and f^{-1} can be discarded from their transaction tree.

Using the three rules, we can reason about the equivalence of schedules with compensating operations. The correctness criterion requires that a schedule can be stepwise transformed into a serial execution of the committed transaction roots by applying the compatibility rule, the reduction rule, and the compensation rule.

4 Prototype Architecture

The described semantics-based correctness model can be cast into a locking protocol, which is driven by an application-defined compatibility table for the exported high-level operations. This protocol follows the implementation techniques for multilevel transactions [22, 23]; however, an additional problem arises in the presence of LTs. Since LTs may invoke native DML statements directly, LTs may interfere

incorrectly with GTs. The approach used to solve this problem is to treat GTs as open nested transactions only with respect to other GTs, and to treat them as closed nested transactions with respect to LTs (i.e., locks acquired by a subtransaction are retained by the parent) [15, 18].

A federated transaction management prototype is being built that incorporates several commercial DBMSs [17]. Messages at the native DML interface between the LDB server and its clients are intercepted, and fed into an additional layer of DML locking and logging on top of the LDB. On top of the DML-level transaction management, semantic locking and high-level logging for possible compensation are implemented in a distributed manner. Note that our approach does not require any changes to existing application programs.

References

[1] Ansari, M., Ness, L., Rusinkiewicz, M., Sheth, A., Using Flexible Transactions to Support Multi-system Telecommunication Applications, VLDB Conference, 1992

[2] Beeri, C., Bernstein, P.A., Goodman, N., A Model for Concurrency in Nested Transaction Systems, Journal of the ACM, 36(1989),2

[3] Beeri, C., Schek, H.-J., Weikum, G., Multilevel Transaction Management: Theoretical Art or Practical Need?, International Conference on Extending Database Technology, Venice, 1988, Springer, LNCS 303

[4] Breitbart, Y., Garcia-Molina, H., Silberschatz, A., Overview of Multidatabase Transaction Management, VLDB Journal, 1(1992),2

[5] Breitbart, Y., Georgakopoulos, D., Rusinkiewicz, M., Silberschatz, A., On Rigorous Transaction Scheduling, IEEE Trans. on Softw. Eng., 17(1991),9

[6] Dayal, U., Hsu, M., Ladin, R., A Transactional Model for Long-Running Activities, VLDB Conference, 1991

[7] Deacon, A., Schek, H.-J., Weikum, G., Semantics-based Multilevel Transaction Management in Federated Systems, Technical Report, Dept. of Computer Science, ETH Zurich, 1993

[8] A.K. Elmagarmid (ed.), Database Transaction Models for Advanced Applications, Morgan Kaufmann, 1992

[9] Garcia-Molina, H., Salem, K., Sagas, ACM SIGMOD Conference, 1987

[10] Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., Salem, K., Modeling Long-Running Activities as Nested Sagas, IEEE Data Engineering Bulletin, 14(1991),1

[11] Gray, J.N., An Approach to Decentralized Computer Systems, IEEE Trans. on Softw. Eng., 12(1986),6

[12] Gray, J.N., Anderton, M., Distributed Computer Systems: Four Case Studies, Proceedings of the IEEE, 75(1987),5

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

[14] Levy, E., Korth, H.F., Silberschatz, A., A Theory of Relaxed Atomicity, ACM PODC Conference, 1991

[15] Muth, P., Rakow, T.C., Weikum, G., Brössler, P., Hasse, C., Semantic Concurrency Control in Object-Oriented Database Systems, IEEE Data Engineering Conference, 1993

[16] Raz, Y., The Principle of Commitment Ordering, VLDB Conference, 1992

[17] Schaad, W., Schek, H.-J., Weikum, G., Architecture and Implementation of a Federated Transaction Management Prototype, Technical Report, Dept. of Computer Science, ETH Zurich, 1993

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

[19] Schek, H.-J., Weikum, G., Ye, H., Towards a Unified Theory of Concurrency Control and Recovery, ACM PODS Conference, 1993

[20] Veijalainen, J., Eliassen, F., Holtkamp, B., The S-Transaction Model, in [8]

[21] Waechter, H., Reuter, A., The ConTract Model, in [8]

[22] Weikum, G., Principles and Realization Strategies of Multilevel Transaction Management, ACM Trans. on Database Systems, 16(1991),1

[23] Weikum, G., Hasse, C., Multi-Level Transaction Mangement for Complex Objects: Implementation, Performance, Parallelism, to appear in: VLDB Journal, 1993

[24] Weikum, G., Schek, H.-J., Concepts and Applications of Multilevel Transactions and Open Nested Transactions, in [8]

Transcending the Serializability Requirement

Rajeev Rastogi[*] *Sharad Mehrotra*[*] *Henry F. Korth*[†] *Abraham Silberschatz*[‡]

1 Introduction

The concept of serializability as the notion of correctness for certain "new" database applications or in certain distributed database environments has been recently shown to have unacceptable practical consequences. For example, in applications such as computer-aided design, where transactions are of a long duration, adopting the serializability correctness criterion may result in long-duration waits imposed by locking protocols [4]. In *multidatabase systems* (MDBS), which is an integration of a number of autonomous local database management systems (DBMSs), ensuring serializability may result in a low degree of concurrency and, depending upon the interface exported by the local DBMSs, may not even be achievable without the loss of *local autonomy*.

In this paper, we propose two complementary approaches for relaxing the serializability requirement. The first approach is based on exploiting the knowledge of the integrity constraints of the system. The fundamental technique used is to partition the database into relatively independent subdatabases, each with its own consistency constraints. We show how the restricted nature of integrity constraints between these subdatabases can be exploited to relax the serializability requirement. The second approach is based on exploiting the semantics of transactions. In this approach, a transaction is broken into a set of subtransactions, with each of which a type is associated. We show how the specification of the set of acceptable/unacceptable interleavings of the subtransactions can be used to relax the serializability requirement while still preserving database consistency.

We further show that both the proposed approaches are applicable to relaxing the serializability requirement in MDBS environments. In an MDBS, there are two types of transactions– *local transactions* that execute at a single DBMS outside the control of the MDBS software, and *global transactions* that execute at multiple local DBMSs under the MDBS's control. Each local DBMS ensures the serializability of the schedules that result from the execution of the transactions at its site. The challenge in transaction management in MDBSs is to integrate the local DBMSs without requiring any modifications to either the software or to the pre-existing applications of the local DBMSs (i.e., without violating the autonomy of the local DBMSs). In such an environment, ensuring global serializability has proven to be difficult task [2], and the proposed schemes that ensure global serializability turn out to permit a low degree of concurrency. In contrast, schemes based on our approach can be used in MDBS environments to permit a high degree of concurrency and thus improved performance without jeopardizing database consistency and without requiring modifications to either the local DBMS software or to the local applications.

[*]Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188.

[†]Matsushita Information Technology Laboratory, Panasonic Technologies Inc., 2 Research Way, Princeton, NJ 08540-6699.

[‡]AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974-0636. On leave from the University of Texas, Austin, where this work was done. Work partially supported by NSF grants IRI-9003341 and IRI-9106450, by the Texas Advanced Technology Program under Grant No. ATP-024, and by grants from the IBM corporation and the H-P corporation.

2 Exploiting Knowledge of Integrity Constraints

Integrity constraints distinguish inconsistent database states from consistent ones. A database state is said to be consistent if it satisfies all the integrity constraints of the system. In this section, we explain how the knowledge of the integrity constraints of a system can be utilized to relax the serializability requirement. Let us consider a database system with the set of data items D and a set of integrity constraints $IC = C_1 \wedge C_2 \wedge \cdots \wedge C_{n-1} \wedge C_n$, where each constraint C_i is defined over the set of data items D_i and $\bigcup_{i=1}^{n} D_i \subseteq D$. Let us assume that $D_i \cap D_j = \emptyset$, for all i, j, $i \neq j$. Thus, effectively, the database D consists of a set of subdatabases $D_1, D_2, \ldots, D_n$, each subdatabase D_i with its own consistency constraints C_i. Furthermore, these subdatabases are independent in the sense that there are no constraints that involve data in more than one subdatabase. In such a system, database consistency can be preserved by ensuring that the resulting schedules are *predicate-wise serializable* (PWSR). A schedule S is said to be a PWSR if its *projection* on the set of data items D_i is serializable, where the projection of S on D_i is the subsequence of S consisting of all the operations that access data items in D_i, $i = 1, 2, \ldots, n$. Note that every serializable schedule is PWSR; however, there are PWSR schedules that are not serializable.

The importance of PWSR as a correctness criterion in CAD/CAM applications was established in [4]. Below, we establish its importance to MDBS environments. However, before we do so, we first note that PWSR schedules preserve database consistency only under certain appropriate restrictions on the system [8]. For example, PWSR schedules can be shown to preserve database consistency if each transaction program has *fixed-structure*. A transaction program is said to have a fixed-structure if its execution from any database state results in the same sequence of database operations (i.e., read and write operations). Another orthogonal way to ensure that PWSR schedules preserve database consistency, without requiring transaction programs to have fixed-structure, is by ensuring that the schedule S besides being PWSR is also *cascadeless* [1]. A schedule is said to be cascadeless if abortion of any transaction in S does not result in abortion of any other transaction in S. We next establish the applicability of PWSR to MDBSs.

Consider an MDBS consisting of local DBMSs located at sites $s_1, s_2, \ldots, s_n$. Let the set of data items at site s_i be denoted by DB_i. In an MDBS the set of integrity constraints can be viewed as $IC = C_1 \wedge C_2 \wedge \cdots \wedge C_n \wedge C_G$, where each constraint C_i (referred to as local constraints) is defined over the set of data items $D_i \subseteq DB_i$ and C_G (referred to as a global constraint) is defined over data residing at multiple different local DBMSs. Let D_G be the set of data items over which C_G is defined (data items in D_G may be located over multiple local DBMSs). Note that, by definition, $D_i \cap D_j = \emptyset$, $i \neq j$. Let us assume that $D_i \cap D_G = \emptyset$, $i = 1, 2, \ldots, n$. Further, since local transactions execute only at a single local DBMS, if they were to update data items in D_G they may violate the constraint C_G. We thus assume that local transactions do not update data items in D_G.

Let us first consider the case in which an MDBS does not contain any global constraints; that is, $D_G = \emptyset$. In this case, since each local DBMS ensures serializability of its local schedule, the global schedule is PWSR. Thus, database consistency can be preserved without any global coordination (and thus fully preserving autonomy) if either transaction programs are fixed-structure or if local DBMSs besides ensuring serializability also ensure that local schedules are cascadeless.

Let us now consider the case where $D_G \neq \emptyset$ (i.e., the MDBS contains global constraints). If local transactions do not read data items in D_G, then PWSR can be ensured by ensuring that the global schedule is *two-level serializable* (2LSR) [5]. A global schedule S is said to be 2LSR if:

- each of the local schedules is serializable, and
- the restriction of S to operations belonging to only global transactions is serializable (i.e., if local transaction operations are deleted from S, then the resulting schedule is serializable).

If, however, local transactions do access data items in D_G, then 2LSR schedules may not be PWSR. In such a case, we can force 2LSR schedules to be PWSR by certain external mechanisms (for example, by forcing any two global transactions that access data items in D_G to conflict directly.). In this case, however, it can be shown that 2LSR schedules can by themselves preserve database consistency if global transaction programs do not have *data dependencies* [5]. A transaction program is said to have no data dependencies if its execution at a local DBMS is independent of its execution at other local DBMSs. In [5, 8], we have identified additional restrictions on transactions and integrity constraints under which 2LSR and PWSR schedules preserve database consistency.

3 Exploiting Transaction Semantics

In this section, we develop an approach that exploits semantics of transactions to relax the serializability requirement. Let us consider a database system in which each transaction consists of a number of subtransactions with each of which a type is associated. Let us assume that the application administrator *a priori* specifies the various subtransaction types along with the set of interleavings of the subtransactions that does not result in a loss of database consistency[1]. A transaction manager utilizes this specification to permit only acceptable, and prevent unacceptable interleavings of the transactions. To make our idea more concrete let us consider a simple example.

Suppose that a banking database supports three different types of subtransactions: *balance* – that returns the balance in an account; *debit* – that debits the balance in an account by a specified amount; *credit* – that credits an account by a specific amount. Consider two transactions in this environment: *transfer* – that transfers money from one account to another and consists of a *debit* subtransaction followed a *credit* subtransaction, and *approx_audit* – that returns an approximate and conservative sum of the balances in various different accounts and consists of one or more *balance* subtransactions. Let the application semantics be such that an *approx_audit* transaction must not overestimate the total money it sees in the various accounts being audited; it can, however, underestimate the total money in the various accounts, in case there are concurrently executing *transfer* transactions that transfer money between the accounts being audited.

In the above example, an interleaving of multiple *transfer* transactions in between the subtransactions of an *approx_audit* transaction is unacceptable since it could potentially result in the *approx_audit* transaction returning the total which is an overestimate of the sum of money in the various accounts being audited. On the other hand, most other interleavings of the transactions of type *approx_audit* and *transfer* are quite acceptable and will not result in a loss of database consistency.

To prevent undesirable interleavings of transactions the application administrator specifies the set of undesirable interleavings to the transaction manager. In our approach, undesirable interleavings are specified as *regular expressions* over the alphabet consisting of the different types of the subtransactions in the system. Thus, in the above banking example, the undesirable interleaving can be expressed as the following regular expression *audit* (*debit, credit*)$^+$ *audit*. Protocols that can be used by the transaction manager in order to prevent undesirable interleavings (that are specified as regular expressions over the transaction types) can be found in [7]. Note that our approach to exploiting transaction semantics is simple yet powerful. Depending upon the knowledge of the transaction types and the application semantics, our approach may be used to permit exactly the set of interleavings that do not violate data consistency.

Our approach to exploiting transaction semantics to relax the serializability requirement can be used

[1] We assume that the subtransaction types and the specified interleavings are fixed *a priori* by the application administrator to keep our discussion simple. The approach is equally viable in case application programmers specify the new transaction types and the set of acceptable interleavings dynamically.

in MDBS environments as follows. The global transactions access data at the various local DBMSs by invoking a fixed set of procedures at the local DBMSs. Each such procedure invocation results in a subtransaction of a global transaction. Each global subtransaction has a type associated with it which is the type of the local DBMS procedure whose execution resulted in the subtransaction. The set of (un)acceptable interleavings can be specified as regular expressions over the types of the local procedures as mentioned above. Algorithms that can be used to prevent unacceptable interleavings in an MDBS can be found in [7].

A similar approach to exploiting the transaction semantics, was proposed in [3]. Unlike us, in their scheme for specifying interleavings, transactions are grouped into *compatibility sets*. Steps of transactions whose types belong to a single compatibility set are permitted to interleave freely, while steps of transactions belonging to distinct compatibility sets are not permitted to interleave at all. Our approach to specifying unacceptable interleavings using regular expressions over the types of the subtransactions in the system can be used to specify interleavings among transactions that cannot be specified using compatibility sets. For example, the unacceptable interleaving that we presented earlier in the context of the banking organization example cannot be specified using compatibility sets.

Finally both our approaches to exploiting semantics of integrity constraints and transactions are complementary. In [6], we present a concurrency control scheme that combine the two approaches for preserving database consistency in MDBS environments.

References

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

[2] Y. Breitbart, Garcia-Molina H., and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2), 1992.

[3] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[4] H. F. Korth, W. Kim, and F. Bancilhon. On long duration CAD transactions. *Information Sciences*, 46:73–107, October 1988.

[5] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *Proceedings of the First International Conference on Parallel and Distibuted Information Systems, Miami Beach, Florida*, 1991.

[6] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Relaxing serializability in multidatabase systems. In *Proceedings of the Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, Mission Palms, Arizona*, February 1992.

[7] R. Rastogi, H. F. Korth, and A. Silberschatz. Using semantics of operations to address waits in database systems. Technical Report TR-92-45, Department of Computer Science, University of Texas at Austin, 1992.

[8] R. Rastogi, S. Mehrotra, Y. Breitbart, H. F. Korth, and A. Silberschatz. On correctness of non-serializable executions. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Washington D.C.*, 1993.

Accessing Imprecise Data: An Approach Based on Intervals

Roger S. Barga *Calton Pu*
Dept. of Computer Science and Engineering
Oregon Graduate Institute
{barga,calton}@cse.ogi.edu

Abstract

In many real world applications (even in banking), imprecise data is a matter of fact. However, classic database management systems provide little if any help in the management of imprecise data. We are applying methods from interval arithmetic, epsilon serializability, and other related areas to help the application designers in the management of naturally imprecise data. Our approach includes operators on imprecise data that give bounds on the result imprecision and algorithms that constrain the imprecision propagation in the database.

1 Introduction

Traditional database management systems provide support only for precise data, though in the physical world data is often imprecise. An important class of examples is the scientific data such as incomplete recording of data, instrument noise, measurement error, computational model imprecision, and data aggregation of one kind or another. Another example is the "fuzzy" data managed by Epsilon Serializability algorithms [PL91, DP93]. In these cases, the application designer is left with the unpleasant decision whether to ignore the imprecise information and store some approximation to the precise value, or to manage the imprecision by hand. Our objective is to provide database management systems that support both precise and imprecise data directly.

In many situations where precise data is not available, much more useful information on the imprecision other than 'value unknown' or 'predicate is possibly true' is available. Different approaches to the problem of managing imprecise information have been proposed. These range from a consistent way of handling incomplete information [Lip79] to recent work on fuzzy data models [SMF90] and probabilistic data models [BGMP92]. One of the most common and useful forms of information available to an application designer is the ability to bound the amount of imprecision or error in the value of a data item. We think that it would prove valuable if the database management system was able to represent and store this type of information on imprecision, as well as provide operators to manipulate the imprecision bounds in a consistent and useful manner. In our examples, the most natural way to represent imprecise data is through an *interval*. For simplicity of presentation we are concerned with numerical data only in this paper.

The immediate problems we wish to address here are how to represent imprecise information (particularly bounded imprecision), the development of a data model for imprecise information, and how to manipulate imprecise data in a consistent manner, particularly when a database state is being transformed. Our basic strategy for dealing with bounded imprecision is to permit each component of a tuple to be an interval of values drawn from the corresponding database domain. We are working on an algebra for interval relations as an extension of relational algebra, using methods from interval arithmetic to manipulate the imprecision bounds.

2 Interval Relational Data Model

The interval relational data model is defined as an enhanced relational database that allows imprecise attribute values and imprecise queries; both are expressed in terms of intervals. As in classical relational database theory, an interval relational database is defined as a set of relations where each relation is a set of tuples. The key departure of the interval relational data model is that tuple components are not confined to single elements drawn from the underlying domain; instead, as an attribute value, an interval is used to represent imprecise data. An example is presented in Figure 1. An interval database

Time	Temperature	Wind Speed	Humidity
1230	[23...25]	[7...10]	[0.10...0.16]
1400	[28...42]	[13...32]	[0.22...0.34]
1530	[44...48]	[9...14]	[0.48...0.62]

SELECT$_{Temperature \subset [20...45]}$ WITH PRECISION 0.90

Figure 1: Interval relation and query.

consists of relations, where a relation $R(t_1, \ldots, t_n)$ is a subset of the Cartesian product $I_1 \times I_2 \times \cdots \times I_n$ of interval domains I_i; each I_i is the Cartesian product $D_i \times D_i$ constructed from an attribute domain D_i $(1 \leq i \leq n)$. A pair $(d, d') \in I_i$ represents the interval $[d, d']$, with an interval width of zero $(d = d')$ representing a precise value. In the interval relational data model, key attributes require precise values.

It is the ability to use an interval of values from a domain as a component in a tuple that provides us with the basic structure for storing imprecise information. An interval representation also provides an intuitively appealing scheme for understanding imprecise information. The width of the interval of a tuple component is a measure of the imprecision of the stored information: the larger the interval the more imprecise the information. The intuition is that for the Ith component of a tuple, an interval containing the entire domain D_i corresponds to information of the worst precision, while an interval of width zero in D_i represents information of the highest precision[1].

With imprecise values specified using intervals, a measure of relative imprecision can be derived. We give just one example here, although many other useful alternative definitions exist.

Definition 1 (Relative Imprecision): The relative imprecision of an interval data element is defined from a maximum imprecision bound Δ_{max},where $\Delta_{max} = max(\mid val(Attr)$ - $low(Attr) \mid, \mid val(Attr)$ - $High(Attr) \mid)$, by $\varepsilon = \frac{\Delta_{max}}{|val(Attr)|}$.

where $val(Attr)$ is the value of the data element stored in the database, without the imprecision bounds.

As a consequence of being able to quantify the relative imprecision of the values assigned to data elements, a tuple may be selected from an interval relation based on the precision of the data it contains. This affords the application designer the ability to control the quality of data used in further experiments and decision making. In addition, bounds on the imprecision of data values is available to bound imprecision in results derived from simulation models or aggregate functions which use this data as input.

3 Interval Relational Algebra

An interval relational algebra is proposed as an enhancement of the relational algebra. The fundamental interval relational algebra operators are UNION, DIFFERENCE, CARTESIAN PRODUCT, PROJECT, and SELECT. These operators are the interval counterparts of the primitive operations in the classical

[1] The problem of missing information is outside the scope of this paper. However, it is conceivable that one could represent the fact that an attribute is *"missing but applicable"* using an interval that contains the entire domain D_i.

relational algebra, and are intended to directly extend the corresponding precise classical operations to the 'non-precise' attributes of the interval relation. We limit our discussion here to the SELECT operation. (Other interval relational algebra operators are discussed in [BP93].)

The syntax of the SELECT operation at level of precision P applied to an interval relation I with selection condition C is: $\text{SELECT}_C(I)$ WITH PRECISION P. The selection condition C is a Boolean expression specified on the attributes of interval relation I. The Boolean expression is made up of a number of clauses of the form *Arg op Arg*, where *Arg* is either a constant interval value or the name of an attribute in an interval relation, and *op* is one of the comparison operators $\{ \in, =, <, \leq, \geq, >, \neq, \supset, \subset \}$. Clauses can be arbitrarily connected by the logical connectives AND, OR, and NOT to form a general selection condition.

Many applications often require query capabilities involving aggregate and statistical functions. The interval representation allows for a natural definition of a full range of aggregate functions over imprecise data: *max, min, mean, count, avg, sum*, and *product*, based on interval arithmetic. We discuss this in detail in [BP93].

4 Bounding the Imprecision on Data

When a data element acquires imprecision due either to external directive, or from the result of a transaction (e.g., UPDATE operations), importing and exporting imprecise values, we would like the DBMS to update and maintain bounds on the amount of imprecision introduced for each data item. Using our interval representation, the amount of imprecision imported by a transaction can be recorded in the database as the interval bounds of an attribute, or the bounds of an existing interval value can be updated following the rules of interval arithmetic.

Classic transaction models do not include inconsistencies such as data imprecision, since a transaction is defined as a program that transforms a consistent database state into another consistent state. We have introduced the concept of *epsilon serializability* [PL91] (ESR) as a family of correctness criteria that bound the amount of inconsistency introduced into transactions. Efficient divergence control algorithms [PHK+93] guarantee epsilon serializability at a cost close to that of classic concurrency control algorithms, from which the divergence control algorithms were derived. In addition to transaction control, consistency restoration algorithms [DP93] can reduce the inconsistency introduced into the database, given some information on the correction of data values.

Divergence control and consistency restoration both support the interval notation of data imprecision. Although originally developed to extend classic (commercial) transaction processing, these algorithms are being adapted for bounding the imprecision on data in general, including scientific data and in particular, the interval relational data model sketched in the previous sections. To achieve this, we specify the amount of imprecision allowed on each data object, denoted as data-ϵ-spec, in analogy to the specification on the amount of allowed transaction inconsistency, denoted as trans-ϵ-spec, in epsilon transactions (ETs). Each data item may contain some imprecision (stored with the interval value and managed by the Interval Relational DBMS), limited by its data-ϵ-spec.

When ETs access imprecise data in an interval relational DBMS, the data imprecision is accumulated by the ET. If the imprecision is tolerable (compared to trans-ϵ-spec) then the ET commits; The query algebra of the interval relational DBMS can be used to later retrieve the imprecise data values committed by transactions relaxed with ESR. Otherwise, the ET may wait, abort, or trigger a consistency restoration method. ESR algorithms are then used to constrain the imprecision propagation.

As an example, consider the collection and processing of environmental data from a number of remote sensing devices. Each device introduces instrument noise and measurement imprecision, which can be quantified and used to bound the precise values for the collected data. Differences in instrument type, quality, and placement also effect the precision of the data and result in varying width interval bounds. Using an ESR-based transaction processing system, the bounded amount of imprecision may be recorded into an update transaction and stored in the interval relational DBMS. Additional processing of the imprecise data values, such as aggregate operations to compute the average temperature or total

precipitation over the monitored area, using only the interval relational algebra operators ensures that interval bounds will be properly maintained. The precision of the data retrieved from the collection can be specified by providing a precision requirement along with a query. ESR divergence control algorithms will maintain the amount of imprecision in the query result within this limit.

5 Closing Remarks

In this paper, we have proposed the concept of the interval relation to capture the natural imprecision in much of real world data. The interval relational data model and algebra have both an intuitive appeal and a simple representation. We are developing interval analogs to the fundamental relational algebra operations and new operators that have no counterpart in classical relational systems. We regard *imprecision* and *uncertainty* as two complementary aspects of imperfect information we have of the physical world. Imprecision refers to the value of a data object, while uncertainty refers to the degree of confidence in the value assigned to an attribute. The ability to represent and manage both imprecision and uncertainty in a database has considerable promise for many practical applications such as scientific data management.

Ongoing research on Epsilon Serializability has produced divergence control algorithms [WYP92, PHK+93] and consistency restoration algorithms [DP93] that preserve bounded inconsistency in data. Queries on data with bounded inconsistency can be formulated easily using our interval algebra. We are actively investigating other aspects of managing imprecision as well as methods to manage uncertainty information, and expect to incorporate both into our system in the future.

References

[BGMP92] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, October 1992.

[BP93] R. Barga and C. Pu. Handling inconsistency in scientific data management: An approach based on intervals. Technical Report OGI-CSE-93-005, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[DP93] P. Drew and C. Pu. Asynchronous consistency restoration under epsilon serializability. Technical Report OGI-CSE-93-004, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993. Also available as tech. report HKUST-CS93-002, Department of Computer Science, Hong Kong University of Science and Technology.

[Lip79] W. Lipski. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, September 1979.

[PHK+93] C. Pu, W.W. Hseush, G.E. Kaiser, P. S. Yu, and K.L. Wu. Distributed divergence control algorithms for epsilon serializability. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, Pittsburgh, May 1993.

[PL91] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, May 1991.

[SMF90] S. Shenoi, A. Melton, and L.T. Fan. An equivalence classes model of fuzzy relational databases. *Fuzzy Sets and Systems*, 38:153–170, 1990.

[WYP92] K.L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proceedings of Eighth International Conference on Data Engineering*, pages 506–515, Phoenix, February 1992. IEEE/Computer Society.

Delegation in ACTA to Control Sharing in Extended Transactions †

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Krithi Ramamritham
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

ACTA is a comprehensive transaction framework that facilitates the formal description of properties of extended transaction models. Specifically, using ACTA, one can specify and reason about (1) the effects of transactions on objects and (2) the interactions between transactions. This paper focuses on one of the building blocks of ACTA, namely delegation. A transaction t_i can delegate *to t_j the responsibility for committing or aborting an operation op. Once this delegation occurs, it is as if t_j performed op and* not *t_i. We discuss how the notion of delegation is useful to capture the interactions that take place in extended transactions.*

1 Introduction

ACTA was introduced in [2] to investigate the formal specification, analysis, and synthesis of extended transaction models. Our goal in this paper is to provide a summary of the notion of *delegation*. Unlike in traditional transactions, sometimes extended transactions require the flexibility of not having to atomically commit all the operations invoked by a transaction. ACTA considers the commitment of an operation as a significant event, in addition to the commitment of a transaction. By separating transaction commitment from operation commitment, ACTA allows for a finer treatment of recovery than is allowed with traditional transactions. Thus, it is possible for a transaction to abort and yet for some of its operations to commit. Furthermore, while a transaction is executing, it may selectively abort some of the operations it has performed and yet commit.

Delegation is a powerful concept and is very useful in the controlled commitment of operations. A transaction t_i can *delegate* to t_j the responsibility for committing or aborting an operation *op*. Once this delegation occurs, it is as if t_j performed *op* and *not* t_i and hence t_j has the responsibility to commit or abort *op*. Consider two transactions t_i and t_j where t_i performs op_i on an object *ob* and then delegates op_i to t_j; t_j then performs op_j on *ob* and then commits. This commitment implies that in the committed state, *ob* reflects the changes done by *both* op_i and op_j even if t_i subsequently aborts. Had t_j so desired, it could have aborted op_i unilaterally.

The fact that t_j is able to observe the changes done by t_i in deciding whether to commit or abort the delegated operation op_i exemplifies how delegation broadens the visibility of a delegatee. Thus, through delegation, a transaction can selectively make tentative or partial results as well as hints, such as, coordination information, accessible to other transactions.

†This material is based upon work supported by the National Science Foundation under grants IRI-9109210 and IRI-9210588 and a grant from University of Pittsburgh.

In the rest of this paper, after introducing the necessary underlying concepts, we give a detailed description of delegation.

2 Events, History, and Properties of Histories

During the course of their execution, transactions invoke transaction management primitives, such as, *Begin, Commit, Spawn* and *Abort*. We refer to these as *significant events*. The semantics of a particular transaction model define the significant events of transactions that adhere to that model. We use ϵ_t to denote the significant event ϵ pertaining to t.

Transactions also invoke operations on objects. We refer to these as *object events* and use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t.

The concurrent execution of a set of transactions T is represented by H, the *history* [1] of the significant events and the object events invoked by the transactions in the set T. H also indicates the (partial) order in which these events occur. This partial order is consistent with the order of the events of each individual transaction t in T. The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ precedes event ϵ' in history H. It is false, otherwise. (Thus, $\epsilon \rightarrow \epsilon'$ implies that $\epsilon \in H$ and $\epsilon' \in H$.)

Each transaction t in execution is associated with a $View_t$ which is the subhistory visible to t at any given point in time. In simplified terms, $View_t$ determines the objects and the *state* of objects visible to t. A view is a projection of the history where the projected events satisfy some predicates, typically on the *current history* H_{ct}. Hence, the partial ordering of events in the view is preserved.

The occurrence of an event in a history can be constrained in one of three ways: (1) An event ϵ can be constrained to occur *only after* another event ϵ'; (2) An event ϵ can occur *only if* a condition c is true; and (3) a condition c can *require* the occurrence of an event ϵ.

Correctness requirements imposed on concurrent transactions executing on a database can be expressed in terms of the properties of the resulting histories. Here are two common types of properties. Further examples can be found at the end of this section. A **Commit-Dependency** of t_j on t_i, specified by $(Commit_{t_j} \in H \Rightarrow (Commit_{t_i} \in H \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j})))$, says that if both transactions t_i and t_j commit then the commitment of t_i precedes the commitment of t_j. An **Abort-Dependency** of t_j on t_i, specified by $(Abort_{t_i} \in H \Rightarrow Abort_{t_j} \in H)$, states that if t_i aborts then t_j aborts.

Some of the dependencies between transactions arise from their invocation of conflicting operations. Two operations p and q *conflict* in some object state, denoted by *conflict*(p, q), iff their effects on the state of the object or their return values are not independent of their execution order. For instance, suppose operation q is an observer of an object's state and p is a (pure) modifier of the state, i.e., p does not observe the state. (For a simple example, consider an object on which read and write are the only operations supported. Read is an observer and write is a modifier.) Suppose p precedes q in a history, i.e., operation q observes the effects of p. Then, if failure atomicity is desired, then all the direct and indirect effects of an aborting transaction must be nullified. So, t_q, the transaction invoking q, has to abort if t_p, the transaction invoking p, aborts, i.e., t_q has an *abort-dependency* on t_p. Suppose instead that q is a modifier and p is an observer or a modifier. Then, operation q makes p's effects/observations obsolete. If the invoking transactions must be executed serializably, t_q has to be serialized after t_p.

Let us discuss serialization orderings precisely. Let $\mathcal{C}$ be a binary relation on transactions: $(t_i \ \mathcal{C} \ t_j)$ if t_i must be serialized before t_j. Thus, $(t_i \ \mathcal{C} \ t_j)$ if $\exists ob \ \exists p, q \ (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$. Clearly, for a history to be serializable, the $\mathcal{C}$ relation must be acyclic, i.e., $\nexists t \ (t \ \mathcal{C}^* \ t)$ where $\mathcal{C}^*$ is the transitive closure of $\mathcal{C}$.

Depending on the semantics of a transaction and its relationship to others, not all conflicts need produce abort dependencies or serialization orderings. To capture this, with each transaction t in progress ACTA associates a $conflictset_t$, the conflict set of transaction t, to denote those operations

in the current history against which conflicts have to be considered when t invokes an operation. The conflict set is thus a subset of the operation events in the current history, where the events in the subset satisfy predicates, again on H_{ct}.

Other dependencies between transactions arise from the constraints imposed on the manner in which extended transactions are required to be structured. For instance, in the nested transaction model [5], a (parent) transaction *spawns* (child) transactions. A parent can commit only after its children have committed. That is, a parent has a commit dependency on its children. But, if the parent aborts, *and a child has not yet committed*, the child is aborted. In this case, because of the (italicized) qualification associated with this abort-dependency, we say that the child has a *weak abort-dependency* on its parent. The commit dependency of the parent on the child and a weak abort-dependency of the child on the parent are formed when a parent spawns a child.

3 Delegation

A transaction t_i delegates to t_j the responsibility for committing or aborting an operation p when it invokes the event $Delegate_{t_i}[t_j, p]$. Once this delegation occurs, it is as if t_j performed p and *not* t_i. Hence, as a side effect, the dependencies induced by operations performed on the delegated objects are redirected from the delegator to the delegatee. $Delegate_{t_i}[t_j, ops]$ delegates the set of operations *ops* from t_i to t_j.

Via nested transactions, let us illustrate a simple use of delegation. Inheritance in nested transactions is an instance of delegation. Delegation from a child t_c to its parent t_p occurs when t_c commits. This is captured by the following requirement ($\mathsf{Commit}_{t_c} \in H \Leftrightarrow Delegate_{t_c}[t_p, AccessSet_{t_c}] \in H$) where $AccessSet_t$ contains all the operations t is responsible for. That is, all the operations that a child transaction is responsible for are delegated when it commits.

Given the concept of delegation, it is no longer the case that the transaction invoking an operation is the same as the transaction that is responsible for committing (or aborting) the operation. Specifically, once t_i delegates p to t_j, t_j becomes the *responsible transaction* for p, or simply, t_j is responsible for p. This is denoted by $ResponsibleTr(p)$. Note that $ResponsibleTr(p)$ can change as delegations occur and so given H_{ct}, for each operation p in it, there exists a $ResponsibleTr$. If p was never delegated, this transaction is the same as the one that invoked p. Otherwise, this is the transaction to which p was most recently delegated.

Delegation has the following ramifications which are formally stated in [3]:

- $ResponsibleTr(p_{t_i}[ob])$ is t_i, the event-invoker, unless t_i delegates $p_{t_i}[ob]$ to another transaction, say t_j, at which point $ResponsibleTr(p_{t_i}[ob])$ will become t_j. If subsequently t_j delegates $p_{t_i}[ob]$ to another transaction, say t_k, $ResponsibleTr(p_{t_i}[ob])$ becomes t_k.

- The precondition for the event $Delegate_{t_j}[t_k, p_{t_i}[ob]]$ is that $ResponsibleTr(p_{t_i}[ob])$ is t_j. The postcondition will imply that $ResponsibleTr(p_{t_i}[ob])$ is t_k.

- A precondition for the event $Abort_{t_j}[p_{t_i}[ob]]$ (as well as for $Commit_{t_j}[p_{t_i}[ob]]$) is that $ResponsibleTr(p_{t_i}[ob])$ is t_j.

- Delegation cannot occur in case the delegatee has already committed or aborted, and it has no affect if the delegated operations have already been committed or aborted.

- Since once an operation is delegated it is as though the delegatee performed the operation. Thus, delegation (1) brings the delegated operations into delegatee's view if they were not already, and (2) redirects the dependencies induced by delegated operations from the delegator to the delegatee — dependencies are sort of responsibilities.

Delegation can be used not only in controlling the visibility of objects, but also to specify the recovery properties of a transaction model. For instance, if a subset of the effects of a transaction should not be obliterated when the transaction aborts while at the same time they should not be made permanent, the Abort event associated with the transaction can be defined to delegate these effects to the appropriate transaction. In this way, the effects of the delegated operations performed by the delegator on objects are not lost even if the delegator aborts. Instead, the delegatee has the responsibility for committing or aborting these operations. Similarly, as the example of nested transactions illustrated above, by means of delegation, it is possible for a subset of the effects of committed transactions not to be made permanent. This is the simplest method for structuring non-compensatable components of extended transactions, e.g., open-nested transactions [4].

A transaction can delegate at any point during its execution, not just when it aborts or commits. For instance, in Split Transactions [6], a transaction may *split* into two transactions, a splitting and a split transaction, at any point during its execution. A splitting transaction t_a may delegate to the split transaction t_b some of its operations at the time of the split ($\mathsf{Split}_{t_a}[t_b] \in H \Leftrightarrow Delegate_{t_a}[t_b, DelegateSet] \in H$). Here, it is interesting to note that a split transaction can affect objects in the database by committing and aborting the delegated operations even without invoking any operation on them.

Other transaction models using delegation include Reporting Transactions and Co-Transactions described in [3]. A reporting transaction periodically reports to other transactions by delegating its current results. This supports the construction of data-driven computations, e.g., pipeline-like or star-like computations. Co-transactions behave like *co-routines* in which control is passed from one transaction to the another transaction at the time of the delegation. As in the case of reporting transactions, a transaction t_a delegates its current results, contained in $ReportSet_{t_a}$, to its co-transaction t_b by invoking the Join event ($\mathsf{Join}_{t_a}[t_b] \in H \Leftrightarrow Delegate_{t_a}[t_b, ReportSet_{t_a}] \in H$).

In cooperative environments, transactions cooperate by having intersecting views, by allowing the effects of their operations to be visible without producing conflicts, and by delegating responsibilities to each other. By being able to capture these aspects of transactions, the ACTA framework is applicable to cooperative environments.

References

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

[2] Chrysanthis, P. K. and Ramamritham, K. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, Atlantic City, NJ, May 1990.

[3] Chrysanthis, P. K. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions.* PhD thesis, University of Massachusetts, Amherst, MA, September 1991.

[4] Chrysanthis, P. K. and Ramamritham, K. Synthesis of Extended Transaction Models using ACTA. *Submitted for publication*, April 1992.

[5] Moss, J. E. B. *Nested Transactions: An approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[6] Pu, C., Kaiser, G., and Hutchinson, N. Split-Transactions for Open-Ended Activities. In *Proceedings of the Fourteenth International Conference on VLDB*, pages 26–37, Los Angeles, CA, Sept. 1988.

Constraint Based Transaction Management

Johannes Klein *Francis Upton IV**
Digital Equipment Corporation
Activity Management Group
Palo Alto, CA 94301

Abstract

Transaction management requires integration of coordination protocols, concurrency control protocols and communication protocols. Protocol integration is simplified by specifying coordination protocols as logical constraints. Such a framework provides a foundation for specifying and executing the different transaction models and activity types required for work flow systems. This approach has been used to specify and implement Open DECdtm, a portable transaction management service layered on OSF DCE.

1 Introduction

Transaction management requires the integration of coordination, concurrency control, and communication protocols. Protocol integration is simplified by specifying coordination protocols as logical constraints. This framework provides a foundation for specifying and executing the different transaction models and activity types required for work flow systems. The approach has been used in Open DECdtm to offer portable transaction management services layered on OSF DCE. It supports the application (TX), resource manager (XA), and transactional DCE RPC (TxRPC) interfaces specified by X/Open [5]. Open DECdtm provides interoperability with OSI Transaction Processing (OSI TP) [4] and OpenVMS systems using the DECdtm OpenVMS protocol[3]. Protocols are specified by constraints. This simplifies the development of transactional gateways between different data transfer protocols and allows for support of different transaction models in a heterogeneous environment.

Most transaction management protocols are defined in the context of particular data transfer protocols [1]. No agreement has been reached on how to manage transactions across different data transfer models (i.e. request/response, peer-to-peer, queued). To address those issues and to allow applications use of different data transfer protocols within a transaction, Open DECdtm supports synchronization between transaction state and data transfer state.

Today most transaction management protocols only provide support for flat transactions. Those providing support for nested transactions have difficulty using their extended functionality when coordinating transactions in a heterogeneous environment. The approach taken by Open DECdtm allows support for nested transactions in a manner that interoperates with flat transaction protocols by representing each nested transaction as a flat transaction and providing the necessary coordination locally. This is a first step towards supporting different transaction models. An integration with different concurrency control protocols has been prototyped. Commercial usage is limited due to the fact that most lock manager are not built to be integrated with different concurrency control protocols.

*This paper expresses the views of the authors, not opinions or future product plans of Digital Equipment Corp.

2 Model and Coordination Protocols

Transaction processing systems distinguish between four different components: applications (AP) executing user programs; applications accessing local resource managers (RM) using languages such as SQL; applications accessing remote resource managers through services offered by remote applications; and invocations of remote services supported by communication managers (CM). Transaction managers (TM) ensure that local and remote resource managers agree on the same outcome of a transaction. They must also ensure that all messages sent by participating applications are processed before allowing a transaction to commit. Protocols are specified to define the correct behavior of each component. Internally, Open DECdtm models transaction participants as resources. Each resource is managed by one of the above components. Examples of resources are an AP thread, an RM instance, or a specific branch managed by a CM. Resources change state when processing a transaction. Those state transitions are called events. Protocols define conditions on the occurrence of events. Open DECdtm uses constraints to define these conditions[2]. Messages are exchanged between components to indicate the state of a given event and resolve conditions. Messages indicate that events have become TRUE, have become FALSE, or are promised to become TRUE at some later point. Examples of events include *commit*, indicating the transaction has committed, *request*, indicating the desire to begin transaction termination, and *token*, indicating the right to send data on a given communication link. Constraints are defined using *when-then-enable* statements.

when< *condition* >
    then [enable]< *action* >.

Conditions and actions refer to events. All conditions associated with an event must be satisfied before the event becomes enabled, i.e. the event may only become TRUE once it has been enabled, but being enabled does not imply the event will become true.

Take as an example the specification of a coordination protocol for nested transactions. Each participant in a nested transaction indicates completion of its computation with a *finish* event. Commitment of the results is indicated by a *commit* event. Participants coordinating the agreement protocol indicate start of the first phase by issuing a *request* event. Commitment of the *Superior* becomes dependent on the commitment of the *Subordinate* after the *Subordinate*'s computation has been successfully completed. Commitment of the *Subordinate* is unconditionally dependent on the commitment of the *Superior*.

```
constraint Participant (Superior, Subordinate)
  {
  //start termination when reveiving a request
    when request(Superior) then
    request(Subordinate);
  //commit when requested and superior agreed
  when request(Superior) and commit(Superior)
    then commit(Subordinate);
  //commit when subordinate agreed
  when (commit(Subordinate) or not(finish(Subordinate)) then
    commit(Superior);
  }
```

If at runtime state transitions cause updates to recoverable resources, journaling actions are automatically triggered by analyzing the state of affected constraints. Optimizations such as *read-only, volatile, last-participant* and *one-phase* are automatically derived ensuring optimal behavior of Open DECdtm.

3 Conflict Detection and Conflict Resolution

Frameworks for supporting different transaction models must enable integration of concurrency control protocols with transaction coordination protocols. When supporting concurrency control protocols two important aspects need to be addressed: conflict detection and conflict resolution. In most implementations conflict detection is provided by lock managers. Conflict resolution components interact with transaction managers and conflict detection components to resolve conflicts between different transactions. Conflict resolution policies specify how conflicts have to be resolved, thereby defining the concurrency control protocol for the respective transaction model and application environment. Conflict resolution components analyze the coordination protocols established and other application specific data. Since coordination protocols are represented and maintained as logical constraints conflict resolution components are able to retrieve those kinds of information at runtime.

Conflicts between operations are caused by one of the following reasons:

1. An operation modifies the state of a resource observed by a previous operation (Order).
2. An operation observes a modification caused by a previous operation (Dataflow).

If an event indicates completion of a transaction's computation a hybrid of two phase locking and constraint enforcement may be used as conflict resolution policy. This increases concurrency but still ensures isolation.

1. Delay write operations until all current readers finished their computations.
2. Delay read operations until the current writer finished its computation.

As an example take transaction A whose modification is delayed until transaction B's computation has been finished, *when (finish(B) or not(finish(B))) then write(A)*. Transactions reading recoverable resources can only commit if all modifications seen have been committed and recorded on stable storage. Therefore in most cases transactions only access resources after all updates have been committed. However, if constraints ensure that a transaction's commitment depends on the commitment of all previous update transactions modified resources can be accessed as soon as the modifying transaction finished its computation. To check for a commit-dependency conflict resolution components may issue the following query given transaction A wants to access a resource previously modified by transaction B: *query(when commit(B) then commit(A))*. Besides checking basic ordering and coexistence requirements conflict resolution components may also evaluate other attributes associated with transactions such as whether or not transactions are started by users of the same work group.

4 Communication Management

When dealing with transactional communication, there are two important entities:

1. The transaction branch which represents the state of the remote transaction participant.
2. The connection which represents the state of the communication link to a remote application.

Current proposal for communication manager interfaces, such as XA+, only deal with issues related to branch management. However, it is useful to treat connection and branch management separately because of the different nature of their events and their different lifetimes. Failure of a connection may abort ongoing local branches while causing a recovery of local branches which are promised to *commit* or *committed*. The current state of a connection also determines when commitment can be started or

whether a premature request for commitment should cause the transaction to be aborted or cause a state check.

Branches have the traditional transaction events, e.g. *commit, request.* They live until the transaction termination protocol has completed. Connections have events associated with them such as *com_fail*, indicating the connection has failed, and *token*, indicating that data may be sent on the connection. Some connections are only associated with a single branch[4] others are multiplexed between transaction branches[3].

Polarized control[4] is another important aspect in the relationship between connections and branches that must be considered. In this case branches are only allowed to start commitment if all connections are in the *send* state. The transaction manager has to prevent any attempt by the AP to commit a transaction if any of its associated connections are not in send state. This ensures that all messages sent are processed before commitment starts.

Take as an example polarized control (half-duplex) dialogues (conversations) as defined by OSI TP or IBM's SNA LU 6.2[4]. At any time only one application is allowed to send data on a dialog. An application issuing the request to commit must have the right to send on all its dialogues. This ensures that all data is received before an application commits. The transaction manager should reject any commit request if it is issued prematurely. If this "state check" is not handled by the transaction manager, then transactions would have to be aborted to ensure message integrity.

5 Summary

Open DECdtm offers portable transaction services across OSF and VMS platforms. Open DECdtm is layered on OSF DCE with support for TxRPC, XA, and TX as specified by X/Open. Open DECdtm supports the DECdtm Open VMS for VAX protocol and provides interoperability with OSI TP. Open DECdtm's protocols are specified using constraints. This simplifies the task of interoperating with different communication protocols and provides an extensible infrastructure for new transaction models and work flow systems.

Acknowledgements

The authors would like to thank Dieter Gawlick, Bob Taylor, Wayne Duquaine, Edward Cheng, Mike Depledge, Alberto Lutgardo, Dora Lee, Ron Obermarck and Lucia Springer for their support.

References

[1] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufman, 1992.

[2] Klein, J., *Advanced Rule Drived Transaction Management*, IEEE COMPCON, February 1991.

[3] Laing, W., Johnson, J.E., Landau, R.V., *Transaction Management Support in the VMS Operating System Kernal*, Digital Technical Journal, Winter 1991.

[4] Upton IV, Francis, *OSI Distributed Transaction Processing: An Overview*, International High Performance Transaction Processing Workshop, 1991.

[5] X/Open Company, *CAE Specification, Distributed Transaction Processing: The XA Specification*, 1991.

An Extended Transaction Environment for Workflows in Distributed Object Computing

Dimitrios Georgakopoulos *Mark F. Hornick* *Frank Manola* *Michael L. Brodie*
Sandra Heiler *Farshad Nayeri* *Benjamin Hurwitz*
GTE Laboratories Incorporated
Waltham, MA

1 Introduction

Distributed object computing is evolving as a new paradigm for solving complex distributed systems problems. It is supported by Distributed Object Management Systems (DOMSs) [MHGH+92], which are object-oriented environments in which distributed, autonomous, and heterogeneous systems (such as database and legacy systems) can be integrated, and complex, non-traditional applications can be developed. A DOMS integrates local systems by representing their data and functionality as objects, and allowing client applications to access them without knowing their location, access language, or data representation. It must ensure correct and reliable execution across multiple objects. The scope of a DOMS ranges from small centralized systems to massive world-wide information systems.

There is increasing interest in specifying applications in a DOMS environment as *workflows* involving distributed objects. A workflow is a collection of tasks organized to accomplish some business process. Workflows define the structure of business processes, including the order of task invocation, task synchronization, and data-flow, and specify tasks performed by both humans and computer systems using high-level scripting or rule-based languages.

gtefig1.ps

Figure 1: Concurrent execution of multiple workflows.

For example, Figure 1 depicts three telecommunications workflows that access a combination of distributed resources (databases and a telecommunications switch). A basic problem is to ensure the correctness and reliability of such workflows. This becomes increasingly complex as more and more aspects of workflows are automated because: (1) the number of concurrent tasks accessing shared resources increase sharply; (2) the potential for failures of automated tasks or workflows increases; and (3) less human intervention to control process flows or resolve conflicts is possible.

2 Treating Workflows as Extended Transactions

Typically, workflows managers do not ensure *correctness* or *reliability*. But, without such guarantees, concurrent workflow execution may result in, for example, lost updates, dirty or inconsistent retrievals, or even allocating the same non-sharable facilities to two different customers. *Correctness* is ensured if the executions of concurrent tasks and workflows are interleaved in a way that cannot produce incorrect results. For example, consider two extended transactions, T_1 and T_2, that access shared resources R_1 and R_2. If T_1 accesses R_1, but T_2 executes completely before T_1 accesses R_2, when T_1 commits, the serialization order at R_1 is T_1 before T_2, but the serialization order at R_2 is T_2 before T_1. This globally unserializable schedule may produce incorrect results as noted above.

Reliability requires that both tasks and workflows be *recoverable* in the event of failure, and that a well-defined method exists for *recovery*. Providing *task reliability* requires maintaining information, such as logs, that can be used to determine the state of a failed task. Systems that support transactions automatically provide a single failure state (i.e., abort) for tasks that execute as transactions. Systems that do not support transactions require workflow managers to track the possibly multiple failure states of each task and provide necessary recovery actions. Similarly, ensuring *workflow reliability* requires the ability to determine the state of a failed workflow so that completed tasks can be compensated for, and partially completed tasks can be aborted.

Our approach addresses these correctness and reliability issues by expressing workflows as *extended transactions*. Treating workflows as extended transactions is based on the following:

1. workflows have constituent transactions corresponding to workflow tasks
2. workflows have an execution structure corresponding to the workflow structure, and such structure is defined by an *extended transaction model* (ETM)
3. workflows obey a correctness criterion, as specified by the same ETM as in (2)

Workflows are so diverse in their requirements that no single ETM is sufficient to meet the needs of all workflows. Moreover, different workflows often have incompatible requirements which makes defining a single model impossible. Therefore, our objective is to provide an environment within which we can support various ETMs.

Our approach to implementing ETMs in support of workflows combines rules with transaction management mechanisms (TMMs) such as schedulers. In particular, we implement extended transaction structure using rules, and enforce transaction correctness and reliability using schedulers. In our approach, rule definitions include interactions with schedulers, and schedulers are designed to take into account rule executions and the results of rule actions. This has significant advantages over other approaches that use rules alone to specify and implement workflows. Using rules for the specification of correctness criteria is extremely difficult. Their complexity grows exponentially with the number and complexity of the extended transactions.

Although sufficient for implementing workflows as extended transactions, combining rules and schedulers introduces two problems. First, integrating rule-based specifications and schedulers is complex and brings specification to the level of writing programs. Second, combining rules and TMMs does not constitute a homogeneous specification language. To address these problems, we separate the specification of ETMs from their implementation. We specify extended transactions (the workflows) as dependencies between constituent transactions (their tasks). These dependencies, expressed as *dependency descriptors* in our ETM specification language, are translated into combinations of rule definitions and instructions to schedulers to support a particular ETM. The language used for ETM specification, formally defined in [GH92], is similar to ACTA [CR92]. Unlike ACTA, however, the TSME dependency specification language is minimal and does not require complete histories.

3 Transaction Specification and Management Environment

To support the specification of ETMs and their corresponding implementations, we introduce the concept of a Transaction Specification and Management Environment (TSME) [GH92], which comprises a Transaction Model Specification Facility (TSF) and a corresponding *programmable* TMM, as depicted in Figure 2. The TSF accepts specifications of ETMs expressed in terms of dependencies between the kinds of transactions allowed by each ETM; the programmable TMM supports the implementation of ETMs (to ensure transaction dependencies) and provides an environment in which to execute extended transactions. This separation of model specification from implementation allows ETM designers to reason about and check the correctness of models separate from their implementations.

Figure 2: A TSME Facility.

DOMS application programmers may implement extended transactions that behave according to a TSME-supported model, called *model-dependent* transactions, or they may implement *one-of-a-kind* transactions not associated with any TSME-supported model. One-of-a-kind transactions are useful for prototypical implementation of workflow requirements, and can be generalized to ETMs for reuse by other transactions, if needed. If a class of workflows requires an entirely new ETM, this may be specified directly. For example, if an application requires specific kinds of transactions to cooperate (share intermediate results or interleave data accesses) to increase concurrency, this may be specified as a new ETM.

The TSF analyzes a model's specifications to determine its transaction management requirements. Such requirements can be classified as constraints on the execution structure of complex extended transactions required by an ETM, and the correctness criteria such transactions must satisfy. For example, a structural requirement of nested transactions is that a parent cannot commit unless all its children abort or commit. Examples of correctness criteria include serializability, temporal order, and transaction cooperation. The TSF uses a repository to record TSME-supported models.

Specifications are translated to instructions for the programmable TMM which then assembles and configures a run-time environment according to those instructions, for example, by combining the appropriate TSME-provided schedulers, programs that access local systems, and rule definitions. The run-time environment ensures that model-dependent transactions behave according to their model. The TSME also supports the evolution and integration of existing ETMs by allowing the composition of specifications.

Extended transactions consist of a set of constituent transactions and a set of dependencies between them. *Transaction dependencies* specify correctness criteria or transaction execution structure. *Transaction state dependencies* are conditions on transaction state[1] that define the execution structure of

[1] The state of a transaction may be, for example, *prepared*, *committed*, or *aborted*.

extended transactions. We have identified three kinds of state dependencies: backward, forward, and strong. For example, on transactions T_1 and T_2 a backward-commit-begin dependency requires that "T_2 cannot begin before T_1 commits," a forward-commit-begin dependency requires that "T_2 cannot begin after T_1 has committed," or a strong-commit-abort dependency requires that "T_2 must abort if T_1 commits." *Correctness dependencies* specify which concurrent executions of extended transactions preserve consistency and produce correct results. Dependencies in this category include: serialization order, visibility, cooperation dependencies, and temporal order dependencies [GH92]. For example, "T_1 and T_2 do not conflict on any operations of object o_1," and "T_3 must be temporally ordered after T_1 and T_2" are correctness dependencies.

The programmable TMM uses Event-Condition-Action (ECA) rules [DHL90] triggered by transaction state change events to enforce transaction state dependencies. It uses traditional scheduler technology to ensure correctness dependencies. To enforce correctness dependencies, the programmable TMM uses a hierarchical scheduler comprised of: a global scheduler (GS) at the root of the hierarchy, high-level ("local global") schedulers associated with complex objects, and simple object schedulers at the leaves.

A hierarchical scheduler is required because objects may use different, possibly autonomous and heterogeneous schedulers to perform local synchronization of the operations they support. Each component of a complex object may provide a different scheduler, requiring synchronization at each level of the complex object. DOMS transactions issue operations on objects through the GS, which behaves as a multidatabase scheduler. That is, the GS ensures that all object schedulers in a DOMS provide the same correctness guarantees. For example, the GS may use one of the methods described in [GRS92] to ensure that global transactions have the same serialization order in all object schedulers.

To support multiple correctness criteria from various ETMs, the programmable TMM provides a *programmable scheduler* (PS) for objects that neither provide their own scheduler, nor use a local scheduler. The PS accepts commands to alter, for example, the correctness criteria, visibility, and enforcement options it applies to objects under its control. The PS uses a *generalized conflict table* that defines conflicts between semantic operations and allows the specification of transaction cooperation on specific objects and/or for specific operations. A programmable TMM that incorporates the capabilities above is currently being implemented in our DOMS.

References

[CR92] P. Chrysanthis and K. Ramamritham, "ACTA: The SAGA Continues", *Advanced Transaction Models for New Applications*, A. Elmagarmid (ed.), Morgan-Kaufmann, 1992.

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

[GH92] D. Georgakopoulos and M. Hornick, *An Environment for the Specification and Management of Extended Transactions and Workflows in DOMS*, TR-0218-09-92-165, GTE Laboratories Incorporated, October 1992.

[GRS92] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth, "Using Ticket-based Methods to Enforce the Serializability of Multidatabase Transactions", to appear in *IEEE Transactions on Data and Knowledge Engineering.*

[MHGH+92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie, "Distributed Object Management", *Intl. Journal of ICIS*, 1(1), March 1992.

Implementation of the Flex Transaction Model

Omran Bukhres *Ahmed Elmagarmid* *Eva Kuhn* [†]
Computer Science Department
Purdue University
West Lafayette, IN 47907
{bukhres,ake}@cs.purdue.edu

Abstract

A multidatabase system integrates pre-existing and heterogeneous databases in a distributed environment. A multidatabase transaction is a consistent and reliable execution of an application over a multidatabase system. In this article, we first summarize the characteristics of flexible transactions, then present two multidatabase transaction specification languages for workflow applications, the InterBase Parallel language (IPL) and InterSQL. Both languages provide approaches to the definition and mangement of a unit of work. Design issues, components, and examples of both languages are described. The implementation of IPL and InterSQL is also discussed.

1 Introduction

A multidatabase system (MDBS) is a typical heterogeneous distributed system which integrates pre-existing local database systems (LDBSs) to support applications accessing more than one LDBS. InterBase [BCD+93], a multidatabase prototype at Purdue University, is such a system. The purpose of the InterBase project is to define the design and implementation of a global environment for applications to access multiple systems. InterBase supports flexible interactions among heterogeneous systems by employing specially designed agents, termed Remote Systems Interfaces, to mask the heterogeneities of the local systems; by providing a uniform language, the InterBase Parallel Language (IPL) [CBE93], to specify global transactions; and by supporting the Flex Transaction model [Elm92] to coordinate concurrent global transactions. An MDBS transaction is a consistent and reliable execution of such an application; its subtransactions are executed as local transactions on individual LDBSs. Because of the distinct environmental characteristics of MDBSs, MDBS transactions differ greatly from traditional distributed transactions. The preservation of the ACID (atomicity, consistency, isolation, and durability) properties of traditional transactions becomes problematic in MDBS environments. Furthermore, MDBS transactions are vulnerable to the failure of their subtransactions, because these subtransactions are usually executed on remote systems beyond the control of the MDBS.

1.1 Flex Transaction Model

Many researchers have proposed alternative transaction models for MDBS transactions which can tolerate the failure of individual subtransactions [Elm92]. For example, the Flex Transaction model

[†]University of Technology Vienna, Institute of Computer Languages, Argentinierstraße 8, 1040 Vienna, Austria

[ELLR90] relaxes the atomicity and isolation properties of nested transactions to provide users increased flexibility in specifying their transactions. Consider a transaction that is composed of a set of tasks. For each task, the Flex Transaction model allows the user to specify a set of *functionally equivalent* subtransactions, each of which, when completed, will accomplish the desired task. A Flex Transaction is thus resilient to failures in that it may proceed and commit even if some of its subtransactions fail. The Flex Transaction model also allows the specification of dependencies on the subtransactions of a Flex Transaction; these may take the form of *internal* or *external dependencies*. Internal dependencies define the execution order of subtransactions, while external dependencies define the dependency of a subtransaction execution on events (such as the start/end times for the execution) that do not belong to the transaction. Finally, the Flex Transaction model allows the user to control the isolation granularity of a transaction through the use of compensating subtransactions. All of the above features contribute to the flexibility of the Flex Transaction model, rendering it particularly useful for flexible transaction processing in multidatabase systems. An MDBS transaction specification language should therefore allow users to specify the isolation and atomicity granularity of an MDBS transaction to support advanced applications in MDBSs. The Flex Transaction model has been implemented in the InterBase Parallel Language (IPL) [CBE93] and the Vienna Parallel Logic (VPL) language [KPE91]. The Flex Transaction model has also influenced such implementations as MCC's Carnot project.

2 Workflow Aspects of InterSQL

InterSQL combines the transaction and execution specification abilities of IPL [CBE93] with the unified common query language approach of FSQL used in FBASE [Mul92] while adding high-level support for atomic commitment.

An InterSQL transaction program consists of two parts: a subtransaction definition and an execution section. The latter, which forms the *main program*, specifies the execution plan and data flow of the transaction and can reference previously defined subtransactions. Each subtransaction executes at a single system, and subtransaction arguments are used to pass data to and return results from subtransactions. Figure 1 illustrates a sample InterSQL transaction program.

The main program consists of a list of statements. These statements are either SQL statements that access local data, or statements used to control the execution flow. The statements allowed can be summarized as follows:

< *statement_list* > ::= ϵ | < *statement_list* > < *statement* >

< *statement* > ::= < *standard_sql_statements* >
[[< *library_name* > :] < *system_name* > :]
< *procedure_name* > (< *arg_list* >) ;
| **execute in parallel** < *statement_list* >
| **execute one of** < *statement_list* >

The procedure invocation statement is used to invoke procedures. If a procedure is invoked in the main program, it represents the invocation of a subtransaction. The subtransaction must either be defined in the transaction program, or defined in a system library. To specify a system library subtransaction, the system name of the library is prepended to the subtransaction name. For example, **bank1:withdraw(101, 1000);** represents a withdrawal subtransaction at system "bank1" that withdraws $1000 from account number 101. If a procedure is invoked within a subtransaction it refers to a database method (or function). The database method must have been previously defined at the system that is accessed. A subtransaction cannot invoke another subtransaction. The "execute one of" statement is included to permit alternate possible execution paths to achieve global commitment. For example, a user may want to reserve a room at any one of three hotels, would write the following in the main program:

```
transaction simple_bank_transfer {
  subtransaction withdraw () at bank1 {
  update account set balance = balance - 100 where number = 101; }
  subtransaction deposit () at bank2 {
  update account set balance = balance + 100 where number = 101; }
  Main Program
  withdraw();
  deposit(); }
```

Figure 1: InterSQL Example

```
execute one of { reserve_hotel1(); reserve_hotel2(); reserve_hotel3(); }
```

The "execute one of" statement here instructs the program to execute (in this case commit) only one of subtransactions "reserve_hotel1()", "reserve_hotel2()", or "reserve_hotel3()", and to abort the transaction if none of them can be committed. The "execute in parallel" statement has the same effect as listing subtransaction invocations sequentially, except that the subtransaction may be executed in any order, including in parallel. So all the subtransactions invocations listed in an "execute in parallel" statement must therefore be committed for the transaction to succeed. The "execute one of" and "execute in parallel" statements can be arbitrarily nested. Clearly, any combination of acceptable goals can be specified, although one may not specify an execution plan that would execute the same subtransaction twice.

3 The InterBase Parallel Language

The InterBase Parallel Language (IPL) is the transaction specification language of InterBase. IPL allows users to specify all actions associated with a global transaction, such as the control flow and data flow among subtransactions. Moreover, InterBase will automatically execute subtransactions in parallel when it can do so without violating the specified control flow or data flow constraints. IPL therefore retains the essential characteristics of workflow languages.

The features of IPL are most effectively illustrated through an example. Suppose a user wishes to transfer money from either a checking or savings account to a Visa account. The user will make the final determination of the amount from which to withdraw while the transfer is in progress. We assume that the bank system is a distributed multidatabase system which maintains all checking accounts in a `Sybase` database system on the machine `sonata`, all savings accounts in an `Ingres` database system on the machine `ector`, and all Visa accounts in a `guru` database system on the machine `interbase`. The IPL program for this application is shown in the Appendix, and we will refer to it throughout our explanation of IPL. An IPL program consists of three parts that specify, respectively, definition of data types, definition of subtransactions, and dependency relations among subtransactions.

Definition of Data Types: In IPL programs, a subtransaction is associated with a data type which specifies the data structure of the result if the subtransaction is successfully executed (i.e., reaches its ready-to-commit state). The predefined data types provided are **int**, **real**, **boolean**, and **charString**, and users can define complex types through aggregation. In the example, the subtransactions *checking, savings*, and *visa* all produce outputs of the complex type `account`.

Definition of Subtransactions: This part provides mechanisms for specifying subtransactions within an IPL program. In the example, `checking` takes data of the type `inparams` from the input task `in` as its input parameter. The local software system involved is the `sybase` system, which

runs on the machine `sonata`. Its execution step, optional confirm step, and optional undo step[1] are defined between IPL keyword pairs `beginexec` and `endexec`, `beginconfirm` and `endconfirm`, and `beginundo` and `endundo`, respectively.

Dependency Description: The dependency description provides users with a mechanism for specifying the execution order among the subtransactions of a global transaction. The dependency description supports *functionally equivalent* subtransaction alternatives for the goals of transactions. This flexibility allows an increased tolerance of individual subtransaction failures. In the example, since two subtransaction alternatives (*checking, visa*) and (*savings, visa*) can be deduced, the global transaction can therefore tolerate the failure of either *checking* or *savings*.

The major advantage of using IPL is its semantic power and suitability. Through dependency description, programmers in IPL can specify control flow among subtransactions, which gives IPL the flexibility to support parallel execution and synchronization among subtransactions. As shown in the example, IPL provides a method for specifying data flow within a global transaction. IPL permits the construction of mixed global transactions by allowing the extent of compensatability to be specified in the declarations of subtransactions. Commit and abort operations of subtransactions are deferred until their global transactions commit or abort, if they are defined, and thus support atomic transactions. IPL also allows the specification of transactions that may include subtransactions that access non-database systems, and database systems with complex data models, because statements in the native language can be incorporated into IPL programs.

References

[BCD+93] O. A. Bukhres, J. Chen, W. Du, A. K. Elmagarmid, and R. Pezzoli. InterBase : An Execution Environment for Global Applications over Distributed, Heterogeneous, and Autonomous Software Systems. *IEEE Computer*, August 1993. (to appear).

[CBE93] Jiansan Chen, Omran A. Bukhres, and Ahmed K. Elmagarmid. IPL: A Multidatabase Transaction Specification Language. In *Proc. of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA, May 1993.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 507–581, Brisbane, Australia, August 1990.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[KPE91] E. Kuehn, F. Puntigam, and A. Elmagarmid. An Execution Model for Distributed Database Transactions and Its Implementation in VPL. In *Proc. of Extending Database Technology*, 1991.

[Mul92] James G. Mullen. FBASE: A federated objectbase system. *International Journal of Computer Systems Science and Engineering*, 7(2):91–99, April 1992.

[1] The execution step of a subtransaction is executed first. Its confirm step or undo step, if defined, will be executed when the execution of the IPL program is committed or aborted from its prepare-to-commit state, respectively.

Appendix: Example IPL Program

```
program
  record inparams of  /* the inputs to the program from the customer */
    accNum : charString;          /* account number */
    amount : integer;             /* the amount of money to be transferred */
  endrecord;

  record account of   /* the schema of bank accounts such as savings */
    name,    accNum,       suffix : charString;
    balance, preBalance,  amount : integer;
  endrecord;

  /* input task "in" to obtain the inputs of type inparams from the customer */
  input in : inparams endinput;

  subtrans checking (in) : account use sybase at sonata output
    beginexec  /* execution step, in SQL format */
      begin tran sybasebank;
        update bank set amount = -$$in.amount$$ where accNum = $$in.accNum$$;
        update bank set preBalance = balance where accNum = $$in.accNum$$;
        update bank set balance = balance - $$in.amount$$ where accNum = $$in.accNum$$;
        select name, accNum, suffix, balance, preBalance, amount from bank where accNum = $$in.accNum$$;
    endexec
    beginconfirm  /* confirm step, in SQL format, two-phase commit */
      commit tran sybasebank;
    endconfirm
    beginundo  /* undo step, in SQL format, two-phase commit */
      rollback tran sybasebank;
    endundo
  endsubtrans;

  subtrans savings (in) : account use ingres at interbase8 output
    beginexec  /* execution step, in SQL format */
      update bank set preBalance = balance where accNum = $$in.accNum$$;
      update bank set amount = -$$in.amount$$ where accNum = $$in.accNum$$;
      update bank set balance = balance - $$in.amount$$ where accNum = $$in.accNum$$;
      select name, accNum, suffix, balance, preBalance, amount from bank where accNum = $$in.accNum$$;
    endexec
    beginundo  /* undo step, in SQL format, compensation */
      update bank set balance = balance + $$in.amount$$ where accNum = $$in.accNum$$;
    endundo
  endsubtrans;

  subtrans visa (in) : account use guru at interbase output
    beginexec  /* execution step, in SQL format */
      update bank set preBalance = balance where accNum = $$in.accNum$$;
      update bank set amount = -$$in.amount$$ where accNum = $$in.accNum$$;
      update bank set balance = balance - $$in.amount$$ where accNum = $$in.accNum$$;
      select name, accNum, suffix, balance, preBalance, amount from bank where accNum = $$in.accNum$$;
    endexec
    beginundo  /* undo step, in SQL format, compensation */
      update bank set balance = balance + $$in.amount$$ where accNum = $$in.accNum$$;
    endundo
  endsubtrans;

  dependency
    checking or savings : visa;
    visa : accept;
  enddep;

endprogram
```

Workflow Support in Carnot

Christine Tomlinson *Paul Attie* *Phil Cannata*[†]
Greg Meredith *Amit Sheth*[‡] *Munindar Singh* *Darrell Woelk*[§]
MCC, 3500 West Balcones Center Drive
Austin, Texas 78759

Abstract

The Carnot project has been developing a variety of technologies oriented towards the integration of heterogenous data and information resources. This article will describe several of these technologies: the extensible services switch (ESS), the tree spaces communication facility within the ESS used for script and results distribution, and task flow constraints and the declarative resource constraint base (DRCB). The latter is used to capture organizational knowledge about the way in which various resources are used and the constraints between the resources that are (often implicitly) maintained by various applications.

1 Introduction

The Carnot project in the Information Systems Division of MCC is developing an extremely flexible framework that supports the integration of heterogeneous components in a distributed computational setting. The overall goal is to provide more effective interoperability among heterogeneous information resources and applications than has previously been tractable. The principal need for such systems is in enterprise integration, both within and among organizations. The reader is referred to [Can91], [Huh92], and [Woe92] for further information on the Carnot project at MCC.

A principal feature of the Carnot technology is its support for the description and management of heterogeneous, distributed workflows. Figure 1 presents the major components that are involved. The Extensible Services Switch(ESS), in Figure 1 provides interpretive access to communications resources, information resources and applications resident at a site in a distributed system. There are two sorts of paradigms that may profitably be used to understand the ESS. First, the ESS can be thought of as a component of a distributed command and control interpreter that is used to implement heterogeneous distributed transaction execution and generalized workflow control. Second, the ESS may be viewed as a programmable application layer communications front-end for applications and other resource managers within a distributed information system. The ESS is essentially a programmable glue for binding software components to one another in a manner that enhances interoperability.

The ESS is constructed on top of a high-performance implementation of an interpreter for the Actor model [Agh90], enhanced with object-oriented mechanisms for inheritance and reflection. The language of the interpreter is called Rosette[Tom92] and has been developed over a period of five years for research in both parallel algorithm development and interpretive control of distributed applications.

[†]On assignment at MCC from Bellcore
[‡]Bellcore, Piscataway, NJ 08854-4182
[§]On assignment at MCC from NCR

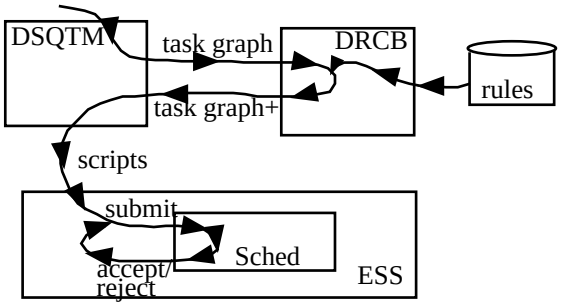


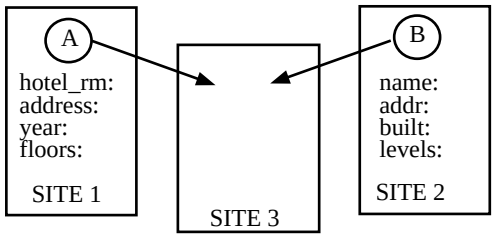
Figure 1: Control flow for scripts

Many different communications resource and distributed processing abstractions are supported by the ESS. These include: ByteStreams, RosetteStreams, Remote evaluation, TreeSpaces, and OSI. The ByteStream provides access to an uninterpreted stream of bytes while the RosetteStream provides access to a stream of Rosette expressions. This latter provides basic support for remote evaluation and is used by several of the client applications of the ESS. The TreeSpace abstraction is discussed below. Support for OSI includes an ASN.1 type system, interfaces to the Association Control Service and the Remote Operations Service.

The TreeSpace abstraction is built on RosetteStreams and provides functionality similar to that described in [Agh93] and [Car89]. A TreeSpace is similar to a directory in a Unix file system. It provides a collection of named entities which may be of several different classes: TreeSpace, LeafSpace, GroupSpace, or SpaceLink. LeafSpaces provide basic pattern directed retrieval of messages by anonymous recipients. They may be used to achieve various forms of synchronization since an attempt to retrieve a message via a pattern for which there is no match leads to the requestor blocking until a message is deposited by some other actor. GroupSpaces provide a multicast facility based on subscription lists, while SpaceLinks are used to attach a sub-tree of one TreeSpace to another TreeSpace. This latter facility is used to create networks of ESSes by attaching TreeSpaces at each ESS to one another under names that may be the site name or some other functional name representing the use of the ESS in the system.

2 Work Flow Processing

The Distributed Semantic Query/Transaction Manager(DSQTM) in Figure 1, dynamically expands a query to include access to all semantically equivalent and relevant information resources[Woe92] and also groups any updates to these resources as a set of independent tasks that interact according to a dynamically defined relaxed transaction semantics [Att93], [Woe93]. The DSQTM accepts user specifications of queries and updates and generates an initial task graph. The task graph is passed to the declarative resource constraint base (DRCB), which is implemented using Carnot's knowledge based distributed communicating agents. The DRCB uses rules to perform two types of transformations on the task graph. First, *articulation axioms* are used to expand the task graph to include nodes representing other information resources that contain relevant information. Articulation axioms express a set of logical relations between a local database schema and a global schema represented in a knowledge base. Articulation axioms are generated using the Enterprise Modeling and Model Integration tools that are part of the Carnot semantic services [Huh92]. For example, Figure 2 illustrates an initial query for a relation **A** at *site 1* and an expansion of the query to include access to a relation **B** at *site 2*. The results are to be sent to *site 3* where the union will be performed.



Original Query: SELECT A.hotel_rm, A.floors FROM A WHERE a.year < 1940
Augmented Query: UNION Original Query WITH
SELECT B.name, B.levels FROM B WHERE B.biult < 1940

Figure 2: Tree space example

```
Site 1:
(Out ['site3 'gtid42 'temp1
        (do-query itasca-agent-1
          "SELECT A.hotel_rm, A.floors FROM A WHERE A.year <1940")])

Site 2:
(Out ['site3 'gtid42 'temp2
        (do-select  ingres-agent-2
           "SELECT B.name, B.levels FROM B WHERE B.built <1940")])

Site 3:
 (do-semantic-union [["B"[[=1="one"] [=2="two"]]]]
          [[In 'gtid42 'temp1][In 'gtid42 'temp2]]))
```

Figure 3: Tree space scripts

Second, the articulation axioms and inter-resource constraints are used to derive additional update subtasks that must be performed in order to maintain enterprise-wide consistency. The execution of these subtasks is constrained by dependencies declared in a language similar to [Chr92] or [Kle91]. The DRCB augments the task graph with these additional subtasks and their derived interdependencies. The augmented task graph is passed back to the DSQTM which annotates it with physical resource locations. The DSQTM then generates an optimal query plan for each query node in the task graph. The query plan specifies the ordering of joins and the flow of intermediate results among the remote systems.

Finally, the task graph is used to generate a set of scripts to be executed by the ESS at each site that is involved in the task flow. A script controls the execution of subtasks at a site, controls the flow of data to and from the site, does necessary data value mapping between databases, and controls the transaction semantics. The scripts are executed under the control of the ESS using TreeSpaces to perform basic synchronization among tasks and inter-task dependency actors to enforce constraints among task events. For example, Figure 3 illustrates the scripts for performing the selects at each of site 1 and 2 and the union at site 3. The commands **Out** and **In** are used to move actors through the TreeSpaces at each of the three sites. The scripts are also distributed to each site via the distributed TreeSpaces. Although not illustrated in the example of Figure 3, the **sched** function of Figure 1 is performed by message exchanges among inter-task dependency actors. Tasks submit requests (e.g., **do-query** or **commit**) to an inter-task dependency actor with which they are associated and receive a status notification when any dependencies mentioning the requests are resolved.

DSQTM has been implemented as an ESS actor. The DSQTM actor responds to a set of messages that include the typical functions provided by a database server (connect to database, begin transaction,

execute query, etc.). Most of the functionality of DSQTM is provided by C and C++ functions, which are called through the ESS foreign function interface. This allows the ESS to provide flexible control of the concurrent execution of functions while implementing function kernels in C and C++.

References

[Agh90] G. Agha. Concurrent Object-Oriented Programming. *CACM*, Volume 33, Number 9, September 1990.

[Agh93] G. Agha and C. Callsen. ActorSpace: An Open Distributed Programming Paradigm. *Proceedings Principles and Practice of Parallel Programming*, 1993.

[Att93] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. *Proceedings of the 19th VLDB Conference* 1993, to appear.

[Can91] P. Cannata. The Irresistible Move Towards Interoperable Database Systems. *Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991.

[Car89] N. Carriero and D. Gelernter. Linda in Context. *CACM*, Volume 32, Number 4, April 1989.

[Chr92] P. Chrysanthis and K. Ramamritham. ACTA: The SAGA Continues. Chapter 10 in [Elm92].

[Elm92] A. Elmagarmid, editor, **Database Transaction Models**, Morgan Kaufman, 1992.

[Huh92] M. Huhns, N. Jacobs, T Ksiezyk, W. Shen, M. SIngh, and P. Cannata. Enterprise Information Modeling and Model Integration in Carnot. In [Pet92].

[Kle91] J. Klein. Advanced Rule Driven Transaction Management. *Proceedings of the IEEE COMPCON*, 1991.

[Pet92] C. Petrie Jr., editor, **Enterprise Integration Modeling**, MIT Press, 1992.

[Tom92] C. Tomlinson, G. Lavender, G. Meredith, D. Woelk, and P. Cannata. The Carnot Extensible Services Switch(ESS)—Support for Service Execution. In [Pet92].

[Woe92] D. Woelk, W. Shen, M. Huhns, and P. Cannata. Model Driven Enterprise Information Management in Carnot. In [Pet92].

[Woe93] D. Woelk, P. Attie, P. Cannata, G. Meredith, A. Sheth, M. Singh, C. Tomlinson. Task Scheduling Using Intertask Dependencies in Carnot. *Proceedings SIGMOD 93.*

On Transactional Workflows

Amit Sheth *
Bellcore
RRC-1J210, 444 Hoes Ln.
Piscataway NJ 08854
amit@ctt.bellcore.com

Marek Rusinkiewicz
University of Houston
Houston, TX 77204-3475
marek@cs.uh.edu

1 Introduction

The basic transaction model has evolved over time to incorporate more complex transactions structures and to take the advantage of semantics of higher-level operations that cannot be seen at the level of page reads and writes. Well known examples of such *extended transaction models* include nested and multi-level transactions. A number of *relaxed transaction models* have been defined in the last several years that permit a controlled relaxation of the transaction isolation and atomicity to better match the requirements of various database applications. Correctness criteria other than global serializability have also been proposed. Several examples of extended/relaxed transaction models are reported in [5].

Recently, transaction concepts have begun to be applied to support applications or activities that involve multiple tasks of possibly different types (including, but not limited to transactions) and executed over different types of entities (including, but not limited to DBMSs). The designer of such applications may specify inter-task dependencies to define task coordination requirements, and (sometimes) additional requirements for isolation, and failure atomicity of the application. We will refer to such applications as *multi-system transactional workflows*. While such workflows can be developed using *ad hoc* methods, it is desirable that they maintain at least some of the safeguards of transactions related to the correctness of computations and data integrity. Below, we discuss briefly the specification and execution issues in this evolving field, with emphasis on the role of database transaction concepts.

The idea of a workflow can be traced to Job Control Languages (JCL) of batch operating systems that allowed the user to specify a job as a collection of steps. Each step was an invocation of a program and the steps were executed as a sequence. Some steps could be executed conditionally. This simple idea was subsequently expanded in many products and research prototypes by allowing structuring of the activity, and providing control for concurrency and commitment. The extensions allow the designer of a multitask activity to specify the data and control flow among tasks and to selectively choose transactional characteristics of the activity, based on its semantics.

This area has been influenced by the concept of long running activities [3]. Workflows discussed in this paper may be "long running" or not. Related terms used in the database literature are task flow, multitransaction activities [7], multi-system applications [1], application multiactivities, and networked applications [4]. Some related issues are also addressed in various relaxed transaction models.

A fundamental problem with many extended and relaxed transaction models is that they provide a predefined set of properties that may or may be not required by the semantics of a particular activity. Another problem with adopting these models for designing and implementing workflows is that the

*Please send email for an extended version.

systems involved in the processing of a workflow may not provide support for facilities implied by an extended/relaxed transaction model. Furthermore, the extended and relaxed transaction models are mainly geared towards processing entities that are DBMSs that provide transaction management features (often assumed to be of a particular restrictive type), with the focus on preserving data consistency, and not on coordinating independent tasks on different entities, including legacy systems.

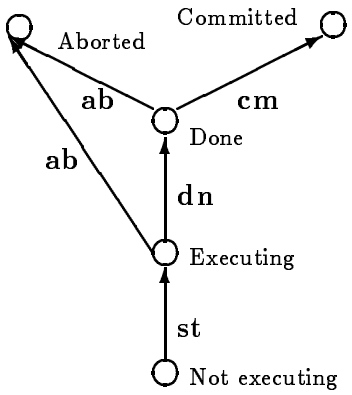
2 Specification of Tasks

A task in a workflow is a unit of work that is represented by sending a message, filling out a form, or executing a procedure, a contract or a transaction. A task can be processed by one or more entities, although we will limit our attention to the cases where a task is executed by only one entity, such as a DBMS or an application system.

An abstract model of a task is a state machine whose behavior can be defined by providing a *state transition diagram* (task skeleton). As with the correctness of traditional transactions, on the workflow level we do not model internal operations of the task – we deal only with those aspects of a task that are externally visible or controllable. In general, each task (and the corresponding automaton) can have a different internal structure resulting in a different task skeleton. One example corresponding to a standard transaction with a visible prepared to commit state, is shown below (cf: [10, 2]).

A task specification may include:

- a set of (externally) visible execution states of a task including an initial state and one or more termination states,
- a set of significant events that lead to transitions between these states, with each event identified by an attribute such as forcible, rejectable, and delayable (these are required to enforce inter-task dependencies [2]).



A task can be specified independently of the entity that can execute it or by considering the capabilities and the behavior of the executing entity. In the former case, it may be necessary to determine which entity can execute the task or the workflow system should be able to adequately simulate the states not supported by the task's executing entity. The latter case, in which a task is specified for execution by a specific entity or a specific type of entity is usually appropriate when dealing with existing (legacy) systems. The task skeleton then depends, to a large entent, on the characteristics of the system on which the task is executed. Some of the properties of the local system responsible for the task execution, like presence or absence of the two-phase commitment interface, will directly affect the form of the task skeleton and thus, the definition of the activity. Other characteristics of an executing entity may influence the properties of its task, without affecting its structure.

When the task is a transaction executed by a DBMS providing a full range of transaction management functions, we can take advantage of its concurrency control, commitment, recovery and access granting facilities. But when the task is executed by an application system, we must understand the application system semantics that affects its transactional behavior. Rather than developing "global" mechanisms that duplicate the functionality of local systems, we should build a model for managing multi-system workflows that utilizes the known task structures and semantics, coordination requirements of a collection of tasks, and execution semantics of systems that execute the tasks.

Workflow specification also consists of the conditions that affect the execution of tasks. These result from the specification of inter-task and inter-workflow execution requirements discussed next.

3 Dependencies and Correctness Criteria

Once the tasks constituting a workflow are specified, the internal structure of the workflow can be defined by specifying inter-task dependencies. Dependencies can be specified using a variety of software paradigms (e.g., rules, constraints, or programs). In general, dependencies can either be defined *á priori* (statically) or determined dynamically during its execution. In the first case, the tasks and dependencies among them are defined before the execution of the workflow starts. Some of the relaxed transaction models (e.g., [6],[13]) and [7]) use this approach.

A generalization of the static strategy is to have a precondition for execution of each task in the workflow or specific transitions of the tasks, so that all possible tasks in a workflow and their dependencies are known in advance, but only those tasks whose preconditions are satisfied, are executed [1]. Different initial parameters for the task may result in different executions of a task. The preconditions may be defined in terms of *execution states* of other (sibling) tasks, *output values* of other (sibling) tasks, and *external variables* including time and data states. The terms *execution dependencies, data or value dependencies and temporal dependencies* are used in the literature to refer to various scheduling preconditions. In the dynamic case, the task dependencies are created during the execution of a workflow, often by executing a set of rules. Examples of this kind of dependency specifications are found in long-running activities [3] and polytransactions [12].

The tasks of a workflow can communicate with each other through variables, local to the workflow and made persistent by the workflow system. These variables (including temporal variables) may also hold parameters for the task programs. The *data flow* between tasks is determined by assigning values to their input and output variables. In practice, there can be substantial difference in the format and representations of the data that is output by one task and input to another. The corresponding translation needs must be recognized but need not be an integral part of of the workflow model. The execution of a task has effects on the state of a database and the value of its output variable.

Additional aspects of intra- and inter-workflow specifications that are not captured using inter-task dependencies [1], include [11]:

- Failure atomicity requirements that can be defined using acceptable termination states of the workflow (committed or aborted).
- Execution atomicity requirements that define isolation properties of the workflow. Some of these requirements may be specified by providing the coupling modes between the tasks and requiring execution of tasks as atomic transactions,
- Dependencies that span across workflows. For example, it may be required that all tasks of one workflow must follow those of another at every execution entity.

[1] Some of these requirements are referred to as "correctness criteria" in [8].

4 Execution of Workflows

The correct execution of workflows involves enforcing all intertask dependencies, and assuring correctness of interleaved execution of multiple workflows. A scheduler (e.g., [2]) determines allowable transitions of each task based on different system and user events. These are then analyzed before allowing the corresponding transition(s) to take place or before terminating a workflow. By taking into account the semantics of tasks, workflows, and executing entities, we can significantly simplify the control needed to assure the correct concurrent execution of multiple workflows [9].

Two basic approaches to the implementation of a workflow management system can be identified: (a) An embedded approach that assumes that the executing entities support some active data management features. This approach is frequently used in dedicated systems developed to support a particular class of workflows and usually involves modification of the executing entities. (b) A layered approach that implements workflow control facilities on the top of uniform application-level interfaces to execution entities. A workflow manager based on such an approach is developed by the Carnot project at MCC. As a follow-on to the work reported in [1] and partly based on [2], we are currently working on a workflow management project that utilizes the latter approach.

Acknowledgements

Discussions with colleagues at Bellcore (N. Krishnakumar, L. Ness) and collaboration with members of MCC's Carnot project (P. Attie, P. Cannata, M. Singh, C. Tomlinson, D. Woelk) influenced this work.

References

[1] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proceedings of the 18th VLDB*, August 1992.

[2] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the 19th VLDB Conference*, 1993.

[3] U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. In *Proceedings of the 17th VLDB Conference*, September 1991.

[4] E. Dyson. Workflow. In *Forbes*, November 1992, p. 192.

[5] A. Elmagarmid, editor. *Transaction Models for Advanced Database Applications.* Morgan-Kaufmann, February 1992.

[6] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on VLDB*, 1990.

[7] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-transaction Activities. Technical Report CS-TR-247-90, Princeton University, February 1990.

[8] D. Georgakopoulos, M. Hornick, and P. Krychniak. An Environment for Specification and Management of Extended Transactions in DOMS. Technical Report September, GTE Laboratories Inc., 1992.

[9] W. Jin, L. Ness, M. Rusinkiewicz, A. Sheth. Concurrency Control and Recovery of Multidatabase Workflows in Telecommunication Applications. In *Proceedings of the SIGMOD Conference*, May 1993.

[10] J. Klein. Advanced Rule Driven Transaction Management. In *Proceedings of IEEE COMPCON*, 1991.

[11] M. Rusinkiewicz, A. Cichocki, P. Krychniak. Towards a Model for Multidatabase Transactions, *International Journal of Intelligent and Cooperative Information Systems*, Vol 1, No. 3, 1992.

[12] A. Sheth, M. Rusinkiewicz, and G. Karabatis. Using Polytransactions to Manage Interdependent Data. In [5].

[13] H. Wachter and A. Reuter. The ConTract Model. In [5].

Issues in Operation Flow Management for Long-Running Activities

Umeshwar Dayal *Ming-Chien Shan*
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA.

Abstract

Complex enterprises support heterogeneous, distributed, multi-system computing environments. Many activities in these enterprises are of long duration and consist of multiple operations that require different services, may execute on heterogeneous information systems, and may involve interaction with multiple individuals. Also, activities must be resilient to failures. In this paper, we explore the relevant issues that need to be addressed for supporting the automation of activities in such an environment. The Activity/Transaction Model of [3] meets most of these requirements. Based on this model, we are designing and implementing support for operation flow management as part of the Pegasus project [1].

1 Introduction

Today, economic imperatives are forcing enterprises to look for new information technologies to streamline their business processes. Key requirements include integrating heterogeneous information resources of the enterprise, and automating mission-critical applications that access these shared information resources. The Pegasus project at Hewlett-Packard Laboratories aims to meet these requirements. The Pegasus approach to integrating heterogeneous information resources is described in [1]. Here, we outline the issues we are exploring for automating operation flows in complex business activities.

Many of these activities are of long duration and consist of multiple operations executed over (possibly) heterogeneous systems. Examples of such applications are order processing, insurance claims processing, manufacturing, and patient tracking in hospitals. Such an activity typically starts with some business operation that is usually captured as a database transaction, a program execution, or even a human interaction. The execution of this operation may trigger other asynchronous and/or deferred operations. For most existing activities, the linkage among related operations is encoded in the application programs themselves, or documented in operational manuals to be enforced manually. Automating the control and data flows among operations will speed up the delivery and improve the quality of products or services offered by the enterprise, thus giving it a decisive competitive edge.

Two levels of effort are involved: the efficient and reliable implementation of individual operations, and the management of the overall flow among operations of an activity. The former is the subject of transaction processing in database systems. The latter resembles the *programming in the large* concept and aims at managing enterprise business activities along with their interacting operations. It involves the *high-level specification* of flows, and provides the *operational glue* and *environment support* for automating and managing the flows, recovering from failures, and enforcing consistency.

There are numerous systems that support *groupware* or *workflow* in an office systems context. However, activity management differs from these in some crucial respects. First, there is a difference in

scale. Workflow systems are intended mainly to automate document flows among human workers in an office. Activities, in contrast, span many organizations within an enterprise and even across enterprises. Workflow systems typically use electronic mail for delivering tasks to human workers. Little support for automatic operation execution, monitoring workflows, enforcing consistency, or recovering from failure is provided. The operations of an activity may be performed by application systems in addition to machines, robots, or humans; hence, mechanisms for invoking and monitoring application processes are necessary. The operations of an activity may access shared information resources, hence must be subject to concurrency control and recovery. Activities are defined and their status recorded in a database, which is used by the activity management system to monitor and control the flows.

In [2], we discussed the major challenges in supporting operation flow management. In Section 2, we summarize the issues that we are addressing in the Pegasus project. Then, in Section 3, we describe the approach we are taking to addressing these issues.

2 Major Issues

2.1 Operation flow model and language

Most of today's enterprise applications are implemented as a sequence of operations encoded as traditional transactions. The flows within and among these operations are coded into the application logic. If a transaction fails during execution, it is rolled back and possibly restarted. However, there is little support for failure handling or recovery across a multi-transaction application. This must also be coded into the application logic or be taken care of manually by humans. Extended transaction models [4, 5] provide some but not all of the required inter-transaction dependencies, error handling, and recovery capabilities. An operation flow model with more control flexibility and expressive power is needed.

To specify how enterprise activities are composed from well-defined operations, a high level specification language is needed. The language should support:

1. Basic operation definition: This includes the definition of the application step that is to be executed, the parameters to be passed to this step, any constraints to be satisfied before and after the step executes, and whether the step is to be executed as an atomic transaction.

2. Nonelectronic operations: Not every operation can performed as a database transaction or even computer-based process. The model and language should allow interaction with humans or external devices, for example, notifying a human to perform a task, or accepting human intervention to indicate that a task has been initiated and/or completed and what the results are.

3. Control and data flow definition: This includes defining the linkages and dependencies among operations. Sequential, parallel, conditional, and iterative control flows must be expressible. The data (parameters) passed from one operation to a downstream one must be specified. It would also be useful to allow the flows to be dynamically modified or overriden.

4. Failure and Exception handling: The semantics of activities have to be defined. Since activities are of long duration, treating an entire activity as an atomic transaction is infeasible. Failures should not cause an entire activity to be rolled back. Instead, partial rollback with forward recovery must be supported. Since an activity typically consists of several transactions, some of which may already have committed, rolling back may require compensating these committed transactions. Forward recovery may involve restarting the operations of the activity (from the point to which it was rolled back) or executing alternate operations. The language should allow the specification of compensation and alternate operations, and policies for rollback and forward recovery. These policies may include intervention by sophisticated users.

5. Business Rules and Constraints: Most enterprises have business rules and operational constraints. For example, special relationships may exist with preferred customers or suppliers; the quantity of goods stocked in a warehouse may be based on projected seasonal fluctuations in demand. It should be possible to define these business rules and constraints, so that they can be enforced during the execution of the activity.

6. Security and role resolution: Operations in an activity may be performed by humans or other organizational resources. The same person can dynamically assume different roles for different operations. Access constraints and organizational (e.g., role resolution) policies must be defined to govern the assignment of resources to operations.

2.2 Flow optimization and monitoring

While the model and language issues have been studied to some extent [3], the overall efficiency and control issues related to activity management have been less explored. These include flow analysis, optimization, and scheduling; and flow monitoring and control.

1. Flow analysis, optimization, and scheduling: A critical objective in automating business activities is the scheduling of operations and resources to reduce total elapsed time. To facilitate scheduling, cost-based flow optimization should be investigated. The problem requires estimating the costs of individual operations, compensation and alternate operations, and the costs of different scheduling and execution strategies. When dynamic modification of flows is permitted, these cost estimations become even harder. The cost model will need to include the occurrence probabilities of events that can modify the flow (e.g., failure events that cause rollback or alternative execution paths). Technologies developed in other areas such as Petri nets and query optimization can be used to facilitate this work. Temporal constraints (such as deadlines) further complicates scheduling, making the problem akin to scheduling transactions is real-time database systems.

2. Flow monitoring and control: Services must be provided for reliably recording events (e.g., the commit of an operation, the occurrence of a failure) and for invoking appropriate actions when these events are detected. This implies that the flow manager must be an active system. It must also reliably implement the linkages and dependencies among operations. Finally, it must log the execution history and status of an activity, and answer queries for this information. The information might be input to visualization tools that display the activity's execution and enable dynamic modification of the flow, or to analysis, reporting, and decision support tools.

2.3 Architecture

Given the trend towards open systems and standards, the flow manager must coexist with and take advantage of related components that are likely to exist in an open distributing computing environment. Recently, standards organizations such as OSF and OMG have announced major components relevant to operation flow management implementation. The Distributed Computing Environment (OSF/DCE) defines the architecture and basic services for distributed application development environments. It includes a location broker, which provides a remote request invocation service with location transparency, and security, naming, and transaction management services (Encina). The Distributed Management Environment (OSF/DME) defines a framework for event management. DME based products are now available and support the monitoring of network and system events. The OMG is working on an even broader domain of relevant technologies. The Common Object Request Broker and various object services, including naming and event services, can be expected to play an important role in the development of operation flow management systems.

3 Our Approach

In the Pegasus project, we are developing a general framework for operation flow management, utilizing standard components, and emphasizing performance and system robustness. We are designing a complete operation flow model and language, based on the Activity/Transaction Model (ATM) proposed in [3], which will address all the issues discussed above.

In the ATM approach, *activities* are treated explicitly as units of execution. An activity consists of multiple steps, each of which can be an activity or a transaction. Steps can be interactive and/or can include the invocation of an application program. Activities have weaker semantics than traditional database transactions: they are neither atomic nor serializable.

Activities are execution instances of *activity classes*. Control and data flow within an activity is specified via a predefined *script* in the activity class definition. The activity class also includes formal input and output parameters (which are bound to actual data when the activity is invoked and when it terminates). Activities may be nested to arbitrary levels, and the steps of an activity may execute sequentially or concurrently. The script language includes constructs for sequential and parallel execution, iteration, and branching; and supports data passing between steps. The flow specified in the script may be dynamically modified by *rules* triggered by events that occur as the activity executes.

The model supports various dependencies among transactional steps. A downstream step may execute in the same transaction scope as its predecessor, be deferred to the end of the predecessor transaction, or execute as a separate transaction that may be commit dependent on the predecessor.

Flexible exception handling is supported. Exception handlers may be attached to any step. Also, partial rollback with compensation, and forward recovery are supported. All of these can be specified in the script or via rules.

We plan to enhance HOSQL, the object-oriented data language of Pegasus, with features supporting operation flow specification and management. We will implement an operation flow manager that will provide flow optimization, monitoring, control, and query services. This will not replace a traditional transaction manager. Rather, it will use the services of local transaction managers in the component database systems to ensure the atomicity of operations that are executed against a single database system. If coordinating distributed transactions is deemed necessary, we will use a standard transaction manager service. In addition, some of the functional components defined by OSF or OMG will be evaluated and used as the basis of our implementation.

REFERENCES

[1] R. Ahmed, et. al., "An Overview of Pegasus", *Proc. of 3rd Intl. RIDE: IMS Workshop*, Vienna, Austria, April 1993.

[2] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M. Shan, "Third Generation TP Monitors:Database Challenges", *Proc. of ACM SIGMOD Int'l Conf. on Mgmt. of Data*, Washington, D.C., May 1993.

[3] U. Dayal, M. Hsu, and R. Ladin, "A Transactional Model for Long-Running Activities", *Proc. 17th Intl. Conf. on Very Large Data Bases*, Barcelona, Sept. 1991.

[4] A.K. Elmagarmid, *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.

[5] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers Inc., 1992.

Workflow Model and Execution *

Meichun Hsu *Ron Obermarck*
Digital Equipment Corporation
Activity Management Group
Palo Alto, CA 94301
{mshu,ronober}@pa.dec.com

Roelof Vuurboom
Digital Equipment Corporation
Workgroup Systems
7300 AE Apeldoorn, Netherlands
roelof@echo.tds.philips.nl

1 Introduction

A workflow system supports execution and monitoring of business processes, often accommodating multiple users and workgroups in these processes. The Echo workflow system has been developed based on a close collaboration with selected industrial users. In this paper we first present the Echo workflow system. Using the Echo workflow model as our example, we describe how workflow execution can be abstracted into inter-related process objects, whose state transitions are reliably recorded, tracked, and notified to participants of the workflow processes. We conclude with a discussion on the relationship between the process objects in workflow systems and transactions in TP systems.

2 The Echo Workflow Model

The workflow front end is usually tailored to a specific customer set. Depending on the workflow provider, the focus may be insurance claims processing, CIM, or an aspect of health care. In many instances, the system embeds, or contains interfaces for, image processing, forms or text processing. The presentation of most modern workflow systems is object-oriented GUI, WYSIWYG, and multi-media.

We present an overview of the ECHO (Electronic Case Handling Office) workflow system, and the model upon which it was designed. The application focus during development was the insurance industry. The system has been installed in customer sites in Europe. The ECHO workflow model itself has many attributes which combine to form a general model for workflow systems.

The Process Model

Echo's CTDL (Case Type Description Language) is a high level business process specification language. The language allows specification of both the procedures and their associated data.

The process is modelled as a partially ordered directed graph, consisting of nodes connected by directed arcs, where the nodes represent *activities* to be performed in the course of the process and the directed arcs the ordering of the allowed processing of the nodes.

There are four basic node types in a process: elementary activity (also known as a step), procedure abstraction (i.e., a nested composite activity), decision (i.e., a nested composite activity with process-data-dependent branching), and iteration (i.e., a composite activity with dynamic number of instantiations). These basic constructs allow for expressions of a rich set of procedures. Figure 1 illustrates a business process defined in terms of the four basic node types.

*This paper expresses views of the authors only, not opinions or future product plans of Digital Equipment Corporation.

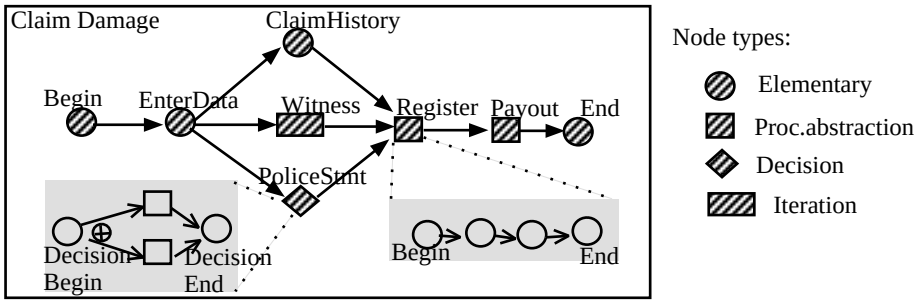


Figure 1: An Illustration of a Business Process "Claim Damage"

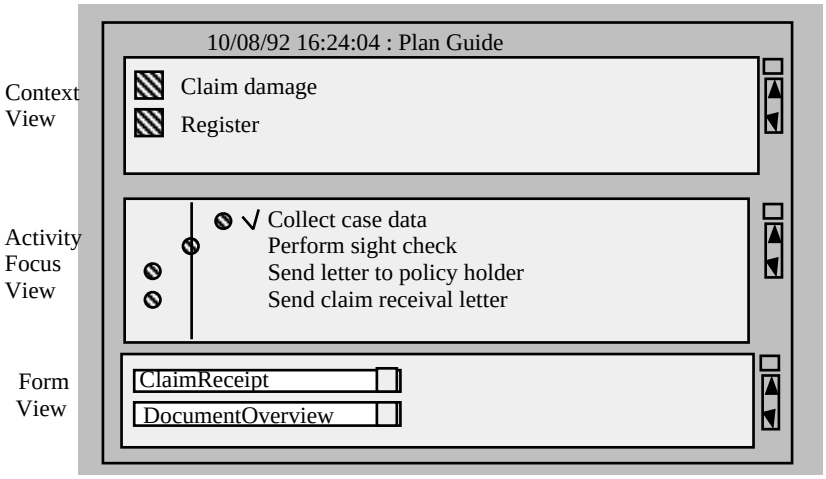


Figure 2: Illustration of Visual Plan Guide

The Visual Plan Guide and the Wavefront

The *visual plan guide* is a front-end tool in Echo that provides guidance and control for the participants of the business processes at execution time. It provides a graphical overview of the process status. The state visualized by the process guide represents the current progress achieved on the process; i.e., which parts of the work sequence have been completed, which are currently under process (known in Echo as the *wavefront*), and which are still to be done. The plan guide provides the user the ability to execute activities (nodes). The capability to execute a node in a process changes dynamically with the progression of the wavefront. Each node specifies authorisations (*roles*) required to perform operations.

Figure 2 illustrates the visual plan guide. The plan guide contains three views. The *context view* shows the position of the current activity in the nested structure of the process. The *activity focus view* provides a graphical representation of the plan at the current level. It presents the nodes as being in one of the three regions: past (checked), current (i.e., wavefront), and future. It provides access to the lower level view of any of the composite nodes in the view. For an elementary node, the associated application is invoked through which the user can complete that node. Completing the node changes the state of the node from open to completed. The *form view* provides direct access to forms (containing the data associated with the process) available in that particular context.

The visual plan guide allows a user to navigate through the process by going down one level by opening a subplan, or by going up a number of levels by closing a node in the context view. The contents of the context view, activity focus view, and form view will adapt to the current context.

Feedback Control

An essential part of human activity is the presence of feedback loops providing the ability to correct

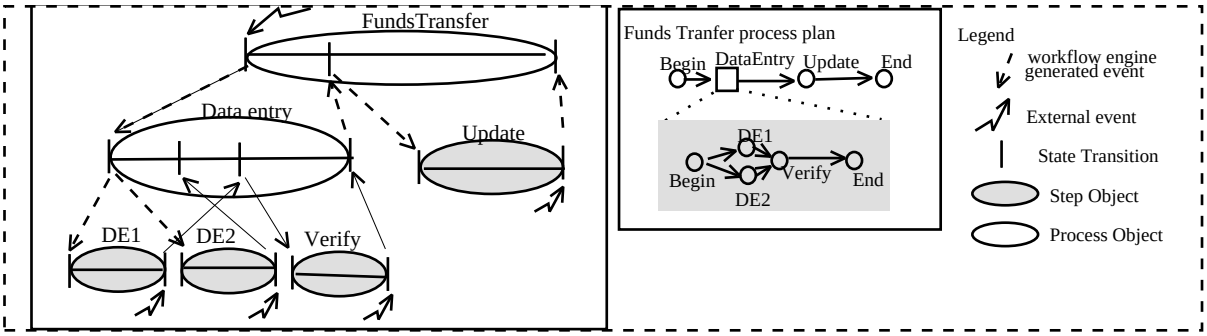


Figure 3: An Illustration of a Process Object Family Tree

and redo activities which may have been insufficiently or incorrectly carried out.

Feedback control is a primary mechanism for exception handling in Echo. The Echo model supports feedback loops by providing authorized users the ability to "roll back" the process to any point in the past, and to "redo" the activities from that point on. For example, a user may request to redo the *Collect case data* step as shown in the activity focus view in Figure 2, even though the wavefront of the activity has progressed past that step.

3 Reliable Workflow Execution

There is a distinct layering between the front end (the "client") and the engine in a workflow system. Functionally, the engine keeps track of the states and context data of processes, handles events occurring to the processes, and moves the processes forward by interpreting the events against the process plans and against the constraints among related processes. The engine is generic in that it focuses on triggering steps in the process plan while delegating the actual performance of the elementary steps as well as the user interface, such as the visual plan guide functions described previously, to the client components. We characterize the behavior of a workflow engine with a generic execution model described below.[1]

Process Execution Model

The engine abstracts the business processes as *process objects*. A process object is created by a creation event. The creation event gives the process object a process plan, and the data which defines the process object's initial data state. Once created, a process object becomes live and goes through state transitions according to the process plan until terminated.

Each state transition of a process object is triggered by an event. Upon receiving an event, the process object performs a state transition according to the process plan and the process state. The state transition may result in sending events to other process objects. Coordination among the process objects is achieved through enforcing conditions on their state transitions, and through channeling externally and internally generated events to related process objects.

A process object may send creation events to create child process objects. A child process object may be *chained* or *nested*. During the termination state transition, a nested child process object sends out an event to its parent process object, triggering a state transition in the parent. Process objects form a family tree, with the root being the top process object, and the leaves being the elementary step objects. Figure 3 illustrates a process object family tree with nested child processes.

External client components also generate events. External events constitute an interface between

[1] A preliminary version of this execution model was presented in the 1991 Workshop in High Performance Transaction Systems, Pacific Grove, CA, September 1991.

the workflow engine and the client components. An external event can trigger creation of a top process object, or it can be targeted at some process object or an elementary step object. External events are shown as squigle arrows in Figure 3.

The state of the workflow engine is represented by the collection of the states of the process objects in the engine. This state can be revealed to the client components (e.g., to be shown in a visual plan guide). It is through accessing the process object states that the client components are informed of pending steps. A client component carries out the tasks in a pending step and sends a step finish event (an external event) to the step object. The step object in turn terminates and sends a termination event to its parent, enabling the latter to move forward in its life cycle.

Reliable and Atomic State Transitions

The process objects in a workflow engine are generally long-running objects with multiple externally visible state transitions and are hardly atomic. However, state transitions of the process objects are executed as ACID transactions, ensuring that each state transition is failure-atomic and persistent. Transactional state transitions imply that the events that participate in the state transitions are consumed or produced atomically and reliably. Unconsumed events are persistently captured and are part of the states of the process objects in the system.

The workflow engine may be layered on a DBMS and transaction manager, and rely on them to provide a reliable base, or can implement its own reliable storage and coordination.[2]

In order to reliably integrate with client component's transaction processing, the workflow engine must be aware of, and be able to reliably record, the failure of the client transaction as well as its successful completion. This implies that with current standard transaction managers, the workflow system must participate as a "resource manager". Current transaction managers do not reliably inform applications of the failure of transactions in which the application was involved. Resource managers involved in the transaction are so informed.

4 Concluding Remarks

There is one valid model of transaction which seems constant, and that is the ACID model. Various optimizations and extensions have been proposed, but unless they maintain the ACID property, they are really coordinated activities.

Workflow is a coordinated activity set which controls a business process. There is nothing atomic about the aggregate progression and regression which may occur in the real-world processing of an insurance claim, for example. One of the assumptions of workflow is that it includes human interaction and decision. That does *not* imply that workflow is unreliable.

To be reliable across the spectrum of failures which can occur, the workflow state transitions made by its process objects must either be reliably captured during normal processing, or recreated during recovery processing. A database management product which supports transactions supplies the means to atomically capture the record of state transitions, as well as the redundancy to guard against loss of the record due to media failure. The recovery of the state of a workflow when the system uses a database manager is defined by the last successfully committed transaction (which is the last workflow state transition made by its process objects) before the failure occurred.

A reliable workflow management system does not extend the transaction to a business process. Rather, it uses transactions to reliably capture the state changes made by its process objects.

Acknowledgement: The authors wish to thank colleagues at Apeldoorn, Holland, and Palo Alto, CA, for their contribution to the work on Echo and RFM.

[2]Transactional queues can also be used to reliably capture unconsumed events (refer also to the paper by Bernstein, Hsu and Mann, "Implementing Recoverable Requests using Queues," *Proc. ACM SIGMOD Conf., 1990.*

ActionWorkflow^{TM} as the Enterprise Integration Technology

Raul Medina-Mora *Harry K.T. Wong* *Pablo Flores*
Action Technologies, Inc.

1 Introduction

We characterize "workflow management" as systematic organizational communication, coordination, and actions among people. For the past ten years we and our colleagues at Action Technologies have been developing our workflow theory, which we call ActionWorkflow^{TM} and computer software implementation of the theory, which we call ActionWorkflow System (AWS). The reader is referred to [1,2] for a more in-depth description of the theory and implementation of AWS.

2 Basics of the ActionWorkflow^{TM} Methodology

The design methodology focuses on the agreements between people. It shows the structures of agreements between people to produce customer satisfaction. The methodology classifies all units of work into workflows, and identifies for each workflow, a *performer* who is doing work for a *customer*. Each workflow unit is graphically represented as a *workflow loop* (see figure 1). Processes can be modeled as a series of workflows, with different participants assigned the roles of customer, performer, and observers within each workflow.

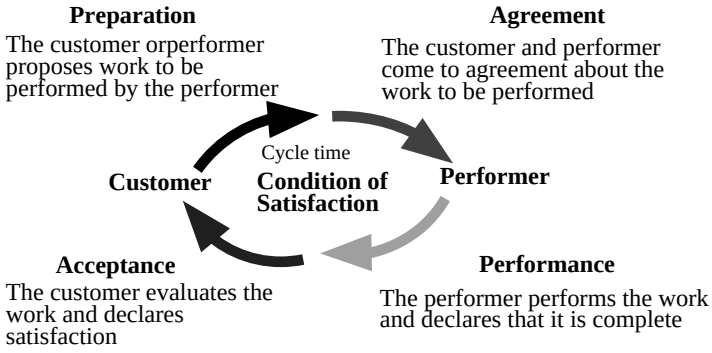


Figure 1: Basic Structure of a Workflow Loop

Each workflow is divided into the four phases in which customers and performers coordinate with each other: The first phase is *preparation*, in which the customer prepares to ask for something, often, for example, by filling out a form or preparing an e-mail message. When the customer asks for something, the workflow enters a phase of *negotiation*, in which the customer and performer agree on the work to be performed, clarifying what are termed the *conditions of satisfaction.* Once this agreement has happened, the workflow enters the phase of *performance*, in which the work is done and

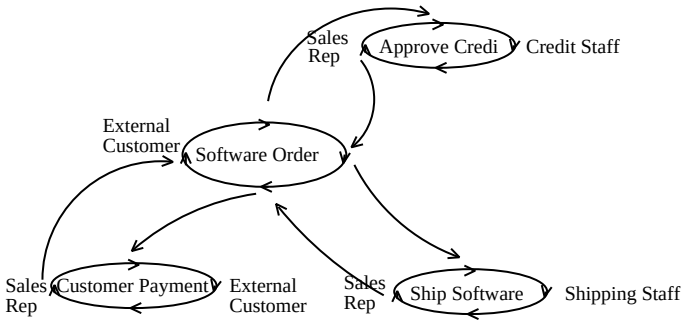


Figure 2: A Simple Business Process Map

the performer declares that the work is complete. The workflow enters then a final, critical, *acceptance* phase, in which the customer either reports satisfaction or dissatisfaction with the work.

Most business processes are made up of many workflows, and the workflows are connected. For example, as can be seen in Figure 2, a customer orders software over the phone, and this kicks off several new workflows designed to ensure that the customer gets the product and the company gets paid. We represent the relationships among workflows by putting all of a business process' workflows on a map, and drawing links between them indicating what phase in a parent workflow kicks off a sub-workflow.

Process maps are integral parts of both Action's design methodology and its workflow technology. Maps create a common graphical language that ties together all stages of process automation, from the early stage of analysis and design through to development and application deployment.

3 AWS Implementation

The AWS consists of the following products in the Windows and OS/2 environments.

ActionWorkflow Analyst^{TM} The Analyst is a graphical tool for non-programmers such as analysts and for business consultants. It is used to analyze and design the flow of work within an organization.

ActionWorkflow Application Builder^{TM} The Application Builder is a GUI application that allows a business process designer to specify the business process map and from which a "definition" database is generated.

ActionWorkflow Manager^{TM} The ActionWorkflow Manager is implemented as a server managing the workflows defined by the Application Builder. The Manager's services can be accessed using a collection of ActionWorkflow APIs.

ActionWorkflow Standard Transaction Format^{TM} **(STF) Processors** ActionWorkflow technology provides an additional interface to access workflow services called the Standard Transaction Format (STF). STF enables an open application development environment in which applications can access workflow services using ActionWorkflow STF Processors via messaging, database access, and application-to-application communications. STF Processors are one of the key facilities for ActionWorkflow users to integrate external data and applications into a common business process.

ActionWorkflow Reporter^{TM} The ActionWorkflow Reporter is a GUI application that provides facilities for managers and process participants to monitor business process operations.

4 Highlights of AWS

ActionWorkflow technology embodies tools to facilitate the entire life cycle of business process development, including analysis and design (modeling), generation, and management. This approach to process automation yields the following important benefits:

Human actions as system events - Human actions (such as approving a travel request or requesting a software shipment) constitute the events that trigger AWS to perform predefined actions, such as initiating new workflows, taking automatic action in other workflows, invoking external processes and integrating their results, and sending notifications to other participants. These human actions can be explicitly taken, or taken by the system on behalf of the users, according to the process definition.

Automatic Status Tracking - Process participants can view the exact status of their own tasks, and they can also determine the exact status of work in the entire process. Workers can ask, "What do I need to do next?" or "Where am I late?" and receive specific answers.

Deadline Management - Deadlines of individuals' tasks within a business process can be represented and supported. Reminders can be sent automatically before the passing of a task deadline. Follow-up messages can also be sent based on a recurring policy specified by the process designer.

Supervisory Tools - Managers of a business process can obtain the business process status in his/her organization. Questions can be asked such as, "Who is late on this task?" "What happened to the order?" "What needs to happen to complete the process?"

Business Process Performance Monitoring - The overall performance of a business process can be queried and monitored. "Hot spots" and problem areas can be identified in order to improve the process. The manager can ask questions such as: "What is the average delay of the credit checking task?" "What is the throughput for John Smith?"

Incremental Automation Customers and peformers of workflows can be replaced by software agents. The replacement can be carried out over time, thereby allowing a business process to be automated incrementally. This allows for increased emphasis on decision-making and developing new offers.

Business Process Versioning A workflow-enabled process can be changed dynamically, without having to first stop and flush the old process. Both processes, the old and the new upgraded one, can exist side-by-side until all work items already begun have completed their pass through the old process. At that point the old process is automatically removed, and all work continues to flow through the new process definition.

5 Research Opportunities Supporting Workflows

AWS is implemented on top of database systems. Our experience of building and using the system has revealed several important and interesting database and systems research areas that will enhance our workflow work. They are described as follows:

ActionWorkflow as Semantic Business Model. Just like the Object Oriented DBMSs enhance the representation power of data models, we believe the ActionWorkflow scheme of representing business processes provides an interesting direction of data modeling by introducing people as the central part of the model along with data and processes. Our experience of using this model has convinced us that in order to represent a complex situation such as how a business is being run, one needs to go beyond the traditional data analysis approach of looking at just data.

Process Integrity. A business process goes through many related steps before it is completed. The current database transaction model requires us to divide a business process into many small transactions. A mechanism of ensuring the integrity of process data spanning the life time of a business process is very beneficial. The current research on extended transaction shares similar motivation, we believe that our process model provides a solid context for such research.

Alerting Mechanism. A strong advantage of our technology is that it can integrate multiple applications together. STF Processors are examples where diversified applications can be tied together through a business process. A sophisticated database alerting mechanism will facilitate the implementation of these Processors because they can be activated by certain pre-specified conditions in the data and the appropriate workflow can be invoked. Fruitful results are likely if one concentrates on adopting some of the current active database research supporting workflow applications.

Time. Time is handled in ActionWorkflow very differently from most databases. It is "process centric" rather than "data centric". Time is an intrinsic part of a workflow. It is used by the user

to ensure that the person responsible for a workflow will act on the workflow. If the person is late, the time element is used to generate a follow up message. This treatment of time could be a useful mechanism in a database system where data and processes can be tied together in a proactive manner.

Process data type. The business process map has very rich semantics associated with it, including graphical, textual, and spatial information. We found the blob data type available in most database systems to be general but semantically weak. What is needed is a data typing mechanism to store and manipulate these large objects in a more natural and efficient manner.

Security based on Organizational Roles. The security mechanism in ActionWorkflow is based on the privileges assigned to organizational roles, and not to people directly. A separate, many-to-many mapping between roles and individuals within an organization can be set and reset dynamically by authorized people and workflow processes. An interesting research direction is to incorporate the concepts of organizational roles and role-to-individual mapping to the grant/revoke scheme of the relational systems.

"Practical" Distributed Database. Using ActionWorkflow as an enterprise integration solution, we found that we almost never dealt with distributed data but rather distributed data AND applications. While the current distributed database research will yield very useful results, we found that a more practical, arms-length approach to distributed data and applications fits many real world situations really well. The STF Processors in ActionWorkflow are such examples that we manage to tie many existing large databases and their applications together in the same business process.

6 Summary and Conclusion

After several decades of data processing, we have learned that we have not won the battle of modeling and automating complex enterprises. These enterprises typically have the requirements of coordinating multiple people in multiple applications over long duration across multiple organizations. Currently isolated islands of automation are used to satisfy local and intermediate needs. But the bridging of these islands is typically very complex and expensive. The reason is that the current approach based on traditional database systems, programming languages, and more recent technique such as Object Oriented programming, does not provide the environment for this type of applications.

We have been working on such an environment for the past ten years, with products such as The Coordinator^{TM} MHS, and recently AWS. The AWS allows the modeling of complex business processes using primitives so that declaratively you can build workflow models capturing the complex coordination among multiple individuals. The model can be used as an integration platform to tie multiple applications and their data together while retaining their autonomy. Enterprises can be automated with people at the core of the process. New and existing applications can participate in the workflow processes and become part of the whole system. More importantly, our technology is driven by a proven and sound business process methodology which guides users to model their complex processes. This tight coupling of methodology and technology provides a powerful environment of enterprise modeling and automation. We believe there are many potentially fruitful research opportunities in database systems in the context of this higher level semantics of business processes in AWS.

References

[1] Flores, F. and Winograd, T. *Understanding Computers and Cognition,* Addison-Wesley, 1987

[2] Medina-Mora, R. Winograd, T. Flores, R. and Flores, F. "The Action Workflow Approach to Workflow Management Technology", *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW)*, Toronto, Canada, November, 1992.

Workflow and Transactions in InConcert

Dennis R. McCarthy *Sunil K. Sarin*
XSoft Advanced Information Technology
Four Cambridge Center, Cambridge, MA 02142
{mccarthy,sarin}@xait.xerox.com

1 Introduction

Workflow technology is becoming increasingly visible as a means for organizations to improve their productivity and competitive position, via automation and reengineering of business processes. Workflow systems offer organizations the ability to model, execute, report on, and dynamically control work processes that typically involve multiple people in collaboration. Individual users also have the opportunity to see how their tasks interact with other users' tasks and how these all fit together in a business process to achieve organizational goals.

The InConcert workflow management system [Sarin91] is designed to address both the above organizational objectives as well as improve individual users' ability to manage their work. By presenting an interface in terms of "tasks" that are meaningful to the user (e.g., preparing the user's contribution to their department's annual budget), the nature of interaction between user and computer is raised to a more productive level. The user no longer needs to deal with low-level concepts like files (sometimes fashionably called "objects") and applications that are invoked against these files. Rather, the user can view their work as a collection of assigned tasks perhaps organized by priority, deadline, or other criteria. On selecting a task to work on, the user is presented with a task context in which the necessary documents and applications are directly accessible. The user may well invoke multiple diverse applications (e.g., a word processor and a spreadsheet) as well as perform work outside the computer system (e.g., make a phone call) in order to accomplish a single task.

2 Workflow Model

The primary concept in the InConcert workflow model is that of a *job*, which is a multi-person collaborative activity. A job consists of *tasks*, each of which is a unit of work that can be performed by one person. Tasks can be decomposed into subtasks, to obtain a hierarchical work breakdown structure (WBS). Tasks at the same level in such a hierarchy may have ordering *dependencies* defined among them, such that a "dependent" task cannot be worked on until the "precedent" task has been completed. The definition of dependencies allows for as much or as little concurrency among tasks as is appropriate. Figure 1, for example, shows some tasks that can be performed in parallel and others that must be performed in the specified order.

Each task in a job has an assigned *role*, which is a logical placeholder (with an appropriate generic name, e.g., "Editor" for a technical report) for the *user* (person or program) that will perform the task. Each task in a job may also have any number of *references*, which are placeholders for *documents* that are likely to be needed (for update, or as reference material) in performing the task. Documents are "routed" among users implicitly in InConcert, by virtue of the same document being referenced in multiple tasks.

sarinfig1.ps

Figure 1: Example Task Structure

Documents in turn are simply abstract data objects which have *content* that can be manipulated by the appropriate application (e.g., word processor or spreadsheet). The content of a document resides in a *repository* which is an abstraction of document storage and retrieval services. Documents may also have *links* to other related documents.

An InConcert job is activated by instantiating a *template* which provides a predefined task structure for performing a given business process. In the course of execution of a particular job, the task structure can be modified by adding and deleting tasks and dependencies. This provides users with the flexibility needed to deal with the inevitable exceptions and deviations from the idealized process descriptions. For example, a document review job with the structure shown in Figure 1 above could be modified to have additional or fewer reviewers, or an additional review cycle after the first round of comments.

InConcert maintains an *audit log* which is a record of significant *events*. Events consist of state changes such as task becoming ready to work on or overdue, or document checkout or checkin, as well as additional event types that are defined by the application. *Triggers* can be defined which cause a specified *action* to be invoked when a particular kind of event occurs. The action invoked may be creation of a new InConcert job, notification of users via electronic mail, or invocation of an application-supplied procedure (via remote procedure call - see Section 3).

Jobs and tasks, which can last for a long time, have a natural correspondence with long "engineering transactions". However, the actions performed in a task are often subject to human judgment and may be hard to predict. Users may therefore need to perform *ad hoc* coordination (within the contexts of the tasks they are working on) using the following capabilities:

- Document *checkout* and *checkin* ensure that the same document is not modified concurrently by multiple users. Users may view, read only, a document that is checked out by another user, and may determine who has the document checked out and in what task and job.

- Document *version histories* provide a record of past checked-in versions. Because checkout and checkin in InConcert are always performed in the context of a task, it is possible for users to determine what changes were made in a given task (or job).

The above primitives provide the foundation for users to perform their own concurrency control and recovery. For example, if a user wishes to relinquish a task and not work on it, or if the user to whom they are responsible is not satisfied and wishes to have the task "redone", InConcert does

not automatically roll back (or compensate for) the changes performed by the user within the task. Rather, the users may examine the current state and past versions of the relevant documents in order to determine what to undo and what to retain. As we gain experience with customers' use of InConcert, we may apply more sophisticated transaction management concepts [Elmagarmid92] where appropriate.

3 System Architecture

The InConcert system is a practical realization of the model described above. The goal was to produce a system with the following architectural attributes:

Open and Extensible: Workflow management systems bring together users and their work, including the documents and applications they use to perform that work. The organizations that will use workflow systems already have computer networks, personal productivity applications (e.g., word processors and spreadsheets), and documents (stored in files, databases, etc.). To be useful, a workflow management system must fit into the organization's existing computing environment.

InConcert is open and extensible in a variety of ways. Existing personal productivity applications can be integrated with InConcert by simply adding an entry to a configuration file. When a user opens a document icon from an InConcert graphical user interface, the application is launched and loaded with the document content.

The concept of repository, which is an abstract service for storing and retrieving document content, also allows for extension and customization. The basic repository type provided by InConcert reads and writes document contents as operating system files (which may be network-accessible). Additional types of repositories can be integrating by implementing content read and write (and delete) operations that conform to a specified interface.

InConcert triggers are also extensible. The event for a trigger can be detected by the InConcert server or by an external application; InConcert provides functions for defining and reporting the latter kind of event. The action for a trigger can be executed by the InConcert server or by an external application. When a trigger with such an external action fires, the InConcert server performs a remote procedure call using information specified when the action was registered by the application.

InConcert includes an application programming interface (API) as well as graphical user interfaces. The graphical user interfaces access InConcert through this API. Everything that can be done through an InConcert graphical user interface can also be done through the API. Developers can use the API to produce user interfaces or agents that are customized for a particular organization or workflow. C and C++ language bindings are provided for the API.

Based on Standards: The InConcert process and document management services can be thought of as "middle-ware". That is, InConcert is implemented over conventional computing infrastructure, and provides its users with a semantically richer set of objects (e.g., jobs, tasks, documents) and operations. InConcert uses standard interfaces to underlying services: POSIX for the operating system and file system; SQL for the DBMS (until an object-oriented database standard becomes available); TIRPC for the network; Motif, OpenWindows, or MS-Windows for the window system. This ensures that InConcert is easily ported to new computing environments that provide different implementations of these interfaces. Furthermore, the graphical user interfaces provided with InConcert conform to the look and feel standards for the desktop environments in which they operate.

Object-Oriented: InConcert is object-oriented, both in its interface and implementation. The entities in the model are implemented as object classes, and the operations in the API are methods for the object classes. Users can define subclasses of the predefined model classes, and can add new attributes to any class. In the graphical user interface, objects are presented as icons, and methods are presented as (selection sensitive) menu choices. In the application programming interface, the entities in the

model are expressed as C++ classes, and the operations on an entity type are class member functions. The InConcert system is implemented in C++.

Client-Server: InConcert employs a client/server architecture. InConcert client applications (e.g., graphical user interfaces, agents, administrative utilities) typically run on desktop computers (PC's, workstations). The InConcert server typically runs on a server host. Heterogeneous networks are supported (e.g., MS-Windows client and UNIX server). A client may interact with multiple servers in a large organization.

The InConcert API is the interface between client and server. The API is a library consisting of object classes that correspond to the entities in the model. The operations on each class are member functions in this client library, which is linked with the client application. A client application obtains services by calling these API functions. When a client calls a function that retrieves or updates a shared persistent object, the client library makes a remote procedure call to the InConcert server.

The server is responsible for managing the persistent objects that are shared among clients. The state of the object instances are stored in a relational DBMS. In executing an API operation, the server reads or updates the state of objects by reading or updating tables in its database. A single API call by a client may result in multiple database accesses. The server executes each API call in its own database transaction. This guarantees that each API call executes atomically, and that multiple API calls can be processed in parallel subject to DBMS concurrency control.

The audit log of events is stored in a database table, and can be examined using the query and reporting capabilities of the DBMS. The audit log is also used by the server for triggering actions, using an asynchronous execution model [McCarthy89]. That is, actions triggered by an event are performed in transactions separate from the transaction that reported the event. When a transaction writes an event to the audit log, descriptions of actions to be performed, if any, are also written to the database. This guarantees execution of triggered actions, even if the server fails.

Although the transaction mechanism of the underlying DBMS provides consistency and atomicity, these guarantees do not extend to InConcert operations outside the DBMS. For example, updating a document's content involves both writing a new content to the repository and updating the document's metadata (stored in the DBMS) to reflect the new content handle. The combination of these two cannot be made atomic, and it is possible after a failure to have a content in the repository that is not referenced by any document in the database. Such unreferenced contents need to be cleaned up after the fact using administrative tools. Similarly, when invoking an application-defined triggered action, it is not possible to make the remote procedure call execution and the marking of the action as completed (by deleting the triggered action description) atomic, because the former is not under the control of a DBMS transaction. These problems could be alleviated if a distributed atomic transaction facility [Gray93] were available, that was not limited to DBMSs alone but supported a published standardized commit protocol in which all of the data stores and application procedures could participate in order to ensure atomicity.

REFERENCES

[Elmagarmid92] A.K. Elmagarmid, ed. *Database Transaction Models for Advanced Applications.* Morgan Kaufman Publishers, 1992.

[Gray93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman Publishers, 1993.

[McCarthy89] D.R. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. *Proc. ACM SIGMOD Conference*, May-June 1989.

[Sarin91] S.K. Sarin, K.R. Abbott, and D.R. McCarthy. A Process Model and System for Supporting Collaborative Work. *Proc ACM-SIGOIS Conference on Organizational Computing Systems*, November 1991.

../JUNE93/back_inside.ps

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

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