Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows*

Y. Breitbart,* A. Deacon,* H.-J. Schek,* A. Sheth,† G. Weikum*

*Dept. of Computer Science, ETH Zürich, CH-8092, Switzerland
{breitbar, deacon, schek, weikum}@inf.ethz.ch
†Bellcore, 444 Hoes Lane, Piscataway, NJ 08854, USA
amit@ctt.bellcore.com

Abstract

Workflow management is primarily concerned with dependencies between the tasks of a workflow, to ensure correct control flow and data flow. Transaction management, on the other hand, is concerned with preserving data dependencies by preventing execution of conflicting operations from multiple, concurrently executing tasks or transactions. In this paper we argue that many applications will be served better if the properties of transaction and workflow models are supported by an integrated architecture. We also present preliminary ideas towards such an architecture.

1 Introduction

Many applications in a current data processing environment involve operations on multiple systems. Examples of multi-system application include travel reservation (often discussed in literature), service order processing [1], and loan processing in a bank. Current transaction-oriented information systems provide only limited support for managing the control and data workflow of multi-system applications. TP-monitors [7, 13] provide mechanisms for task coordination in the form of "queued transactions"; however, the task coordination structure is typically difficult to understand and manage, as it is scattered through the application program code.

There is thus a need to separate control and data flows in a multi-system application from the rest of the application code. A workflow model uses a declarative control and data flow specification language to facilitate a separation between the application code, and the control and data flows between different tasks of the application. A number of workflow activity models [1, 6, 17, 18], and models to specify intertask dependencies [2, 11, 14], have been proposed. These models have generally focused on modeling application level tasks, and dependencies among these tasks. The emphasis in these approaches has been on understanding the application tasks and their correct execution; therefore we call these models "application-centric."

In loosely coupled environments with highly autonomous systems, it may be sufficient that data consistency is enforced independently in each database. In particular, if integrity constraints do not span multiple systems, there is no need for establishing higher-level 'spheres of control' to prevent tasks of different workflows from interleaving in an incorrect manner. In a more tightly coupled multi-system environment, however, there is a strong need to manage more general dependencies and to enforce consistency between data stored in different databases. Thus, there is a need to prevent interference between concurrent workflows and handle failures as supported in classical transaction management. The emphasis in the multidatabase transaction approaches has been on controlling how data operations may interleave to produce a correct execution; therefore we call these approaches "data centric."

A careful combination of the features of the data-centric multidatabase transactions and application-centric workflow models can permit simplified specification of workflow correctness and a more efficient implementation. In this paper we outline an approach towards such a combined model. The proposed approach aims to integrate many features of a workflow specification model [2, 17] and an open nested transaction model [20] in such a way that the weaknesses of each model will be compensated by the strengths of the other.

The paper is organized as follows. In the next section we present a loan request example that is used through the remainder of the paper to introduce our approach. Sections 3 and 4 briefly review task de-

*This position paper is based on discussions while A. Sheth was on a short sabbatical at ETH Zürich.

pendencies of a simple workflow model, and semantic serializability and atomicity issues as supported in the open nested transaction model, respectively. In Section 5 we formulate the basic principles of our approach to the integration of the workflow and multidatabase transaction models. Section 6 concludes the paper.

2 Workflow Example

Consider a simplified example of a bank loan request processing. The workflow for the loan request processing is represented in graphical form in Figure 1 and consists of the following tasks:

Enter Loan Request (EnLR):
Accepts the amount and terms of the loan and records the relevant information into a loan database.

Client Credit Worthiness (CCW):
Retrieves client information and assesses the credit worthiness. This can be a very complex query when the control of various subsidiaries and relationships in holding companies must also be considered.

Risk Evaluation (REv):
Evaluates the risk to the bank if the loan were granted, which again may be a very complex query.

Risk Update (RUp):
Updates the value of the bank's total involvement. This is performed only if the Risk Evaluation succeeds. It also implicitly assumes that the loan request will eventually be granted, which is not necessarily the case.

Risk Update Compensation (RUpC):
Compensates the effects of the above Risk Update task should the loan request be ultimately rejected. This needs to be performed only if the Risk Update succeeded. The compensation cannot simply restore the original value of the bank's total involvement that existed prior to the Risk Update task executing, since other tasks may have changed the value in the interim. Rather the compensation is performed by decrementing the value of the total involvement.

Risk Exception (REx):
Approves the risk to the bank if the loan were granted. This task is invoked if the regular risk evaluation could not succeed or could not be executed because of temporary unavailability of some data. The client may be important and have a convincing argument why the loan should be granted, despite the evaluation failing. Under these circumstances, an executive officer of the loan department would have to approve the risk.

Enter Decision (EnD):
Enters the decision to either grant or reject the loan request and returns information on the agreed terms of the loan. The loan can only be granted if the Client Credit Worthiness, and the Risk Evaluation or Risk Exception tasks returned positive results, although the bank agent may still, for some other reason, reject the loan request.

The important characteristics of the application classes that we consider are: (1) multiple agents may invoke "competing" loan request workflows simultaneously; (2) the workflows involve tasks that share multiple heterogeneous databases; (3) the consistency of the bank's total involvement and client credit worthiness is of crucial importance for the correct processing of loan requests; (4) loan requests are usually processed within a few minutes, and at most within a few hours; (5) there is a need for a "correct" execution of workflow as determined by the intertask dependencies of the workflow, as well as a "correct" execution of concurrently executed tasks of different invocations of the workflow.

3 Workflow Specification

Dependencies allow the computational structure of workflows to be made declarative. This is important for expressing application semantics relevant to a workflow execution correctness and workflow data consistency requirements. Some common forms of dependencies are described below that are often required in a workflow specification.

3.1 Control Flow and Data Flow Dependencies

We assume that during the execution of a task in the workflow, the task can pass through several stages defined as task states (for example, task termination state, or task start state, or task waiting for execution state, etc.). Control and data flow dependencies are essentially of the form:

> *Task T_j can enter state s_j only after task T_i has entered state s_i.*

For example, the following is a control flow dependency: *Task T_j can (enter the state) start executing only after task T_i has started execution.* These type of dependencies may capture both a concurrent and sequential tasks execution and indicate data flow dependencies between tasks. Control flow dependencies may be based on data flow dependencies. For example, the Enter Loan Request task passes the amount

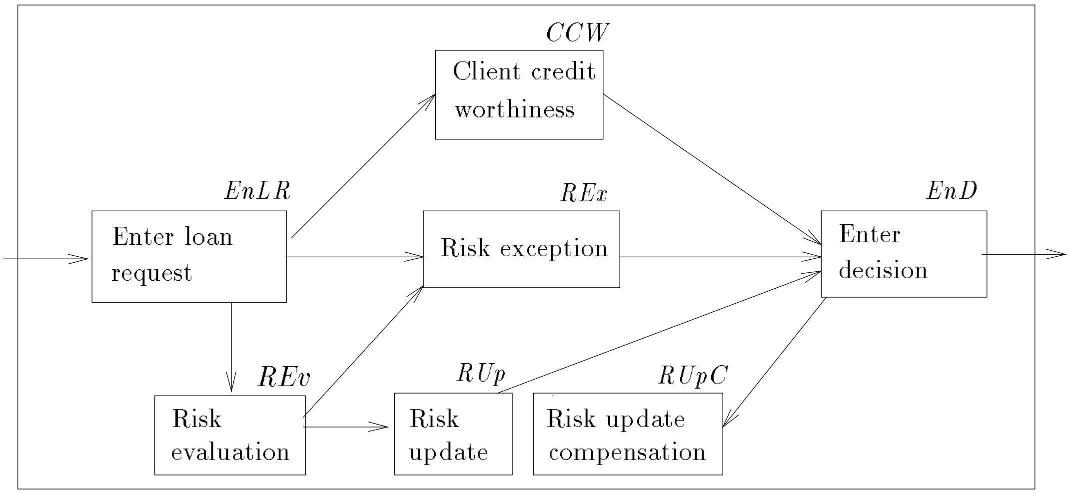

Figure 1: Execution precedence graph for loan request workflow.

and terms of the loan request to the Risk Evaluation task. The Risk Evaluation task would then execute only if the loan amount is between one hundred thousand and ten billion Swiss Francs.

In the generalized form, multiple such dependencies may hold between tasks. These dependencies can be expressed using a precedence graph, with nodes representing tasks and edges representing ordering constraints. The precedence graph showing the control specification for the loan request example is depicted in Figure 1. Some details have been omitted to simplify the presentation. A more rigorous form of such specifications could be based on, for example, predicate-transition nets [15] or propositional logic [2].

3.2 Contingency Dependencies

We distinguish between a semantic success or failure of a task, and a successful completion of a task indicated by a task commit. For example, a negative decision on a loan request represents a semantic failure as far as a loan request is concerned, while at the same time task Enter Decision has successfully committed. On the other hand, Risk Evaluation task may abort (for the lack of information for the task to proceed, for example) while Enter Decision task may make a positive decision on a loan request. In the case of tasks semantic failures we need to consider

contingency dependencies that are normally of the following form:

> *If task T_i semantically fails, then execute task T_j as a "contingency" task.*

The Risk Exception task provides an example of a contingency dependency. It must be executed if the Risk Evaluation task returns a negative result or cannot be executed to completion (which constitutes a semantic failure for Risk Evaluation).

A contingency dependency is similar to a control flow dependency, with a conditional execution depending on the return value and termination status of one or more previous tasks. The difference here, however, is that in the contingency dependency is defined between tasks based on a semantic failure of a task, while control dependencies define a relationships between different states of different tasks.

3.3 Task Termination Dependencies

Termination dependencies have the form:

> *If task T_i terminates in state s_i, then task T_j must terminate in state s_j,*

where a task may terminate in one of the following states:

- committed with a positive result ("semantic success"),
- committed with a negative result ("semantic failure"), or
- aborted (for system-internal reasons or program errors).

By explicitly referring to the three possible termination states given above (or, maybe, even more states if failure and success are further refined), the meaning of a task dependency can be stated precisely. For example, one can specify that the Enter Decision task must eventually be executed and committed if the initial Enter Loan Request task is committed, and that a semantic failure or an abort of the Client Credit Worthiness task should force the Enter Decision task to "fail" semantically (i.e., to commit with a negative result).

If these kinds of dependencies refers only to the commit and abort states and do not consider the semantic outcome of the tasks, then they are called commit/abort dependencies. If they do refer to the semantic success or failure of the task, then they are called success/failure dependencies.

Compensation dependencies are another special case of task termination dependencies that are often identified explicitly. The general form is:

> *If task T_i has committed and if some subsequent task T_j fails semantically or is aborted, requiring the effects of task T_i to be undone, then execute task T_i^{-1}, which must be guaranteed to commit eventually in order to compensate T_i.*

The additional task T_i^{-1} is referred to as a *compensating task* [10, 12]. The Risk Correction task of the loan request workflow is the compensating task for the Risk Update task.

3.4 Critical Task Dependencies

Critical Task dependencies have the following general form:

> *If task T_i fails semantically or aborts, then the entire workflow must fail, i.e., terminate in the semantic failure state.*

A critical task dependency is used for establishing a relationship between the termination states of one or more "distinguished" tasks with the termination state of the entire workflow. The possible termination states of an entire workflow would include a generic success and a generic failure state, where both of these states could be further refined based on the values of the workflow's output parameters.

A task T_i with the above property is called a *critical (or vital) task* [11]. In our loan request example, the Client Credit Worthiness and Enter Decision task are vital tasks.

4 Semantic Transactions

Semantic transaction models ensure data consistency by executing an entire workflow as a single transaction. A workflow submits for execution a sequence of tasks, that are the high-level building blocks (i.e., operations). To provide acceptable performance and simplify the control of interleavings, semantic properties of the workflow tasks are exploited (e.g., commutativity or compatibility). Based on these properties, a semantic transaction model supports "semantic serializability" and "semantic atomicity" of entire workflows. When considered with the correct execution of each individual workflow (observing the specified intra-workflow dependencies) this ensures data consistency across multiple systems.

4.1 Semantic Serializability

Compatibility of two tasks means that the ordering of the two tasks in a schedule is insignificant from an application point of view [19, 20]. A compatibility specification is used by a scheduler to determine which task executions can interleave. Compatibility must be explicitly declared and reasoned about by a human expert. We assume that a compatibility specification for the tasks is defined in the form of a compatibility matrix.

Figure 2 shows a portion of the compatibility matrix for the loan request example. The '+' indicates compatibility and '−' a conflict. The compatibility matrix thus specifies that the Risk Evaluation and Risk Update tasks of different workflows are in conflict. If the Risk Evaluation tasks of different workflows could interleave in an uncontrolled way, the bank's involvement could reach unacceptable levels. Note that a Risk Update Compensation does not conflict with the Risk Updates task, as it cannot increase the bank's involvement and thus cannot cause any serious problems to the bank.

A schedule is semantically serializable if an equivalent serial execution exists with the same ordering of conflicting tasks. The theory of open nested transactions allows us to consider the tasks as further subtransactions that can be executed in an interleaved fashion and committed independently of the schedulers at the topmost workflow level.

Consider as an example the interleaved execution of two loan request workflows. In each case the entire loan request workflow is viewed as a single transaction, and the tasks correspond to operations or, equivalently, subtransactions. A schedule is shown in Figure 3 where the execution order is from left to right. The number of tasks and parameters of tasks

	$EnLR$	REv	RUp	$RUpC$	REx	EnD
Enter Loan Request ($EnLR$)	+	+	+	+	+	+
Risk Evaluation (REv)	+	+	−	+	+	+
Risk Update (RUp)	+	−	−	+	+	+
Risk Update Compensation ($RUpC$)	+	+	+	+	+	+
Risk Exception (REx)	+	+	+	+	+	+
Enter Decision (EnD)	+	+	+	+	+	+

Figure 2: Compatibility matrix of the loan request example.

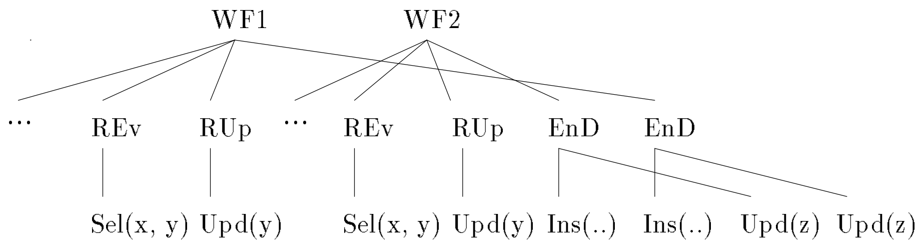

Figure 3: Semantically serializable schedule.

have been reduced to simplify the presentation. The SQL-like operations Insert, Select, and Update, invoked by the loan request tasks, are abbreviated as Ins, Sel, and Upd, respectively.

The schedule in Figure 3 is not serializable in terms of the SQL-like operations at the leaf level, since there is a conflict cycle involving the updates on the data objects y and z. However, the schedule is semantically serializable as it is equivalent to a serial execution with WF_1 preceding WF_2. This can be demonstrated using the following intuitive arguments.

The order of compatible operations can be exchanged. This permits us to change the order of the Upd(z) operation of WF_1's Enter Decision task and the preceding Ins(..) operation of the Enter Decision task of WF_2. In the resulting schedule, all subtrees at the level of loan request operations are isolated. Thus the execution of operations on this level was serializable and thus correct.

Considering the next level up, we can exploit the compatibility of the loan request tasks and "pushing forward" all the tasks of WF_2 to follow the Enter Decision of WF_1. The effect is that WF_2 and WF_1 become isolated trees, where WF_2 follows WF_1. This schedule is equivalent to the original schedule and is equivalent to the serial execution with WF_1 preceding WF_2. The formal proof for this level-by-level equivalence transformation is based on the proof rules described in [3, 16].

4.2 Semantic Atomicity

Semantic transactions are aborted by invoking additional compensating tasks. We assume that a compensating task is specified for each task, and that there are well-defined rules for deriving the actual input parameters of the compensating tasks from the actual input parameters and the return values of the corresponding "forward" tasks. In the loan request example, the compensating task for the Risk Update is the Risk Update Compensation task.

Compensation achieves semantic atomicity in the following sense. Within a schedule, one task invocation is compensated by a second task invocation if the return values of all subsequent task invocations are the same as if neither had ever executed. This may be further relaxed by tolerating some differences in the return values if these differences are acceptable from an application point of view.

To reason about the interference of compensating tasks and forward tasks in concurrent executions, the model of semantic transactions is extended as follows. Compensating tasks are added to the corresponding transactions, and it is required that the resulting "expanded schedule" be equivalent to a serial schedule of the committed transactions [4, 16, 21]. Such forward and compensating tasks then are executed in the same way as other tasks and their correctness can be guaranteed using the semantic scheduling mechanism.

5 Combining Workflow with Transaction Management

From the characterizations of workflow management and semantic transaction management, it follows that the two directions cover complementary aspects. Therefore, we propose an integrated but modular approach in which workflow management and semantic transaction management each take responsibility for the kinds of dependencies that they can handle best. The semantic transaction manager should remain in charge of the following two classes of dependencies:

- commit/abort dependencies, in the system-oriented sense, and
- interleaving dependencies between the tasks of concurrent workflows.

On the other hand, general control flow and data flow dependencies between tasks as well as general termination dependencies should be taken care of by the workflow manager of the combined model. Recall that termination states in this sense refer to semantic success or failure rather than the system-oriented commit/abort dichotomy.

The rationale for the responsibilities of the semantic transaction manager within the combined model is further discussed in the following.

5.1 Commit/Abort Dependencies

We argue that, in the system-oriented sense, commit/abort dependencies (which may form part of control and termination dependencies) can be handled more efficiently by the semantic transaction management component. For example, consider a control dependency that states: if tasks T_i and T_j both commit, then T_i commits before T_j does. This dependency can easily be enforced by a strongly recoverable transaction manager [5]. Similarly, an abort dependency of the form: if task T_i aborts then T_j must abort before T_i, can be enforced by the semantic transaction manager that produces cascadeless and strict schedules [12]. In fact, a majority of dependencies specifying abort and commit relationships between different tasks can and we argue should be handled by the semantic transaction manager.

Provided that the correspondence of forward steps and compensating steps is made known to the semantic transaction manager, compensating tasks could be scheduled by the semantic recovery mechanism automatically rather than having to interpret some form of compensation dependencies. All necessary information about compensating operations can be recorded on the system log at the time when the corresponding forward operation finishes its execution; thus, there is no need for establishing explicit data flow between a forward step and its compensating step.

In our banking application, for example, if the Loan Decision task of WF_1 in Figure 3 were to return 'rejected', then the workflow manager could simply notify the semantic transaction manager to initiate the necessary undo tasks. This requires that the Risk Update Compensation task be executed which undoes the effects of the Risk Update task. The transaction manager can derive the necessary parameters to execute the compensation task from the system log. It is important to observe that the resulting execution, shown in Figure 3, remains serializable as the Risk Update Compensation and Risk Update tasks are considered compatible (see Figure 2). If this had not been the case, the semantic transaction manager could not have permitted the execution in Figure 3 without possibly leaving the database in an inconsistent state.

5.2 Workflow Interleaving

Another important issue that should be controlled by the semantic transaction management is the treatment of data consistency across tasks in the presence of concurrent workflows. The use of a task compatibility specification to control concurrent executions of workflows is much simpler than attempting to define control flow dependencies to achieve the same effect. For example, this would require control flow dependencies between each pair of loan request workflow executions of the form: the Risk Evaluation of the second workflow can only start if the Risk Update of the first workflow has completed, where the first workflow is the workflow that executed the Risk Evaluation task first. Such dependencies need to be defined between all tasks that could possibly interleave in an incorrect way and would require new dependencies to be added when new workflows are defined. This would result in a large number of dependencies having to be managed. In addition, we argue, that this is more difficult to manage correctly than a compatibility matrix for tasks.

The only workflow model that explicitly addresses this issue is the ConTracts model [18]. In this model, invariant predicates can be specified to hold across tasks, and this allows detecting and counteracting possible interference of concurrent workflows. In full generality, however, this predicate-based approach may pose some problems with respect to both ease of specification and efficient implementation. The

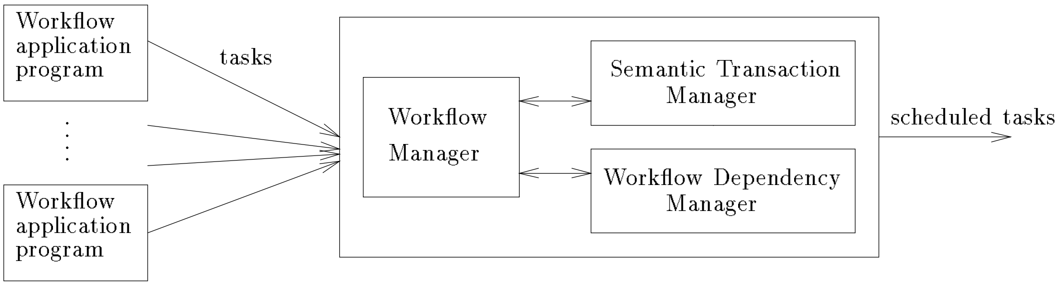

Figure 4: Combined Workflow and Semantic Transaction Scheduler Architecture.

task compatibility approach of the semantic transaction direction, on the other hand, can be handled in a fairly modular way and can be implemented efficiently. One may conceive, however, that compatibility testing could be extended by optionally incorporating pre- and/or post-conditions of operations, thus possibly reconciling the two methods. Another method for controlling interleavings has been proposed in [9].

5.3 Architecture Outline

We propose the following architecture for the integration of workflow and semantic transaction management. A workflow application program is executed by the workflow manager (see Figure 4). The workflow manager is responsible for managing the states of different tasks. It consults the semantic transaction manager and the workflow dependency manager to determine whether the task can enter a specific task state.

The semantic transaction manager checks whether a task can enter a new state without violating any commit/abort or interleaving dependencies. The workflow dependency manager is responsible for ensuring the semantic correctness within each workflow, as specified by the control and data flow, contingency, vitality, and termination dependencies.

If entering a task state violates any dependency, the workflow manager either delays the task execution or aborts the task. If the task execution is delayed, the workflow manager must delay the submission of the task operations until the dependency is enforceable. If the task is aborted then the workflow manager schedules either a compensating task (supplied by the semantic transaction manager) or a contingency task (supplied by the workflow dependency manager).

6 Conclusions

We have examined two major directions in the area of transactional multi-system applications. The first direction, workflow management, views a workflow as a set of independently executed transactions, with various types of execution dependencies between the transactions. The essence of these dependencies is to capture as much behavioral consistency as possible in a declarative manner. However, most workflow models disregard the possible interference of multiple concurrent workflows and can, therefore, not guarantee multi-system data consistency.

The second direction, semantic transaction management, is more conservative in that it views a workflow as a single transaction. Thus, from an application point of view, the guarantees with respect to data consistency are as strong as those of the classical transaction paradigm. The key to achieving acceptable performance without compromising consistency is to exploit the semantics of the application tasks. On the other hand, semantic transaction management does not consider the control flow and data flow between tasks, and no guarantees can be given on the behavioral consistency of large multi-system applications.

Since the two directions cover complementary issues, we propose that appropriate building blocks from workflow management and semantic transaction management be merged into an integrated architecture. The workflow management component of such an architecture would enforce the control and data flow dependencies within a workflow as well as all kinds of semantic termination dependencies. The semantic transaction management component, on the other hand, would manage the compensation dependencies, and the dependencies between incompatible tasks of different workflows to ensure multi-system data consistency.

Acknowledgments: We thank S. Blott and H. Ye of ETH for their contributions in preliminary discussions, and N. Krishnakumar of Bellcore for his helpful comments on the final draft.

References

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

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

[3] C. Beeri, P. A. Bernstein and N. Goodman. A model for concurrency in nested transactions systems. *J. of the ACM*, 36(2), 1989.

[4] C. Beeri, H.-J. Schek and G. Weikum. Multi-level transaction management, theoretical art or practical need? In *LNCS 303*. Springer-Verlag, 1988.

[5] Y. Breitbart and A. Silberschatz. Strong Recoverability in Multidatabase Systems. In *Second International Workshop on RIDE/TQP*, 1992.

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

[7] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao and M.-C. Shan. Third Generation TP Monitors: A Database Challenge. *ACM SIGMOD*, 1993.

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

[9] A. Farrag and M. Özsu. Using semantic knowldege of transactions to increase concurrency. *ACM TODS*, 14(4), 1989.

[10] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, 8(2), 1983.

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

[12] J. Gray. The Transaction Concept: Virtues and Limitations. In *VLDB Conference*, 1981.

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

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

[15] Y. Leu, A. Elmagarmid and N. Boudriga. Specification and Execution of Transactions for Advanced Database Applications. *Information Systems*, 17(2), 1992.

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

[17] A. Sheth and M. Rusinkiewicz. On Transactional Workflows. *IEEE Data Engineering Bulletin*, 16(2), 1993.

[18] H. Wächter and A. Reuter. The ConTract Model. In [8].

[19] G. Weikum, A. Deacon, W. Schaad and H.-J. Schek. Open Nested Transactions in Federated Database Systems. *IEEE Data Engineering Bulletin*, 16(2), 1993.

[20] G. Weikum and H.-J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In [8].

[21] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM TODS*, 16(1), 1991.