Improving Backward Recovery in Workflow Systems

Chengfei Liu† Maria Orlowska‡ Xuemin Lin♯ Xiaofang Zhou‡

† School of Computer and Information Science
University of South Australia, Adelaide, SA 5095, Australia
liu@cs.unisa.edu.au
‡Department of Computer Science and Electrical Engineering
The University of Queensland, Brisbane, QLD 4072, Australia
{*maria,zxf*}*@csee.uq.edu.au*
♯School of Computer Science and Engineering
The University of New South Wales, Sydney, NSW 2052, Australia
lxue@cse.unsw.edu.au

Abstract

The notion of compensation is widely used as means of backward recovery in long-lived transactions as well as business processes supported by workflow management systems. In general, it is non-trivial to design compensating tasks for tasks in the context of a workflow. Actually, a task does not have to be compensatable. In this paper, we first look into the requirements that a compensating task has to satisfy. Then we introduce a new mechanism called confirmation. *With the help of confirmation, we can modify some non-compensatable tasks so that they become compensatable. This greatly improves backward recovery for workflow applications in case of failures. To effectively incorporate confirmation and compensation into the workflow management environment, a three level bottom-up workflow design method is introduced. The implementation issues of this design are also discussed.*

1 Introduction

It is a widely accepted fact that traditional transactions are unsuitable for many applications such as CAD, CASE and automated business processes. The transactions in such environments require to access data held in multiple autonomous database systems for a long duration. To support the long-lived transactions in these environments, it is impossible or unrealistic to go on keeping all the ACID properties [7] supported by traditional transactions. To overcome the limitations of the traditional transaction model, many advanced transaction models have been proposed [1]. Most of them have taken the application semantics into account and provided some semantic mechanisms for programmers. For examples, the Multi-level Transactions [16] allows more concurrency at higher level compared single-level concurrency control. Commutativity of higher level operations can be explored by programmers based on application-specific semantics. Concurrency control for transactions on aggregate attributes has been particularly studied by Reuter [13], Gawlick and Kinkade [4], O'Nell [12]. In their proposed methods, programmers are allowed to make a special request to verify that an attribute bears some relation to a known value. No lock needs to be put on a data item. Similarly, in the NT/PV model [9] and the ConTract model [14], invariants have been used to allow more concurrency.

To guarantee the atomicity of long-lived transactions, *compensating transactions* [6] have been widely used in many advanced transaction models, such as *Sagas* [3], ConTract, Flex [2], Multi-level Transactions and Open-nested Transactions. For a transaction T, a compensating transaction C is a transaction that can *semantically undo* the effects of T after T has been committed. For example, the compensation of a withdrawal can be a deposit. To deal with the problem of long-lived transactions, the Sagas model, for instance, structures a long-lived transaction as a sequence of subtransactions, and each of them is associated with a compensating subtransaction. In case one of the subtransactions in such a sequence aborts the previous subtransactions are undone by automatically scheduling the associated compensating subtransactions. By allowing transactions to release partial results before they complete, we are able to avoid the long-duration waiting problem caused by long-lived transactions.

Reliability is of critical importance to workflow systems [15, 5]. A workflow consists of a set of tasks that are coordinated in order to achieve a common business goal. Each task defines a logical step that contributes towards the completion of the workflow. Failures could occur at various stages within the long lasting life-time of a workflow instance (enactment process). We can classify failures into two separate groups: (1) system failures: failures in the underlying infrastructure (e.g., hardware, network) or failures within the workflow system (e.g., scheduler, databases); (2) semantic failures: failures associated with the execution of workflow tasks (e.g., unavailability of resources, internal decisions). A workflow management system (WfMS) normally deals with system failures by implementing the feature of *forward recovery*. Dealing with semantic failures requires the feature of *backward recovery*, i.e., to eliminate the effects of failed workflow instances. In this paper, we focus on the support of backward recovery.

The notion of compensation is important to workflow systems not only because most workflow instances tend to be long lasting, but also because tasks in a workflow instance may not always be able to undo (e.g., human actions and legacy system processing). One can define compensating tasks which semantically undo the executed tasks of the failed workflow instance [10, 8]. Compensation has been applied to tasks and group of tasks (spheres) to support partial backward recovery in the context of the FlowMark WfMS [11]. Usually, it is assumed a compensating task is associated with a task. However, this assumption is not always true. A task can be non-compensatable if the *forcibility* of the reverse operations of the task cannot be guaranteed by the application semantics. In this paper, we carefully investigate the properties of shared resources and tasks which may be performed on these resources. We find some tasks are non-compensatable because the *reverse* operations of the task may not be always executed successfully. In addition, if the relaxation of isolation on a shared data resource cannot be compromised by a workflow application, the compensation cannot be applied to the tasks which are performed on the resource. As such, we introduce a new mechanism called *confirmation*. By using confirmation, we are able to modify some non-compensatable tasks so that they become compensatable. Once a workflow instance is executed successfully, the confirmation tasks of all executed tasks are executed automatically. This is in contrast to the compensation scenario: Once a workflow instance fails in its execution, the compensating tasks of all executed tasks are executed. In this paper, we focus our presentation in the context of workflows, though the concepts and mechanisms discussed in the paper are also applicable to non-traditional database applications.

The rest of the paper is organized as follows. In section 2, we look into the requirements that a compensating task must satisfy. In section 3, we introduce the mechanism of confirmation and show how it is used to deal with the non-compensatability problem. A bottom-up workflow design method which includes both compensation and confirmation is proposed in section 4, together with some implementation issues. Section 5 concludes the paper.

2 Requirements of a Compensating Task

Suppose a task T is defined in a workflow W, T is called compensatable if the following conditions are satisfied.
(1) *forcibility:* Let C be the compensating task of the task T. Then after T is invoked and executed in any instance WI of W, the execution of C must be guaranteed to be successful within a period of time specified.
(2) *relaxation of isolation:* After T is invoked and executed in any instance WI of W, the shared data resources which T has accessed will be released. This relaxation of isolation on shared data resources is required as the purpose of introducing compensation is to avoid long-duration waiting, otherwise, we should use system level undo instead of compensation.

The following two examples illustrate these two requirements.

Example 1 Suppose a common account is used for effective fanancial management of multiple projects in an organhization. The organisation may have two types of business processes which are specified by two workflows W_1 and W_2, respectively. Instances of W_1 involve a task T_p producing an amount of money and putting it into the common account, while instances of W_2 involve a task T_c consuming funds from the common account.

To model these business processes, a shared data resource called *Common_Account* is used with two operations defined on it: *deposit* and *withdraw*. The tasks T_p and T_c in the business processes can be implemented by invoking the operations deposit and withdraw, respectively. The Common_Account can be described by the following pseudo code. A compensation is associated with the implementation of each operation, it defines the compensating operation of the operation, if needed. Consequently, the compensating tasks of tasks T_p and T_c can invoke the compensation parts of the deposit and withdraw operations, respectively.

```
Common_Account {
    double balance;
    /* operations on the account
    boolean withdraw(double amount);
    void deposit(double amount);
```

```
}

boolean withdraw(double amount) {
    if (balance - amount >= 0) {
        balance := balance - amount;
        return(true)
    }
    else return(false);
Compensation:
    deposit(amount);
}

void deposit(double amount) {
    balance := balance + amount;
Compensation:
    /* not available
}
```

As mentioned in Section 1, concurrency control of aggregate attributes (*balance* in this example) has been well addressed in Reuter's method, Fast Path method and Escrow method. These methods focus on the *forward* behaviour of transactions. In this paper, we study the *backward* behaviour of transactions (workflows), i.e., the compensatability of tasks which access aggregate attributes. For a private account, deposit is always compensatable by withdrawal (and vice versa). However, for the common account as defined in this example, the compensation of the deposit operation is not available. This is because the forcibility of its reverse operation withdraw is not guaranteed by the application. Consequently, the compensating task of T_p is not available either. The following scenario illustrates this.

Let WI_1 be an instance of W_1 and WI_2 be an instance of W_2. Initially, the balance of Common_Account is 0. First, the task T_p of WI_1 is executed which deposits $1,000 to the Common_Account. After that, the task T_c in WI_2 withdraws $800 from the Common_Account. In a later stage, WI_1 fails due to some reason and tries to rollback. This naturally includes withdrawing $1,000 back which it previously deposited into the account. Unfortunately, this withdrawal is unable to execute successfully since part of the amount of money has been consumed by WI_2 and it is possible the execution of WI_2 has already been finished.

There are two system-level solutions to this problem:
(1). The T_p of WI_1 holds the lock of the Common_Account until all tasks of WI_1 finishes.
(2). The T_p of WI_1 releases the Common_Account after it is executed. However, WI_2 must wait for WI_1 to successfully finish. If WI_1 fails and the compensation of T_p can not be executed successfully, WI_2 may need to be cascadedly rollbacked. This means much work done by WI_2 may be lost.

Obviously, both solutions are not applicable since long-duration waiting is unavoidable even when the balance of Common_Account is ample.

In real situation, one may use an approximate approach based on statistics or experienced estimation of, say, the percentage of failed instances. In that way, T_p of most failed instances of W_1 can be compensated. However, there is no guarantee that T_p of all failed instances of W_1 can be compensated, especially if the estimation is over-optimistic. In this case, the organisation may have some policies for *exceptional* compensation. Can we provide guaranteed compensation? We will answer this in the next section.

Example 2 In many service organisations, there may exist one type of business processes which include a task collecting customer information and other types of business processes which include tasks using customer information. Due to the variety of applications, the use of the customer information might be different. Let us first look at a *dirty-read* case where a business process does not have to access accurate information about customers.

To model the business processes in this example, we may have two workflows W_1 and W_2, where W_1 includes a task T_i for inserting customer information while W_2 includes a task T_d which dirty-reads customer information. A shared data resource called *Customer_Info* is needed with two operations *insert* and *dirty_read* for T_i and T_d to invoke, respectively. The following is the definition of Customer_Info.

```
Customer_Info {
    table customer;
    /* operations on Customer_Info
    void insert(tuple cust);
    table dirty_read(string pred);
}

void insert(tuple cust) {
    insert tuple cust to the table customer;
Compensation:
    delete tuple cust from table customer
    using cust.name;
}

table dirty_read(string pred) {
    return("select * from customer where pred");
Compensation:
    /* do nothing
}
```

As seen above, in this dirty-read case, the operation *insert* is compensatable with reverse operation defined which deletes what has been inserted. This is because there is no isolation requirement on the shared data resource Customer_Info. After T_i of an instance, say WI_1 of W_1, inserts a customer tuple into customer table, the table with the new inserted customer tuple (partial result of WI_1) is immediately accessible for T_d of any instance of W_2, regardless whether WI_1 may fail later and thus the inserted customer information may be deleted.

Suppose now the service organisation needs to add a new business process specified by W_3 which needs to *strict-read* Customer_Info via a task T_s. In this case, the operation *insert* defined above is no longer compensatable. This is because the isolation on Customer_Info can no longer be compromised. After T_i of WI_1 inserts a customer tuple, that tuple can not be immediately revealed to outside in case T_s of any instance of WS_3 accesses it. As a result, the compensating task of T_i is no longer available. Even locking (in long-duration) on the customer table cannot be applied as it restricts the use of dirty-read. To support this mixed dirty-read and strict-read scenario by locking, an explicit and sophisticated record-level locking feature must be supported. Unfortunately, this feature is not easy to find in current SQL-based DBMSs.

3 Confirmation

In this section, after analysing the requirements of compensatable tasks, we introduce a new concept called *confirmation* and show how it can be used to cope with the non-compensatability problem. As seen from the above examples, a task can be implemented by invoking a set of operations. Similarly, the compensating task of the task can be implemented by invoking the compensation parts of the set of operations. If a task is compensatable, all operations it may invoke must be compensatable. In the following, we focus on the discussion of the compensatability at the operation level.

3.1 Coping with Non-forcibility

As demonstrated by Example 1, if an operation is compensatable, its reverse operation must be forcible. There are some non-compensatable operations whose reverse operations are absolutely non-forcible. An often-mentioned example is emitting a missile. If a workflow instance contains a task which invokes this kind of non-compensatable operations, the only solutions are either delaying the task to a later stage, or ignoring/manually adjusting the effects of the operation if the workflow instance fails. However, for most non-compensatable operations, their reverse operations are not forcible only under certain conditions. i.e., the reverse operation of a non-compensatable operation cannot be executed successfully only when an *undesired condition* is reached. For example, the compensation of the operation invocation $deposit(\$1,000)$ in Example 1 fails only if the balance decreases to less than $1,000. If the original balance is no less than $800, the execution of the compensation will not encounter a problem. Therefore, if the organisation has a sufficient balance in the common account for most of the time, the undesired condition will not be easily reached. System level locking is a simple way to deal with this non-compensatability problem, but obviously it suffers two severe problems: (1). long duration locking of the data resource until the invoking workflow instances complete successfully. (2). unnecessary locking since update of the data resource will not cause any problem in most cases.

Based on the discussion, it is ideal to provide a semantic level mechanism which can be used to prevent the undesired condition from being satisfied. For this purpose, we propose a new mechanism called *confirmation*. Informally, a confirmation of an operation is a separated part of the operation. The execution of this part is not executed at the time the operation is executed. Instead, this part is executed at a later time for the purpose of confirming the execution of the operation. The motivations for introducing the confirmation mechanism are two fold: (1) to isolate the part of the operation which may affect the compensatability of the operation and execute this part later; (2) to semantically commit the operation at a safe time. Similar to a compensating task, the confirmation task of a task can be implemented by invoking the confirmation parts of the set of operations which have been invoked during the execution of the task. The confirmation parts of all invoked operations in a workflow instance are executed automatically once the system gets the instruction for confirmation.

More precisely, let O_{cf} and O_{cp} the confirmation part and the compensation part of an operation O, respectively. Suppose the confirmation part and compensation part are defined for each operation with the default definition "doing noting". Then after O is executed, two possible situations will happen later. (1) If the invoking workflow instance executes successfully, O_{cf} will be automatically executed later to semantically commit O; (2) If the invoking workflow instance fails, O_{cp} will be automatically executed later to semantically rollback O.

To ensure that the undesired condition will never be reached, we can put the *unsafe* part of an operation (e.g., deposit) into its confirmation part and delay the execution of this part until a safe time later on, say, after an invoking workflow instance succeeds in its execution. At that time, changing the value of the undesired condition by other operations (e.g., withdraw) will not cause any problem because the compensation is no longer needed for this workflow instance. As a result, an operation can always be compensated before the execution of the confirmation part of the operation. In addition, both an operation and its confirmation part can be implemented as two separate short transactions.

Therefore, the shared resources that they may access only need to be locked in a short time.

Note, O and O_{cf} are *forward* parts while O_{cp} is a *backward* part. If the forcibility of forward parts cannot be guaranteed, it will not leave any problem as the invoking workflow instance can always choose to fail or try a contingency plan.

Let us look at how confirmation can help our first example.

Example 3 A modification of Example 1 with confirmation.

```
Common_Account {
    double balance;
    double available_balance;
    /* operations on the account
    boolean withdraw(double amount);
    void deposit(double amount);
}

boolean withdraw(double amount) {
    if (available_balance - amount >= 0) {
      available_balance := available_balance - amount;
      balance := balance - amount;
      return(true);
    }
    else return(false);
Compensation:
    balance := balance + amount;
    available_balance := available_balance + amount;
Confirmation:
    /* do nothing
}

void deposit(double amount) {
    balance := balance + amount;
Compensation:
    balance := balance - amount;
Confirmation:
    available_balance := available_balance + amount;
}
```

As shown above, a new attribute *available_balance* is added to indicate the available balance of the account. A confirmation part is added to the deposit operation for increasing available balance. A workflow instance which invokes a deposit operation can hold its deposited amount of money by delaying the execution of the confirmation part of the operation later, say, until the workflow instance succeeds later in its execution. By doing so, the deposit operation becomes compensatable by the compensation part of the operation, i.e., balance decrement.

Come back to the scenario in Example 1. If the original balance and available balance are all zero, after $deposit(\$1,000)$ is invoked by WI_1, the balance is increased to $1,000. The available balance, however, remains to be zero. Both balance and available balance can be accessed by other workflows for whatever purposes. Before the confirmation part of the operation invocation $deposit(\$1,000)$ is executed, $withdraw(\$800)$ invoked by WI_2 cannot be successfully executed. This guarantees that $deposit(\$1,000)$ invoked by WI_1 is compensatable. If the original available balance is no less than $800 or is increased to no less than $800 (say, after the confirmation of the invocation $deposit(\$1,000)$), there is no problem for WI_2 to successfully invoke $withdraw(\$800)$. This reflects precisely the semantics of the application.

The confirmation mechanism used in this example is different from a concurrency control method such as the *invariant* used by the NT/PV model and the ConTract model. The former is used for guaranteeing successful backward recovery, while the latter is used for increasing concurrency.

3.2 Coping with Isolation

As shown in Example 2, if an operation performed on a shared data resource is compensatable, the isolation requirement on the data resource must be compromised. Usually in a workflow instance, the compensation of an operation is invoked at a later time after the invocation of the operation. If the isolation on the data resource is required, other workflow instances have to wait until the invoking workflow instance finishes. In that case, there is no need to provide compensation at all. However, with the help of confirmation, we can make an operation compensatable while still keeping the isolation requirement on the shared data resources. This can be done by temporarily separating a data resource into an isolation part and a non-isolation part.

Let us look how it works for our isolation example.

Example 4 A modification of Example 2 with confirmation.

```
Customer_Info {
    table customer, temp_cust;
    /* operations on Customer_Info
    void insert(tuple cust);
    table dirty_read(string pred);
    table strict_read(string pred);
}

void insert(tuple cust) {
    insert tuple cust into table temp_cust;
Compensation:
    delete tuple cust from table temp_cust
    using cust.name;
```

Confirmation:
    insert tuple cust into table customer;
    delete tuple cust from table temp_cust
    using cust.name;
}

table dirty_read(string pred) {
    table temp1, temp2;
    select ∗ into temp1 from customer where pred;
    select ∗ into temp2 from temp_cust where pred;
    return(temp1 union temp2);
Compensation:
    /∗ do nothing
Confirmation:
    /∗ do nothing
}

table strict_read(string pred) {
    return("select ∗ from customer where pred");
Compensation:
    /∗ do nothing
Confirmation:
    /∗ do nothing
}

In this example, we use the table *customer* and *temp_cust* to store the non-isolation and isolation parts of *Customer_Info*, respectively. When an *insert* operation is invoked, new customer information is put into *temp_cust*. When the invoking workflow instance finishes successfully, the confirmation part of the operation is executed to confirm the insert operation invocation by swapping the customer information from the table *temp_cust* to the table *customer*. If the invoking workflow instance fails, the invoked insert operation can be easily compensated by removing the customer information from the table *temp_cust*, without affecting other workflow instances which are concurrently accessing the customer information.

With the help of confirmation, long-duration locking can be avoided but isolation on the data resources can still be achieved. For invocations of operations such as *strict_read* where isolation is required, only non-isolation part of customer information is made available for accessing; For invocations of operations such as *dirty_read* where isolation can be compromised, both non-isolation and isolation parts of the resource can be accessed. No interference will occur among workflow instances regardless whether isolation on the data resources is required. As a result, the availability of data resources is maximised. This is ideal for enterprises where a variety of requirements on data resources may exist. However, without the help of confirmation, it is almost impossible to effectively implement the above mixed strict_read and dirty_read scenario where both compensatability and isolation on shared data resources are required.

4 Bottom-Up Workflow Design

In this section we propose a three level bottom-up workflow design method which can easily and perfectly incorporate both compensation and confirmation into a workflow management environment. At the bottom level, data resources are modelled as objects. The only interface to a data resource is via a set of operations together with their compensations and confirmations. This is helpful in workflow environments. For instance, a legacy system can be wrapped as an object with an interface providing a set of *big* operations. Compensation and confirmation can be developed at the time a legacy system is involved in some workflows. The middle level is used to specify tasks. A task can be implemented simply by invoking operations on data resources. The top level is used to specify workflows. As usual, dependencies among tasks of workflows are specified. To support confirmation and compensation, extra control are added at this level. Partial backward recovery can be easily realised by multiple use of confirmation control. In the following, we present workflow design via these three levels first, then discuss briefly the run-time support of workflows designed in such a way.

4.1 Specifying a Workflow

Basically, a workflow is about the coordination of a set of tasks. This is achieved by defining various types of dependencies among tasks, e.g., control flows, data flows, temporal constraints, etc. Usually, a workflow specification language is provided by a WfMS to specify these dependencies. In this paper, we concentrate on how compensation and confirmation can be incorporated into the workflow specification. In supporting compensation and confirmation, we design to add two statements called COMPENSATE and CONFIRM. Specifiers should be allowed to put these statements into the workflow specification to reflect their decisions. This is similar to including ROLLBACK and COMMIT statements in a transaction. The difference between a workflow scenario and a transaction scenario is that execution of COMPENSATE and CONFIRM statements is an application behaviour, while execution of ROLLBACK and COMMIT statements is a system behaviour. We may give another pair of names SEMANTIC-ROLLBACK/SEMANTIC-COMMIT to represent COMPENSATE/CONFIRMATION.

By putting a CONFIRM statement carefully at several places in a workflow, we are able to confirm the executed tasks group by group, thus confirm the execution of the workflow instances step by step. We may call such a group of tasks as *a sphere of joint confirmation* with the similarity to the term *a sphere of joint compensation* discussed in [11]. In most cases, these two spheres can be combined as a single concept. As a result, a workflow instance can be partially confirmed or partially compensated in the unit of a sphere of compensation/confirmation. Once a workflow instance confirms the execution of a group of tasks at a point and fails its execution later, the system can apply partial recovery by compensating those tasks which are executed after that point.

4.2 Specifying a Task

A task specification is mainly concerned with the implementation of the task. When a task needs to access a data resource, it is implemented by invoking an operation defined at the interface of the data resource. A task may invoke multiple operations defined on different data resources. For each task, a compensating task and a confirmation task are automatically defined by the compensation parts and confirmation parts of all operations the task may access. This will be discussed next.

4.3 Specifying a Data Resource

For each data resource, an interface is provided which consists of a set of operations. Tasks using a data resource of this type can only invoke these operations. Beside the operation itself (which we will call it as the *normal part* of the operation in the following discussion), a *compensation part* and a *confirmation part* of the operation must be defined, with the default definition as "doing nothing".
(1). A *normal part* specifies what needs to be executed when the operation is invoked by a task.
(2). A *compensation part* specifies what needs to be executed to eliminate the effect of the normal part invoked previously by a task T. The compensation part is invoked when the compensating task of the task T is executed.
(3). A *confirmation part* specifies what needs to be executed to confirm the work done by the normal part invoked previously by a task T. The confirmation part is invoked when the confirmation task of the task T is executed.

The specifications for shared data resources *Common_Account* and *Customer_Info* have been given in Example 3 and Example 4, respectively.

4.4 Executing a Workflow Instance

When an instance of an above-specified workflow is submitted to the workflow engine of a WfMS for execution, the engine will schedule a compensation process automatically while a COMPENSATE statement is being executed. Similarly, the engine will schedule a confirmation process automatically while a CONFIRM statement is being executed. This can happen as well when an external event triggers the engine requiring COMPENSATE/CONFIRM the workflow instance. When a COMPENSATE request arrives, the engine schedules the execution of all compensating tasks of those tasks which have been executed yet have not been confirmed. This in turn triggers the execution of compensation parts of all operations which have been invoked by the above tasks. The latest point of the group of tasks confirmed is recorded by the system. This point is used as a guide to where the backward recovery should stop. Compensating tasks are executed in reverse order (backward).

Similarly, when a CONFIRM request arrives, the engine schedules the execution of all confirmation tasks which have been executed yet have not been confirmed. This in turn triggers the execution of confirmation parts of all operations which have been invoked by the above tasks. The latest point is also used as a guide to where the confirmation process should start. Confirmation tasks are executed in the same order as their tasks (forward).

During the process of compensation or confirmation, the values of input parameters of compensation part or confirmation part of each invoked operation are provided automatically. This can be done by appropriate computation after the completion of the normal part of each invoked operation and saving the results in the system log.

5 Concluding Remarks

Designing compensating tasks is critical for supporting backward recovery in workflow systems and non-traditional database applications. Due to the semantics of applications and their shared data resources, a compensating task does not always exist for a task. In this paper, we studied the requirements of a compensatable task. Based on our observations, we proposed a novel semantic level mechanism called *confirmation*. The relationship between confirmation and compensation is similar to that between a commit and a rollback. By using confirmation properly, non-compensatable operations on the shared data resources can be rewritten and become compensatable. A three level workflow design framework was presented together with the discussion of its run-time support.

Like a compensation, a confirmation is a semantic mechanism provided to workflow specifiers. Workflow specifiers may use it in a flexible way, based on the requirements of applications. Multiple versions of a confirmation and a compensation may be provided based on certain factors such as time. It is also interesting to build different patterns of compensation and confirmation according to some typical applications. We will investigate these in the future.

References

[1] A. Elmagarmid, editor. *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, 1992.

[2] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, August 1990.

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

[4] D. Gawlick and D. Kinkade. Varieties of concurrency control in ims/vs fast path. *Bull. IEEE Database Eng.*, 8(2):3–10, 1985.

[5] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3:119–153, 1995.

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

[7] T. Harder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[8] B. Kiepuszewski, R. Muhlberger, and M. Orlowska. Flowback: Providing backward recovery for workflow systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 555–557, 1998.

[9] H. Korth and G. Speegle. Long-duration transactions in software design projects. In *Proceedings of the 6th International Conference on Data Engineering*, pages 568–574, 1990.

[10] D. Kuo, M. Lawley, C. Liu, and M. Orlowska. A model for transactional workflows. In R. Topor, editor, *Seventh Australasian Database Conference Proceedings*, volume 18, pages 139–146, Melbourne, Australia, 1996. Australian Computer Science Communications.

[11] F. Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *Proceedings of BTW'95*, pages 51–70, 1995.

[12] P. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[13] A. Reuter. Concurrency on high-traffic data elements. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 83–92, 1982.

[14] A. Reuter. Contracts: A means for extending control beyond transaction boundaries. In *Proceedings of the 3rd International Workshop on High Performance Transaction Systems*, 1989.

[15] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1994.

[16] G. Weikum and H-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.