Ensuring Task Dependencies During Workflow Recovery

Indrakshi Ray, Tai Xin, and Yajie Zhu

Department of Computer Science
Colorado State University
{iray,xin,zhuy}@cs.colostate.edu

Abstract. Workflow management systems (WFMS) coordinate execution of multiple tasks performed by different entities within an organization. In order to coordinate the execution of the various tasks in a workflow, task dependencies are specified among them. These task dependencies are enforced during the normal execution of the workflow. When a system crash occurs some tasks of the workflow may be committed, some may be partially executed and others unscheduled. In such a situation, the recovery mechanism must take appropriate actions to react to the failure. Although researchers have worked on the problem of workflow recovery, most of these work focus on restoring consistency by removing the effects of partially executed tasks. However, these work fail to address how to ensure task dependencies during workflow recovery. In this paper, we consider the workflow recovery problem and propose a recovery scheme that ensures the satisfaction of dependencies in a workflow and restores consistency as well.

1 Introduction

Workfbw management systems (WFMS) are responsible for coordinating the execution of multiple tasks performed by different entities within an organization. A group of such tasks that form a logical unit of work constitutes a workfbw. To ensure the proper coordination of these tasks, various kinds of dependencies are specified between the tasks of a workfbw. The scheduler of a workfbw is expected to enforce these dependencies during normal execution of a workfbw. In addition, we must also ensure that these dependencies are enforced in the event of a failure. In this paper we focus on how to ensure dependencies when system crashes or failures occur.

A large body of research exists in the area of workfbws [1, 5, 7, 9, 10]. Researchers [1, 2, 7] have focussed on how to specify workfbws, how to ensure the correctness of the specification, and how to control the execution of tasks in a workfbw in order to satisfy the dependencies. Workfbw recovery, however, has received very little attention. Most of the research [5, 10, 9] in workfbw recovery focusses on how to restore a consistent state after a failure and they do not address the issue of enforcing task dependencies. In this paper we provide an automated approach to workfbw recovery that not only restores consistency but also ensures the satisfaction of task dependencies.

Before describing our approach, let us illustrate the problem that occurs if task dependencies are not enforced during recovery. Consider a workfbw W_k consisting of two tasks: reserving a room in a resort (t_{ki}) and renting a car (t_{kj}). These two tasks

t_{ki} and t_{kj} have a dependency between them: if t_{ki} begins then t_{kj} should begin. The scheduler enforces this dependency by starting t_{kj} after t_{ki} begins. Suppose t_{kj} gets completed before t_{ki} and a crash occurs before t_{ki} completes. Since t_{kj} is completed and t_{ki} is incomplete, the recovery algorithm to restore consistency undoes only t_{ki}. This results in an undesirable situation. Thus, restoring consistency by removing the effects of partially executed tasks is not enough for workflow recovery: the recovery algorithms must also take into account the effect of task dependencies.

In this paper we elaborate on the different kinds of task dependencies present in a workflow and show how these task dependencies impact the recovery process. We discuss what information is needed and how this information is stored to perform an effective recovery. Finally, we give a recovery algorithm that restores consistency and enforces task dependencies.

The rest of the paper is organized as follows. Section 2 discusses some related work in this area. Section 3 describes our workflow model and enumerates the different kinds of dependencies. Section 4 identifies the information necessary for workflow recovery. Section 5 presents our workflow recovery algorithm. Section 6 concludes the paper with some pointers to future directions.

2 Related Work

Although a lot of research appears in workflow, we focus our attention to those discussing workflow dependencies and workflow recovery. An approach for specifying and enforcing task dependencies have been proposed by Attie et al. [2]. Each task is described by a set of events, such as, start, commit and rollback. Dependencies between the tasks connect the events of various tasks and specify a temporal order among them. These dependencies are specified using Computation Tree Logic (CTL). With respect to recovery, the authors mention what data is needed to recover from failures but do not provide details. They also do not address how task dependencies can be enforced during recovery.

Eder and Liebhart [5] classify workflow as document-oriented workflow and process-oriented workflow, and identify potential different types of failure. This paper proposes different recovery concepts, such as forward recovery and backward recovery, and enumerates how the workflow recovery manager can support these concepts. But no detailed approach for workflow recovery process is provided.

Failure handling and coordinated execution of workflows was also proposed by Kamath and Ramamritham [9]. The approach describes how to specify different options that can be taken in the event of a failure. The workflow designer can then choose from these options and customize the recovery process to ensure correctness and performance. Thus, a workflow designer needs to have a comprehensive knowledge about both the business process and the workflow model in order to perform the recovery.

3 Model

We begin by extending the definition of a workflow given in the Workflow Reference Model [8].

Definition 1
[Workflow] A *workflow* W_i is a set of tasks $\{ t_{i1}, t_{i2}, \ldots, t_{in} \}$ with dependencies specified among them that achieve some business objective.

Next, we define what a task is. We assume that each task in a workfbw is a transaction as per the standard transaction processing model [3].

Definition 2
[Task] A task t_{ij} performs a logical unit of work. It consists of a set of data operations (read and write) and task primitives (begin, abort and commit). The read and write operations that task t_{ij} performs on data item x are denoted by $r_{ij}[x]$ and $w_{ij}[x]$ respectively. The begin, abort and commit operations of task t_{ij} are denoted by b_{ij}, a_{ij} and c_{ij} respectively.

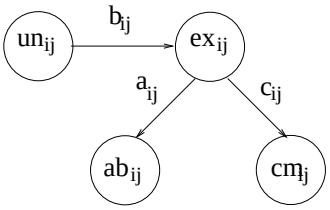

Fig. 1. States of Task t_{ij}

A task t_{ij} can be in any of the following states: *unschedule* (un_{ij}), *execute* (ex_{ij}), *commit* (cm_{ij}) and *abort* (ab_{ij}). Execution of task primitives causes a task to change its state. Executing the begin primitive (b_{ij}) causes a task to move from unschedule state (un_{ij}) to execute state (ex_{ij}). Executing commit primitive (c_{ij}) causes the task to move from execute state (ex_{ij}) to commit state (cm_{ij}). Executing abort primitive (a_{ij}) causes the task to move from execute state (ex_{ij}) to abort state (ab_{ij}). This is illustrated in Figure 1.

To properly coordinate the execution of tasks in a workfbw, dependencies are specified between the task primitives of a workfbw. Task dependencies are of different kinds. Some researchers classify the dependencies as *control flow dependencies*, *data flow dependencies* and *external dependencies*. In this paper, we focus only on control fbw dependencies.

Definition 3
[Control flow Dependency] A control fbw dependency specifies how the execution of primitives of task t_i causes the execution of the primitives of task t_j.

The common types of control fbw dependencies found in workfbw are enumerated below. For a complete list of the different kinds of dependencies, we refer the interested reader to the work by Chrysanthis [4].

1. *Commit Dependency*: A transaction t_j must commit if t_i commits. We can represent it as $t_i \rightarrow_c t_j$.
2. *Abort Dependency*: A transaction t_j must abort if t_i aborts. We can represent it as $t_i \rightarrow_a t_j$.
3. *Begin Dependency*: A transaction t_j cannot begin until t_i has begun. We can represent it as $t_i \rightarrow_b t_j$.
4. *Begin-on-commit Dependency*: A transaction t_j cannot begin until t_i commits. We can represent it as $t_i \rightarrow_{bc} t_j$.
5. *Force Begin-on-commit Dependency*: A transaction t_j must begin if t_i commits. We can represent it as $t_i \rightarrow_{fbc} t_j$.
6. *Force Begin-on-begin Dependency*: A transaction t_j must begin if t_i begins. We can represent it as $t_i \rightarrow_{fbb} t_j$.
7. *Force Begin-on-abort Dependency*: A transaction t_j must begin if t_i aborts. We can represent it as $t_i \rightarrow_{fba} t_j$.
8. *Exclusion Dependency*: A transaction t_i must commit if t_j aborts, or vice versa. We can represent it as $t_i \rightarrow_e t_j$.

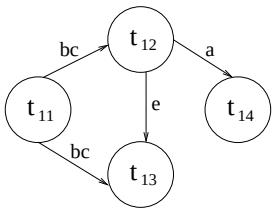

Fig. 2. Tasks and Dependencies in Example Workflow

Example 1. Consider a workfbw W_1 consisting of the following set of tasks $\{t_{11}, t_{12}, t_{13}, t_{14}\}$ where each task is described below.

Task t_{11} Reserve a ticket on Airlines A
Task t_{12} Purchasing the Airlines A ticket
Task t_{13} Canceling the reservation
Task t_{14} Reserving a room in Resort C

The various kinds of dependencies that exists among the tasks are shown in Figure 2. There is a begin on commit dependency between t_{11} and t_{12} and also between t_{11} and t_{13}. This means that neither t_{12} or t_{13} can start before t_{11} has committed. There is an exclusion dependency between t_{12} and t_{13}. This means that either t_{12} can commit or t_{13} can commit but not both. Finally, there is an abort dependency between t_{14} and t_{12}. This means that if t_{12} aborts then t_{14} must abort.

4 Information Needed by the Recovery Algorithm

Workfbw recovery is much more complex than the recovery in traditional transaction processing system. Consider Example 1. Suppose a crash occurs while t_{12} is executing and after t_{11} and t_{14} have committed. The traditional recovery mechanism will consult the log and undo t_{12}, but it will not have enough information to perform the recovery such that the dependency $t_{12} \rightarrow_a t_{14}$ is satisfied.

The recovery process will need to know the state of the workfbw when it crashed and it will also need to know the actions needed to perform the recovery. In our algorithm, the log records store the information about the state of the workfbw. The workfbw schema stores the information about the actions needed for workfbw recovery. We describe these below.

4.1 Workflow Schema

In any organization there are a finite number of types of workfbw. We refer to these as workfbw schemas. A *workflow schema* defines the type of a workfbw. A workfbw schema is specified by (1) the types of inputs and outputs needed by workfbws instances satisfying this schema, (2) the specification of the types of tasks, and (3) the dependencies between these tasks.

Each workfbw is actually an instance of a workfbw schema. We denote the schema associated with workfbw W_i as WS_i. We denote the type of task t_{ij} as $ty(t_{ij})$. Each type of task is specified by (1) the types of inputs and outputs, if any, needed by tasks of this type, (2) recovery attributes for this type of tasks, (3) compensation-fbw for this type of tasks, and (4) alternate-execution-fbw for this type of tasks.

Each type of task is described by attributes. Those attributes that are needed by the recovery process are known as the recovery attributes. Recovery attributes can be *compensatable*, *re-executable* and *alternate executable*. Compensatable identifies whether tasks of this type can be compensated or not. Re-executable indicates whether the tasks of this type can be re-executed or not. Alternate executable signifies whether tasks of this type can be substituted by alternate tasks or not.

For any type of task $ty(t_{ij})$ that is compensatable, the *compensation-flow* includes the set of types of tasks $\{ty(t_{ik}), ty(t_{il}), \ldots, ty(t_{in})\}$ that can compensate this type $ty(t_{ij})$, the task dependencies between them, and the inputs to the compensation process. For any type of task $ty(t_{ij})$ that is alternate executable, the *alternate flow* includes the set of types of tasks $\{ty(t_{ik}), ty(t_{il}), \ldots, ty(t_{in})\}$ that can substitute this type $ty(t_{ij})$, the task dependencies between them, and the inputs to the alternate execution process.

The information about all the workfbw schemas is maintained in stable storage.

4.2 Workflow Log Records

In order to recover from a workfbw system failure, the state information of a workfbw need to be logged onto some stable storage. We propose that such information be stored in the system log. Execution of a workfbw primitive, a task primitive, or a task operation results in the insertion of log a record. Moreover, the logs also contain records associated with checkpointing.

Execution of a begin primitive in a workflow results in the insertion of the following log record. $< START\ W_i, WS_i >$ where W_i indicates the workflow id and WS_i indicates the schema id that corresponds to the workflow W_i. The completion of the workflow is indicated by a log record $< COMPLETE\ W_i >$.

Execution of the primitive begin for task t_{ij} results in the following records being inserted in the log: $< START\ t_{ij} >$ where t_{ij} is the task id. Note that the task id includes information about the workflow id of which the task is a part. Similarly, execution of the primitives commit or abort result in the following log records for task t_{ij}: $< COMMIT\ t_{ij} >$ or $< ABORT\ t_{ij} >$. Execution of operations also cause log records to be inserted. A write operation causes the log record $< t_{ij}\ X, v, w >$ where X is the data item written by the workflow and v and w indicate the old value and the new value of the data item written by task t_{ij}. The inputs to the task and the outputs produced by the task are also recorded as log records.

Checkpointing Our checkpointing technique, based on the nonquiescent checkpointing performed on an undo/redo log in the standard transactional model [6], involves the following steps.

1. Write a log record $< START\ CKPT(t_{wi}, t_{yj}, \ldots, t_{zk}) >$ and flush the log. The tasks t_{wi}, t_{yj}, ..., t_{zk} denote all the tasks that have started but not yet committed or aborted when the checkpointing started.
2. Write to disk all dirty buffers, containing one or more changed data item.
3. Append an $< END\ CKPT >$ record to the log and flush the log.

5 Recovery Algorithm

We assume that a system crash can take place at any time. A crash may occur after a workflow is completed or during its execution. When a crash occurs, the task of a workflow may be in unschedule, commit, abort, or execute state (please refer to Figure 1). If a crash occurs when the task is in the execute state, then we say that the task has been *interrupted*.

Our recovery algorithm proceeds in three phases. The first phase is known as the *undo* phase. In this phase, the log is scanned backwards to identify tasks that have been committed, aborted or interrupted. The aborted and interrupted tasks are undone in this phase by restoring the before image values of updated data items. After an interrupted task t_{ij} has been undone, we write a $< RECOVERY\ ABORT\ t_{ij} >$ log record and flush this to the disk. The purpose of this log record is to indicate that this task t_{ij} has been undone, but the dependencies associated with the tasks have not been dealt with.

The second phase is the *redo* phase. In this phase the the data items updated by committed tasks are set to the modified values. The third phase is the *re-execution and dependency adjustment* phase. In this phase, we start from the very first workflow that was interrupted. From the schema, we identify all the tasks of the workflow that have been committed, aborted, interrupted and unscheduled. Starting from the first task that was interrupted, we try to re-execute this task. If the task can be re-executed, then we do not have to worry about the dependency implications of this task. Otherwise, we have

to take care of the dependencies of this task. This, in turn, may require some other tasks to be aborted or re-scheduled. The dependencies of these tasks must now be considered. We continue this process until all the dependencies have been considered. The tasks to be scheduled are inserted into a list called *toschedule* that is submitted to the scheduler. Finally, we write an $< ABORT >$ log record for all the tasks that were interrupted.

Algorithm 1
Workfbw Recovery Algorithm
Input: the log, file containing workfbw schemas
Output: a consistent workfbw state in which all the task dependencies are enforced

Initialization:
/* lists keeping track of incomplete workfbws and tasks in them */
$completeWF = \{\}$, $incompleteWF = \{\}$
$globalCommitted = \{\}$, $globalInterrupted = \{\}$, $globalAborted = \{\}$
/* lists keeping track of tasks in each workfbw */
$committed = \{\}$, $interrupted = \{\}$, $unscheduled = \{\}$
$adjustSet = \{\}$, $tempAborted = \{\}$, $toschedule = \{\}$
/* the endCkptFlag set to false */ $endCkptFlag = 0$
Phase 1 Undo Phase
begin
do /* Scan backwards until the $< START\ CKPT >$ */
get last unscanned log record
case the log record is $< COMMIT\ t_{wi} >$
$globalCommitted = globalCommitted \cup \{t_{wi}\}$
case the log record is $< ABORT\ t_{wi} >$
$globalAborted = globalAborted \cup \{t_{wi}\}$
case the log record is $< RECOVERY\ ABORT\ t_{wi} >$
if $t_{wi} \notin globalCommitted \cup globalAborted$
$globalInterrupted = globalInterrupted \cup \{t_{wi}\}$
case the log record is update record $< t_{wi}, x, v, w >$
if $t_{wi} \notin globalCommitted$
change the value of x to v /* restores before image of x */
if $t_{wi} \notin globalCommitted \cup globalAborted$
$globalInterrupted = globalInterrupted \cup \{t_{wi}\}$
case the log record is $< START\ t_{wi} >$
if $t_{wi} \in globalInterrupted$
write a $< RECOVERY\ ABORT\ t_{wi} >$ record and flush to the disk
case the log record is $< COMPLETE\ W_k >$
$completeWF = completeWF \cup \{W_k\}$
case the log record is $< START\ W_k >$
if $W_k \notin completeWF$
$incompleteWF = incompleteWF \cup \{W_k\}$
case the log record is $< END\ CKPT >$
$endCkptFlag = true$ /* we have found an end ckpt record */

until log record is $< START\ CKPT(..) >$ AND $endCkptFlag = true$
/* Find the incomplete tasks in the $< START\ CKPT(...) >$ record
for each task t_{ij} in $< START\ CKPT(...) >$ record
if $t_{ij} \notin globalCommitted \cup globalAborted$
$globalInterrupted = globalInterrupted \cup t_{ij}$
/* Scan backward to undo the aborted and incomplete tasks */
do
case the log record is update record $< t_{wi}, x, v, w >$
if $t_{wi} \notin globalCommitted$
change the value of x to v /* restores before image of x */
case the log record is $< START\ t_{wi} >$
if $t_{wi} \in globalInterrupted$
write a $< RECOVERY\ ABORT\ t_{wi} >$ record and flush to the disk
until all $< START\ t_{kj} >$ is processed where,
$t_{kj} \notin globalCommitted$ AND $t_{kj} \in < START\ CKPT(...) >$
end
Phase 2 Redo Phase
begin
do /* Scan forward from the $< START\ CKPT(..) >$ found in Phase 1
case the log record is update record $< t_{wi}, x, v, w >$
if $t_{wi} \in globalCommitted$
change the value of x to w /* restores after image of x */
until end of log is reached
end
Phase 3 Re-executing and Dependency Adjusting Phase
begin
for each Workflow $w \in incompleteWF$
begin
Get the workflow schema WS_w
for each task t_{wx} defined in workflow schema WS_w
if task $t_{wx} \in globalInterrupted$
$interrupted = interrupted \cup \{t_{wx}\}$
else if task $t_{wx} \in globalCommitted$
$committed = committed \cup \{t_{wx}\}$
else if task $t_{wx} \notin globalAborted$
$unscheduled = unscheduled \cup \{t_{wx}\}$
/* Processing the Interupted Task, re-executing phase */
for each $t_{wi} \in interrupted$ /* start from the earliest task that was interrupted */
begin
$interrupted = interrupted - \{t_{wi}\}$
if t_{wi} is re-executable **AND** all inputs in log
begin
Re-execute t_{wi}
$committed = committed \cup \{t_{wi}\}$
end

else /* t_{wi} is aborted and dependencies must be checked */
begin
$tempAborted = tempAborted \cup \{t_{wi}\}$
/* Find out all the affected tasks in committed_list */
for each task $t_{wk} \in committed$
if ($t_{wi} \rightarrow_a t_{wk}$ **OR** $t_{wi} \rightarrow_b t_{wk}$)
for each task t_{wj} in compensation fbw of t_{wk}
begin
$toschedule = toschedule \cup \{t_{wj}\}$ /* Compensate task T_{wk} */
$adjustSet = adjustSet \cup \{t_{wk}\}$
end
/* Find out all the affected tasks in interrupted_list */
for each task $t_{wj} \in interrupted$
if ($t_{wi} \rightarrow_{fba} t_{wj}$)
$toschedule = toschedule \cup \{t_{wj}\}$
if ($t_{wi} \rightarrow_a t_{wj}$ **OR** $t_{wi} \rightarrow_b t_{wj}$ **OR** $t_{wj} \rightarrow_c t_{wi}$)
$interrupted = interrupted - \{t_{wj}\}$
$adjustSet = adjustSet \cup \{t_{wj}\}$
$tempAbort = tempAbort \cup \{t_{wj}\}$
/* Find out all the tasks that need to be scheduled in unscheduled_list */
for each task $t_{wm} \in unscheduled$
if $t_{wi} \rightarrow_{fba} t_{wm}$ OR $t_{wi} \rightarrow_{ex} t_{wm}$
$toschedule = toschedule \cup \{t_{wm}\}$
if t_{wi} is alternate-executable
for each t_{wj} in alternate fbw of t_{wi}
$toschecule = toschecule \cup \{t_{wj}\}$
end
/* Processing the tasks that got affected */
while $adjustSet \neq \{\}$
if $t_{wt} \in adjustSet$
for each task $t_{wr} \in committed$
if ($t_{wt} \rightarrow_a t_{wr}$ **OR** $t_{wt} \rightarrow_b t_{wr}$ **OR** $t_{wr} \rightarrow_c t_{wt}$)
for each $t_{wx} \in$ compensation-fbw(t_{wr})
$toschedule = toschedule \cup \{t_{wx}\}$
$adjustSet = adjustSet \cup \{t_{wr}\}$
$adjustSet = adjustSet - \{t_{wt}\}$
end while
Submit *toschedule* to the scheduler
for each $t_{wi} \in tempAborted$
write log record $< ABORT\ t_{wi} >$ and flush the log
/* Reset variables for the next workfbw */
$committed = \{\}$, $interrupted = \{\}$, $unscheduled = \{\}$
$toschedule = \{\}$, $tempAborted = \{\}$
end
end

6 Conclusion

In order to coordinate the execution of the various tasks in a workfbw, task dependencies are specified among them. System crash or failures might occur during the execution of a workfbw. When a failure occurs, some tasks may have been partially executed. This results in an inconsistent state. The recovery mechanism is responsible for restoring consistency by removing the effects of partially executed tasks. The task dependencies may be violated while consistency is being restored. In this paper, we study how the task dependencies impact the recovery process and propose an algorithm that restores consistency and also respects the dependencies.

In future, we plan to give more details about this recovery process. For instance, we plan to discuss the kinds of checkpointing that are possible in workfbw systems, and how to optimize the recovery process. Finally, we would also like to expand our work and address issues pertaining to survivability of workfbws. This will include how we can recover from attacks caused by malicious tasks on the workfbw, how we can confine such attacks, and prevent the damage from spreading.

References

1. V. Atluri, W. Huang, and Elisa Bertino. An Execution Model for Multilevel Secure Workflows. In *Database Securty XI: Status and Prospects, IFIP TC11 WG11.3 Eleventh International Conference on Database Security*, August 1997.
2. Paul C. Attie, Munindar P. Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and enforcing intertask dependencies. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 134–145. Morgan Kaufmann, 1993.
3. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.
4. P. Chrysanthis. *ACTA, A framework for modeling and reasoning aout extended transactions*. PhD thesis, University of Massachusetts, Amherst, Amherst, Massachusetts, 1991.
5. J. Eder and W. Liebhart. Workflow Recovery. In *Proceeding of Conference on Cooperative Information Systems*, pages 124–134, 1996.
6. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice-Hall, 2002.
7. 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.
8. D. Hollingsworth. Workflow Reference Model. Technical report, Workflow Management Coalition, Brussels, Belgium, 1994.
9. M. Kamath and K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. In *Proceeding of the Fourteenth International Conference on Data Engineering*, February 1998.
10. B. Kiepuszewski, R. Muhlberger, and M. Orlowska. Flowback: Providing backward recovery for workflow systems. In *Proceeding of the ACM SIGMOD International Conference on Management of Data*, pages 555–557, 1998.