A Scheme to Specify and Implement Ad-Hoc Recovery in Workflow Systems

Jian Tang[1*] and San-Yih Hwang[2**]

[1] Department of Computer Science, Memorial University of Newfoundland
St.John's, Newfoundland, A1B 3X5 Canada
[2] Department of Information Management, National Sun Yat-Sen University
Kaohsiung, 80424 Taiwan

Abstract. Pre-defining a business process can substantially simplify the process design and the implementation of run time support. However, requiring that all the instances of the business process follow a fixed pattern (even if conditions are allowed) does not offer users sufficient flexibility to make changes to the process structures. These changes may be necessary due to the occurrence of exceptions, or other ad-hoc events. Some exceptions/events may be predictable in advance, and therefore can be incorporated into the process definition. However, not all exception/events can be predicted at the process definition time. When this happens, no corresponding exception-specific provision can be incorporated. The design and implementation of the system support for unpredictable exception/event handling is therefore a more complicated issue. In this paper, we study a special case of changing the structure of a business process in the context of unpredictable exception/event, namely, redirecting the control flow at run time in an ad-hoc manner. This phenomenon is termed ad-hoc recovery. We concentrate on two aspects in supporting ad-hoc recovery: the kind of interface that should be used and the increased functionality that must be built into the workflow database. For the latter, we also suggest implementation strategies to maximize the performance.

1 Introduction

The interests in workflow technologies were originated from office automation in early 80's. However, it was not until recently that people started realizing the important roles the workflow technologies play in increasing the productivity for business organizations. In a business environment, it is common that completing a single job requires the execution of many different activities at geographically separated locations. These activities must be executed in certain orders and some activities can be executed only under certain conditions. In other words, the execution of the activities must be coordinated. A workflow, also called a

* This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, individual operating grant OGP0041916.
** This research was sponsored by MOEA and supported by Institute for Information Industry, R.O.C.

business process, is concerned with the coordination of the activities in a business environment.

In many cases, the execution pattern of a business process can be pre-defined. Pre-defining a business process can substantially simplify the process design and the implementation of run time support. The research in this direction results in several proposals for workflow modeling techniques and implementation schemes. For example, The Workflow Management Coalition [16] proposes a workflow reference model which summarizes the main components the business processes normally require in a business environment as well as user interfaces for run time support. The issue of workflow modeling is also studied in works [4, 5, 8, 10, 11, 13]. In [15, 14], the authors propose tools for generating implementation efficiently. In [1, 9, 18], the authors study issues of enforcing intertask dependencies.

A problem of requiring that all the instances of the business process follow a fixed pattern (even if conditions are allowed) is that it does not offer users sufficient flexibility to make changes to the process structures. These changes may be necessary due to the occurrence of exceptions, or other ad-hoc events. Some exceptions/events are predictable in advance, and therefore can be incorporated into the process definition. One example is the conditional structures discussed in [12, 17], whereby depending upon different conditions, different structures can be selected at run time. However, not all exception/events can be predicted at the process definition time. When this happens, no corresponding exception-specific provision can be incorporated. The design and implementation of the system support for unpredictable exception/event handling is therefore a more complicated issue. Different aspects relating to this issue are investigated by several works recently [3, 2, 6].

There are many types of exceptions in a business process. Some of them are already allowed in the commercial systems, e.g., changes of participants assigned to a task; some of them are difficult to achieve, e.g., arbitrary alterations to the process structures. However, there exist types of exceptions that can be implemented. In this paper, we study a special case of changing the structure of a business process in the context of unpredictable exception/event, namely, redirecting the control flow at run time in an ad-hoc manner[3]. This phenomenon is termed *ad-hoc recovery*. Ad-hoc recovery is unpredictable in nature since at the process definition time it is not known when it will occur, what will be the cause, what kind of information users require to make a decision, and where the control flow will be redirected to. We concentrate on two aspects in supporting ad-hoc recovery: the kind of interface that should be used and the increased functionality that must be built into the workflow database. For the latter, we also suggest implementation strategies to maximize the performance. To the best of our knowledge, no other work has addressed this issue systematically.

[3] Note that by just redirecting the control flow at run time may not be sufficient to handle an exception. In addition, further actions may have to be taken, e.g., compensation of completed tasks. The kinds of actions to be performed, however, are orthogonal to our work.

The rest of the paper is organized as follows. In Section 2, we present a workflow model, outline failure recovery and introduce ad-hoc recovery. We then describe an architecture of workflow management system that will be used in the later discussions. In Section 3, we discuss in detail how ad-hoc recovery is supported. We describe the role of the interfaces and use examples to show how they are used, and characterize the ad-hoc recovery in terms of the functionalities of the DBMS. In Section 4, we propose implementation schemes. We conclude the paper by summarizing the main results and pointing out the future research directions.

2 Workflow and Workflow Management Systems (WFMS)

2.1 A workflow model

A workflow is an instance of a business process (or simply, process). This instance will achieve a prescribed goal when it terminates successfully. The execution of a workflow involves a number of tasks, each of which may run in a separate processing engine. A task defines some work to be done, and it may be activated more than once in any particular workflow. Each activation generates a task instance, which makes a transition between two workflow states. In general, at the modeling level we are not concerned with the detailed internal structure of a task, except for certain meta-information. Exactly what kind of meta-information is included in the model is application-dependent. In most cases this includes at least the following: task ids, states (not-activated, active, terminated-success, terminated-failed, etc.), instance ids, roles and agents. Depending upon the requirements by business applications, we may also want to include other information such as the starting/ending time of each task instance, as well as the related input/output values. A task is associated with a number of roles, and different tasks may be associated with the same role. The role is a placeholder for the agents that are responsible for the execution of each task. Agents can be either software or human agents.

Tasks may be executed at different (and usually heterogeneous) sites. The possible control flow in the workflow is modeled in terms of the transition, also called ordering, between the activations of the tasks. Each transition is associated with some conditions. These are predicates on workflow states. A transition may or may not fire in a particular workflow, depending upon the truth value of the attached condition. If a transition does not fire, then the task at the sink end will not be executed. The execution of each task may require input data from its preceding tasks and/or the global workflow. It may produce output data to its succeeding tasks or the users of the entire workflow.

In the following, sometimes it is necessary to refer to a workflow in progress. A workflow in progress can be described in terms of *execution dag*, as exemplified in Figure 1.

Each node in the dag denotes a task instance. Since a task may be invoked more than once, different task instances may result from different invocations

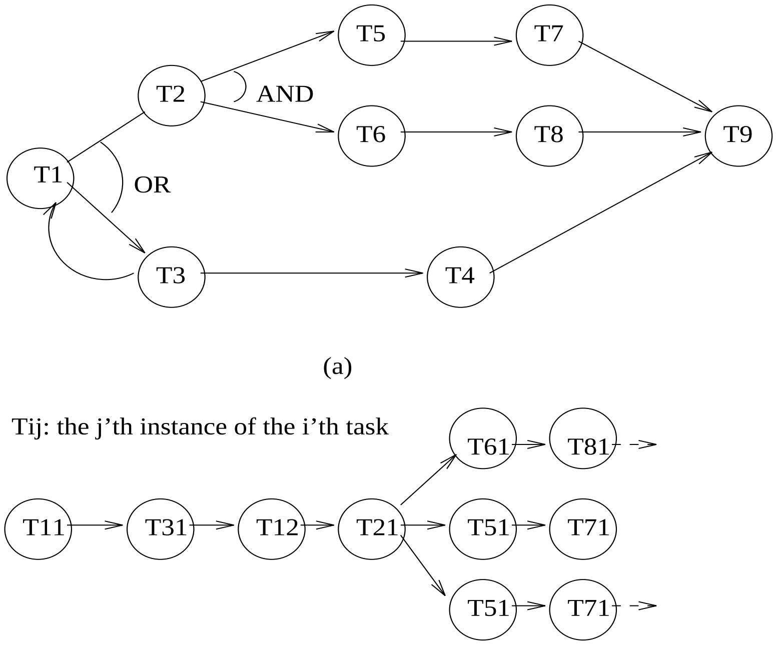

Fig. 1. (a) The structure of a business process (b) A sample execution dag

of the same task, such as T11, T12 in the diagram. In addition, due to the ad-hoc recovery, the same task instance may appear more than once. For example, after the first execution of T71, the user is not satisfied with the results and requests for the re-execution of T51, as shown in Figure 1(b). The arcs in the dag specify the order that was followed by the task instances. Thus a dag is a condensed version of the execution history for the workflow. Before the start of the workflow, it is empty. Whenever a task instance terminates, it is added to the dag as a leaf node. This expansion continues until the entire workflow terminates. The set of leaf nodes is termed *current stage* of the workflow.

2.2 WFMS architecture

A WFMS is a software tool that enables users to define and instantiate business processes. Defining a business process implies the specification of various components of the model in a particular business environment. During the execution, a WFMS coordinates the execution of the tasks. WFMSs can have different architectures to accommodate different requirements of business applications. However, in many cases a client/server architecture is deemed appropriate. In the following we use a client/server architecture similar to the one suggested in [16]. Figure 2 is an outline of such an architecture.

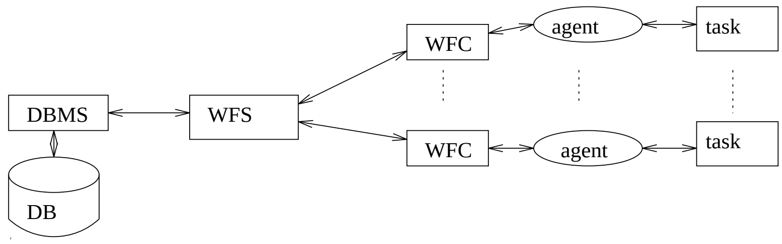

Fig. 2. A client/server architecture of WFMS

A WFMS contains a number of *clients* (WFC), which allow users to define the workflow and issue commands to retrieve information about the execution of a workflow. A server (WFS) is responsible for the coordination of tasks at run time. It monitors the progress of a process instance and schedules the next tasks to run by examining the conditions attached to the relevant transitions. The server is responsible for informing the roles once the tasks they are in charge of are ready. This is done by putting a task into the task-list of that role[4].

In order to schedule the tasks during the execution, each server may at run time make frequent accesses to the information about the process definition as well as that about the state of the process instances. These information are stored in the databases. For easy presentation, we assume each server is attached to a single database.

2.3 Ad hoc recovery

The control flow prescribed in a process definition represents the ideal situations for the business environment. During the real execution, however, based on some unexpected events or whenever he/she feels necessary an agent may initiate an ad-hoc recovery[5] by altering the control 'on the fly'. For example, in a workflow which handles a travel claim, after the applicant fills the travel claim form and passes it to the comptroller's office, the clerk may think that some information is missing or incorrect, and therefore require the claimant redo it. As another example, scientific workflows [19] typically contain tasks which collect experimental data. If after the data being collected the scientists who examine the data find that it is incomplete or not what has been expected, then they may want to repeat the task. This process may repeat for a number of times. These two examples correspond to situations where the control flow must be redirected back. In some cases the control flow may also need to be redirected ahead. For example, in loan application workflow, if the applicant is an important person

[4] Multiple servers can be used to enhance reliability/performance. The number of servers used, however, is orthogonal to our discussion.

[5] The term 'recovery' is chosen to mimic the normal meaning of recovery. However, rather than recovery from failure, the term should be understood here as recovery from 'outdated' (prescribed) control flow.

or if the case is urgent, then it is possible that some tasks, such as credit history checking and risk analysis, that are normally executed may be skipped over and the department manager immediately takes control of the case and makes the final decision.

Due to the non-deterministic nature of ad-hoc recovery, it is difficult to predict in advance when it may occur and how. This means that it is in general not feasible to prescribe into the process definition the possible change of control flow resulting from ad-hoc recovery. A direct implication of this is that the workflow must support ad-hoc recoveries only in terms of exception handling. On the other hand, in some situations the user who wishes to initiate an ad-hoc recovery may not have a complete knowledge about the structure of the workflow, or that of every task. Thus it is not always obvious to a user exactly from which tasks the process should be restarted. This implies that simply attaching a traditional procedural-style exception handler in advance may not be sufficient in coping with ad-hoc recoveries[6]. Users may have to play a more active role and/or the system may have to provide users with more assistance. We will discuss this issue in more detail in Section 3.1.

3 Supporting Ad-hoc Recovery

3.1 Interactions between users and WFMS

Users want to initiate an ad-hoc recovery because they are not satisfied with the outcome of the current task instance(s), or because the current situation changes (economically, organizationally, politically, personally, etc.) from that at the time of process definition. In either case, users must tell the WFMS what their restart choices are. There is normally more than one set of tasks to choose from. For example, if task t_1 generates unsatisfactory output, and gets the inputs from t_2 and t_3. The users may want to re-execute t_1 only, or t_2 and t_1, or all three of them. In general, any predecessor from which is originated a data flow path that ends at t_1 is a potential candidate for the restart.

The interactions between users and WFMS can take place in two ways. One is that a user specifies the set of tasks in a manner supported by the WFMS, and then lets the WFMS take over the remaining of the entire ad-hoc recovery. The remaining job includes searching for the correct representations of the tasks specified by the user and then performing the restart procedure. While this method frees users from most of the work, it lacks flexibility. In some workflows, just by referring information at hand a user may be still uncertain what the best choices are for the restart. They may wish to further examine the data associated with the tasks of the initial choices. For example, A user may want to identify first the tasks which are initiated during a high risk period for inputing polluted sample data. Then among the identified tasks s/he may decide to restart those

[6] This kind of dynamism is a special case in a more general context often cited as *unstructured workflow* in the literature.

which generate suspicious data, or those which were performed by some agents whom s/he does not trust, or simply those which are randomly chosen.

To allow this kind of flexibility, a multiway hand shake style of interaction is more appropriate. A user specifies a query through a well-defined interface and in terms of the attributes of the tasks. The attributes may include, for example, input/output type or value, date/time, agent name (login name), predecessor level (for example, parents and grandparents respectively have levels 1 and 2), etc. The query can be issued against the WFS database schema that describes all the above information. The WFS database schema, for example, could be similar to the one defined by the WFMC [20]. Upon receiving the response, the user is able to examine the values of some other attributes. This interaction can be repeated a number of times, until the user thinks that s/he can make a decision.

To illustrate, still consider the previous example. Suppose the user intends to examine the tasks initiated between 10am and 1pm in April 1, 1996 and then gets an idea about the agents' names who performed those tasks. Based on the information obtained thereafter s/he will decide the tasks to be chosen for the restart. (For simplicity, we assume each agent is in charge only of one task.) We assume the following operations are contained in the interfaces provided for the user:

- TaskSet gettasks_by_when(TimeDate $start\_after$, $start\_before$);
- TaskSet gettasks_by_names(StringSet $names$);
- StringSet getnames(TaskSet $tset$);
- restart(TaskSet $tset$);

The user first makes a request through the client program which contains the corresponding invocations (assuming variables have been properly declared):

$names \leftarrow$ getnames(gettasks_by_when("10:00,April 1,1996","13:00,April 1,1996"));

Suppose, by examining the results in $names$, the user discovers two names, J. Smith and C. Linda, whom the user thinks might be unreliable. S/he then restarts the workflow from the tasks these two agents are in charge of:

$tasks \leftarrow$ gettasks_by_names({"J. Smith", "C. Linda"});

restart($tasks$);

To aid the users in obtaining the knowledge about the kinds of information stored in the system, a browser can be used. More detailed discussion about the structure of a browser is beyond the scope of this paper.

3.2 Additional DBMS functionality

As mentioned before, DBMS is an important component in any workflow system. The information it manages are used by the WFMS to keep track of the ongoing process instances, to schedule the tasks to be executed next, and to perform some other application/system related activities, such as accounting, analysis, simulation, failure recovery, etc.

When the functionality of a WFMS includes the support of ad-hoc recovery, the DBMS must also increase its functionality. Actually, the key role in the ad-hoc recovery is played by the DBMS. This is because redirecting control flow for an ongoing process instance essentially means changing the current state stored in the database for that instance. This is primarily the job of a DBMS. Once the current state has been switched to the other one, the WFMS will do the scheduling as usual, but based on the newly installed state. We assume the DBMS contains a module, ad-hoc recovery manager (AHRM), that is responsible for this job.

Recall that the control flow in ad-hoc recovery may be redirected either ahead or back. If it is redirected ahead, then the current state must be changed to some future states. If it is redirected back, then some previous state must be re-installed as the current state. In the former case, probably there is not much the DBMS can do, except for overwriting immediately the current state with the appropriate future state. This future state must be supplied by the user (since it has not yet been stored anywhere in the database). In the latter case, the DBMS must search on the old states for the one from which the restart can proceed correctly. This is not a simple job, considering the fact that no provision has been previously made to indicate where the desired state might be, and the potentially larger number of old states. Therefore, in the following discussion we discuss the AHRM only in the context of ad-hoc recovery with control flow being redirected back.

3.3 Ad-hoc recovery manager

ad-hoc recovery versus failure recovery In some sense the AHRM is similar to the failure recovery manager (FRM) in a DBMS: both modules restore some previous states. Thus both of them must use log records. However, there are two major differences between the two. Firstly, the FRM can re-install the desired state by using the most recent savepoint (or the start of the uncompleted tasks) as the reference while AHRM does not have such reference. This means that the AHRM must use different mechanism to construct the desired state. Secondly, a failure recovery is a global action which affects all the ongoing process instances. Thus it is usually done in a monolithic action which treats the entire database as a single object. In contrast an ad-hoc recovery can be viewed as a local action which affects individual instances only, namely, those for which the ad-hoc recoveries have been explicitly requested. If more than one process instance has been requested for ad-hoc recovery, then the same procedure has to be repeated for each of them. Thus the overall time overhead may be of a more serious concern than that in failure recovery. We will get to this issue in more detail in Section 4. In the following, we will take a closer look at the structure of failure recovery log, and then illustrate how to modify it to accommodate ad-hoc recovery.

records in failure recovery log We assume three kinds of log records are used in the failure recovery log.

– task data log record: $< T, D, v>$, where T is a 4-tuple <P,I,t,i> where P, I, t, i, respectively, are process id, process instance id, task id, and task instance id, D is a data item pertaining to T (agents' name, starting date/time, etc.), and v a specific value of D.
– task status log record: $< T, u >$ where T has the same form as above, and u is an indicator for the current status of T (e.g, . *not-activated, active*, etc.).
– checkpoint record: CR which contains information to assist in constructing the workflow state at the time this record is created.

Depending upon the requirement of specific workflows, the DBMS may also use other kinds of log records, such as those recording external events, accounting data, simulation results, etc. But the above are the most relevant to our discussion.

Log records are stored in the log following the temporal order of the corresponding operations. When the workflow server fails, all the ongoing process instances are interrupted. Typically, the FRM starts from a log record near the checkpoint and scans the log records forwards to update the database to a state that was generated at the time of failure. It then conducts a backward scan to eliminate the effects of the tasks that are half done. This procedure is very similar to the undo/redo procedures normally used to support the atomicity property of transactions in traditional database systems.

ad-hoc recovery using failure recovery log As alluded to in the previous section, the AHRM may need different log structures for its efficient use. However, to save storage, it is highly desirable for the AHRM to share the same log structure with FRM. This suggests that we should modify the failure recovery log in such a way that the FRM will not be affected while at the same time the functionality of AHRM is supported.

Suppose a user decides to restart the workflow execution from the set of tasks $R = \{r_1, \cdots, r_m\}$. Similar to the operations of the FRM, the AHRM needs to scan the log queue backwards to regenerate the workflow state immediately before the tasks in R are initiated. The problem is: what actions must be taken for each log record during the backward scanning and at which point this scanning should stop.

For a convenient abuse of the symbols, we use r_i to denote also the most recent task instance of r_i at the time when the ad-hoc recovery is initiated. Clearly each r_i is an ancestor of some tasks at the current stage in the execution dag. Let $S = \{s_1, \cdots, s_n\}$ be the set of the descendants of some task instance in R which belong to the current stage. Thus any task at the current stage outside S will not be affected by the ad-hoc recovery. Let $path(r_i, s_j)$ be a path from r_i to s_j in the execution dag. Any task in the path may be repeated after the recovery. On the other hand, if a task is not in the above kind of path for any $r_i \in R$ and $s_j \in S$, then it will not be repeated for sure during the recovery process. In other words, during the backward scanning, for each log record corresponding to a task instance which is not a descendant of any task in R, no action will be taken. For those records which correspond to the descendants of some tasks in

R, all of the values stored in the database for those tasks must be undone. This is necessary since the database state we are trying to reconstruct was originally generated before those tasks were executed. As this backward scanning proceeds, eventually the desired database state will be reconstructed. This is when we have exhausted all the *active* log records for the tasks in R. A log record for a task is active if it is generated after the task is initiated. Not all log records are active. For example, at the beginning of a workflow execution, we may set the status of all the tasks as 'not-activated'. Any log record describing this operation on a task is not active. Clearly, operations described by a non-active log record should not be undone in the backward scan.

We are still left with the problem of how we know if we have exhausted all the active log records for any task in the backward scanning. A straightforward yet inefficient approach is to continue scanning backwards until either the head of the log queue is rearched or a non-active log record is encountered for the task. A better approach is to include a flag in each record which indicates the first record to be generated for a task instance after it is initiated. However, to deal with this flag properly, we need to consider the fact that in ad-hoc recovery some task instances will be initiated multiple times, as illustrated below.

Suppose the desired state has been reconstructed for a process instance. The AHRM will let the WFS rerun that process instance starting from that state, namely the tasks to be executed next are those in R. Note that at this time the backward scanning has stopped at a position somewhere between the beginning and the end points of the log queue. On the rerun, new log records will be generated. A problem is where these log records should be placed. One option is to substitute them for the records for the old run. A problem with this approach is that in some workflows, users may wish to keep the old information for future analysis. Thus completely eliminating the obsolete log records may be unacceptable. Therefore, we adopt the following alternative. The log records for the old run are kept intact in the log, and those generated in the rerun will be appended to the log queue. A question is: when we specify the task instances for the log records for the rerun, should we use a new id, or an identical one to the old run? We adopt the latter. The rationale is that a rerun is in essence aimed at 'repairing' the old run. Thus the rerun of the task instance should (logically) substitute for, not co-exist with, the old one, and therefore is entitled to claim the id of its predecessor. Furthermore, this strategy also makes the query for historical data simple, since different versions of the execution of the same task instance have been explicitly clustered together through the identical task instance id.

As a consequence of the strategy adopted above, the information about a single task instance may be recorded multiple times in the log, i.e., once for the most recent run, and zero or more times for the previous run(s). Those for the more recent run are stored closer toward the end of the log queue. For efficiency as well as correctness reasons we require that the backward scanning procedure stop scanning once it has exhausted the log records for some tasks in R in the most recent run. Thus log records for different runs of the same task instance must be identifiable. This goal is attained by a technique suggested in Section 4.1.

Another issue is that if the user opts not to repeat an actual task in the rerun, or if the task itself is not repeatable, then no new log records will be generated for the task. Instead, the AHRM must simulate the execution of that task by referring to the information stored in the old log records, and restore the necessary attribute values for that task into the database. During this simulation, the old log records should be copied to the new location corresponding to the rerun.

4 Algorithm

4.1 The structure of log records

We adopt a traditional mechanism to identify individual log record, i.e., each log record contains a *log sequence number* (LSN) field. It starts from zero, and incremented by one for the next log record generated. We assume that given an LSN for any record, the AHRM can determine the physical location (usually a disk block address) of that record in a constant amount of time. As illustrated in the previous section, log records for different runs of the same task instances must be identifiable. For this purpose, we include a field, STARTRUN, in each log record. This field takes value of either *true* or *false*. After a task instance has been started, and before it terminates, among all the log records which are generated for it only the one generated first has this field set to *true*. Thus for any task instance, the sequence of all active log records for it can be viewed as a collection of segments. Each segment starts with a record with value of *true* in field STARTRUN, and followed by a sequence of records with value of *false* in this field. We call each of these segments a *run unit* for the corresponding task instance. A diagramatic illustration of run unit is shown in Figure 3[7].

As mentioned before, the AHRM works on per-process instance basis. For each process instance which needs an ad-hoc recovery, the AHRM is concerned only with the log records relating to ('relating to' is interpreted as 'describing a task instance which belongs to') that process instance. Thus most of the log records on the road may be irrelevant. For performance reason, these records should not be scanned. This is done by including a field, NEXTLSN, in each log record. For log record g, if STARTRUN is false, its NEXTLSN field contains the largest LSN of any preceding log record relating to the same task instance; otherwise, it contains the set of the largest LSNs of all the immediate ancestors of the task instance related by g in the execution dag. Thus NEXTLSN fields bind together all the log records relating to a single process instance. In addition, they define a partial order consistent with the execution dag for that process instance.

In the following we use the functional notations to summarize the control information that can be retrieved from each log record g:

[7] We assume only one process and use subscripts to identify tasks and task instances. Thus the four tuple notation in Section 3.3 is simplified here.

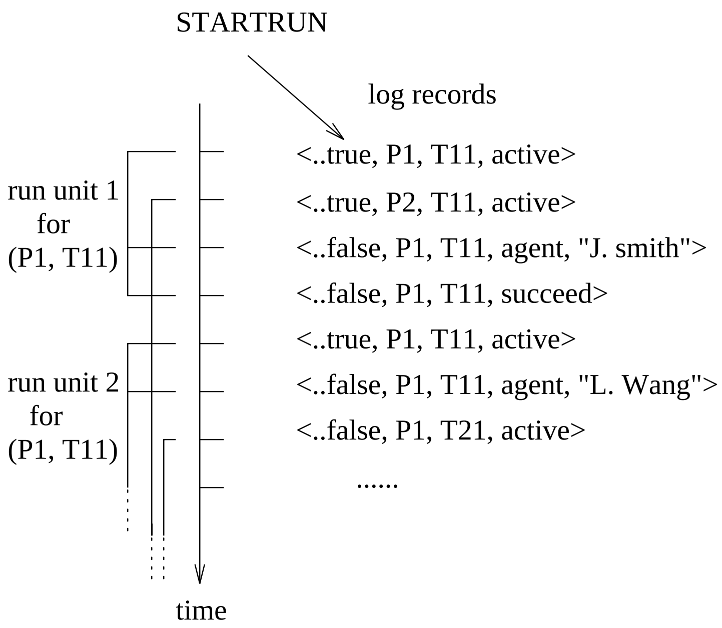

Fig. 3. Run units for task instances

- *Task*(g): returns an id that uniquely identifies the task instance described by g;
- *LSN*(g): returns the log sequence number of g;
- *NextLSN*(g): returns the values in the NEXTLSN field of g;
- *StartRun*(g): returns *true* iff STARTRUN field of g is *true*, which means g is the first log record generated in the current run unit for *Task*(g);

4.2 Algorithm for backward scan

The most work performed in the ad-hoc recovery is to reconstruct the required states for individual process instances. Once this is done, the remaining is straightforward. The workflow server simply resumes the normal execution for those process instances, starting from the newly constructed states. During the execution, log records are appended to the log queue, and at the same time the various fields in the log records are filled with the appropriate values as described above. Thus in the following we will only discuss the algorithm for backward scanning.

Let P be the process instance which is requested for ad-hoc recovery. Let R be the set of task instances to restart from. Let S be the set of all those descendants of the task instances in R which belong to the current stage. The AHRM must start the backward scanning from the record that has the largest LSN among all those relating to some task instances in S. During the backward scan, for any task instance the first run unit that is scanned must be the most recent one for it.

Log records are stored in secondary storage, such as disks. They are usually grouped into disk blocks. A disk block is the basic unit for data transfer between a disk and a main memory. Since during the ad-hoc recovery, the process instance must be suspended and also since the AHRM iterates over a potentially large number of process instances, minimizing the number of block accesses in each iteration is essential to the overall performance. To avoid unnecessary I/O, we let the AHRM fetch only those disk blocks which contain some log records that are related to P, and ignore those which do not. To this end, the AHRM must determine the next block to fetch once the current block is finished. The detail is described in the following algorithm.

AHRM Procedure - Single

{*NextLSNs* is the set of LSNs of the yet-to-be-undone records which follow via the NEXTLSN fields the ones already undone. The largest LSN in this set identifies the next log record to be undone.}

1. *Lock*(P);
2. $NextLSNs \leftarrow \{max\{LSN(g) : Task(g) = t\} : t \in S\}$;
3. $L \leftarrow max(NextLSNs)$;
4. if the log record with LSN $= L$ is not in main memory then
5.   fetch the disk block containing the log record whose LSN $= L$;
6. endif;
7. $g \leftarrow$ log record whose LSN is L;
8. undo the operation described by g;
9. $NextLSNs \leftarrow NextLSNs$ - $\{LSN(g)\}$;
10. if $StartRun(g) = false$ then
11. $NextLSNs \leftarrow NextLSNs \cup \{NextLSN(g)\}$;
12. else
13.   if $Task(g) \notin R$ then
14.   for each log record q such that $LSN(q) \in NextLSN(g)$ do
15.   if $Task(q)$ is a descendant of some member in R in the execution dag then;
16. $NextLSNs \leftarrow NextLSNs \cup \{LSN(q)\}$;
17.   endif;
18.   endfor;
19.   endif;
20. endif;
21. if $NextLSNs = \phi$ then
22.   *Unlock*(P);
23.   Exit;
24. endif;
25. go to step 3;

We assume there is a *Lock*(P) operation which puts a (logical) lock on process instance P. This lock prevents the workflow server from executing any operation relating to P. A matching *Unlock*(P) operation releases the lock on P. It also signifies the workflow server to resume the normal execution of P. Step 2 initializes *NextLSNs* to the set of LSNs in which each one is the largest for a task instance in S. This step requires the AHRM perform an initial backward scan to obtain these LSNs. In step 12, when $Task(g)$ is not in R, and $NextLSN(g)$ contains the log records of other tasks, it is ensured that a log record in $NextLSN(g)$ is included in $NextLSNs$ only when its pertaining task is a descendant of some member in R. In step 21, an empty $NextLSNs$ signifies that the desired state has been constructed, and hence the end of the backward scan.

The previous algorithm can be modified to handle batched requests, which signifies the case where several process instances are requested for ad-hoc recovery and these requests are made in a time interval much shorter than the duration of each process instance. The modified version is similar to the old one, except that this time the log records to be scanned correspond to a bigger set of tasks, which is the union of all the restart sets by the batched requests.

5 Conclusion

In this paper, we study issues relating to ad-hoc recovery, a special case of workflow applications where users are allowed to make changes to the predefined workflow control structure in an ad-hoc manner. This issue is significant since it provides users with flexibility which are sometimes required to adapt the workflow execution to the exceptions or ad-hoc events. We characterize the ad-hoc recovery by comparing it with failure recovery. We describe the kind of interface that can effectively aid users in making a decision. We discuss the system support in terms of special functionality that must be embedded into the DBMS, and suggest strategies that minimize the overhead.

Our work can be extended in several directions. Firstly, although we propose an interface that enables users to interactively query the execution status of the workflow, this interface is basically a passive tool. Users of the interface may have to know the process flow and the participants in order to determine the set of restart tasks. We are currently investigating the issue of proposing suggestions based on previous processing experiences when exceptions occur. Some machine learning or data mining techniques could be employed in dealing with this problem. Secondly, ad-hoc recoveries may occur much more often than failure recoveries. To speed up the ad-hoc recoveries, it may be benefitial to cluster log records for the same workflow. While this approach may reduce the time for ad-hoc recoveries, it may increase the overhead for both failure recovery and task commit. Detailed evaluations need to be performed. Finally, our algorithm locks the entire workflow while ad-hoc recovery is performed. Thus, even tasks not affected by the roll back cannot proceed during the recovery. Finer lock granularity could be employed to achieve higher availability.

References

1. P.C. Attie, M.P. Singh, A. Sheth, M. Rusinkiewicz: Specifying and Enforcing Intertask Dependencies. Proc. of 19th VLDB Conference, VLDB Endowment. (1993) 134–144.

2. P. Barthelmess, J. Wainer: Workflow Systems: a few Definitions and a few Suggestions. Proc. of ACM Conference on Organizational Computing Systems. (1995) 138–147.
3. R. Blumenthal, G. Nutt: Supporting Unstructured Workflow Activities in the Bramble ICN System. Proc. of ACM Conference on Organizational Computing Systems. (1995) 130–137.
4. P.K. Chrysantis, K. Ramamritham: Synthesis of Extended Transaction Models Using ACTA. ACM Transactions on Database systems. **19** (1994) 450–491.
5. U.Dayal, M. Hsu, R. Ladin: A Transactional Model for Long-Running Activities. Proc. of 17th VLDB Conference, VLDB Endowment. (1991) 113–122.
6. C. Ellis, G. Rozeberg: Dynamic Change Within Workflow Systems. Proc. of ACM Conference on Organizational Computing Systems. (1995) 10–21.
7. A. Elmagarmid: (ed.), Database Transaction Models for Advanced Applications. Morgan-Kaufman Publishers, USA. (1992).
8. A. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz: A Multidatabase Transaction Model for Interbase. Proc. of the 16th VLDB Conf. (1990).
9. D. Georgakopoulos, M.F. Hornik: Framework for Enforceable Specification of Extented Transaction Models and Transactional Workflows. Intl. Journal of Intelligent and Cooperative Information Systems. (1994).
10. D. Georgakopoulos, M. Hornik, A. Sheth: An Overview of Workflow Management: from Process Modeling to Workflow Automaton Infrastructure. Dist. and Parll. Databases: An International Journal. **3** (1994) 119-153.
11. G. Kappel, P. Lang, S. Rausch-Schott, W. Retschitzegger: Workflow Management Based on Objects, Rules, and Roles. Bulletin of the Technical Committee on Data Engg. **18** (1995) 11–18.
12. N. Krishnakumar, A. Sheth: Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations. Dist. and Parll. Databases: An International Journal. **3** (1995) 1–33.
13. D. McCarthy, S. Sarin: Workflow and Transactions in InConcert. Bulletin of the Technical Committee on Data Engg. **16** (1993). 53–56.
14. M. Singh: Synthesizing Distributed Constrained Events from Transactional Workflow Specifications. Proc. of 12th Intl. Conf. on Data Engg. (1996) 616–623.
15. B. Salzberg, D. Tombroff: DSDT: Durable Scripts Containing Database Transactions. Proc. of 12th Intl. Conf. on Data Engg. (1996) 624–633.
16. Workflow Management Coalition: Workflow Reference Model. (1996).
17. J. Tang, S. Hwang: Coping with Uncertainties in Workflow Applications. Proc. of Intl. Conf. on Information and Knowledge Management. (1996).
18. J. Tang, J. Veijalainen: Enforcing Inter-Task Dependencies in Transactional workflows. Proc of 3rd Intl. Conf. on Cooperative Information Systems. (1995).
19. J. Wainer, M. Weske, G. Vossen, C. Medeiros: Scientific Workflow Systems. Proc. of the NSF Workshop on Workflow and Process Automation in Information Systems. (1996).
20. WorkGroup 1/B: Workflow Management Coalition Interface 1: Process Definition Interchange. The Workflow Management Coalition. (1996).