Correctness Issues in Workflow Management §

Mohan Kamath and Krithi Ramamritham
Department of Computer Science,
University of Massachusetts,
Amherst MA 01003, USA.
e-mail: {*kamath,krithi*}*@cs.umass.edu*

Abstract. Workflow Management is a technique to integrate and automate the execution of steps that comprise a complex process, *e.g.*, a business process. Workflow Management Systems (WFMSs) primarily evolved from industry to cater to the growing demand for office automation tools among businesses. Coincidentally, database researchers developed several extended transaction models to handle similar applications. Although the goals of both the communities were the same, the issues they focused on were different. The workflow community primarily focused on modeling aspects to accurately capture the data and control flow requirements between the steps that comprise a workflow, while the database community focused on correctness aspects to ensure data consistency of sub-transactions that comprise a transaction. However, we now see a confluence of some of the ideas, with additional features being gradually offered by WFMSs.

This paper provides an overview of correctness in workflow management. Correctness is an important aspect of WFMSs and a proper understanding of the available concepts and techniques by WFMS developers and workflow designers will help in building workflows that are flexible enough to capture the requirements of real world applications and robust enough to provide the necessary correctness and reliability properties. We first enumerate the correctness issues that have to be considered to ensure data consistency. Then we survey techniques that have been proposed or are being used in WFMSs for ensuring correctness of workflows. These techniques emerge from the areas of workflow management, extended transaction models, multidatabases and transactional workflows. Finally, we present some open issues related to correctness of workflows in the presence of concurrency and failures.

1. Introduction

In the last decade, there has been a growing demand for tools that facilitate office automation and enterprise re-engineering. The goal is to improve the efficiency of

§ This work was supported by NSF grant IRI-9314376 and a grant from Sun Microsystems Labs

enterprises by defining *business processes* that integrate related tasks that are executed at different locations within the enterprise. Thus business processes are typically of long duration and may access data from multiple sites. Coincidentally, two approaches have emerged to tackle the needs of such applications.

With efforts primarily from industry, *workflow management* has emerged as a popular technique to integrate and automate the execution of steps that comprise a workflow (business process). *Workflow Management Systems* (WFMSs) provide support for modeling, executing and monitoring the workflows. WFMSs allow the composition of large applications from smaller independently developed applications. Several prototype and commercial WFMSs have been developed and deployed [11, 33, 22, 23, 15, 36]. The workflow community primarily focused on modeling aspects of workflows, so as to accurately capture (i) the data and control flow requirements between the steps that comprise a workflow and (ii) the organizational hierarchy and staff assignments. Several simulation and other analysis tools have been developed for studying and improving the efficiency of workflows. These are essential for addressing the needs for real working environments. However, correctness aspects have largely been ignored.

The database community also sensed the need for developing transaction processing systems to handle the needs of new applications like design and office automation. Realizing the limitations of the *traditional transaction model* for handling long duration applications, several *extended transaction models* (ETMs) [9] were proposed that relax the ACID (Atomicity, Consistency, Isolation and Durability) properties in various ways. Specifically, the focus was on correctness aspects so as to ensure data consistency of sub-transactions that comprise a transaction. By exploiting the semantics of the applications and using relaxed correctness criteria, ETMs provide special features to handle concurrency control and recovery. However, ETMs require all the activities of a task to be transactional and enforce tight integration between the sub-transactions which are too restrictive for many applications. Hence ETMs have not been incorporated into commercial products but for some exceptions like nested transactions [37].

Fortunately, in the last few years there has been a confluence of the two approaches. The database community has applied some correctness concepts like isolation and failure handling requirements from transactions (including ETMs) to *general*† workflows to create *transactional workflows* [41], whose steps primarily correspond to database transactions. Similarly the workflow community has borrowed ideas from ETMs, (*e.g.*, *spheres of joint compensation* [30] motivated by spheres of control [8] and Sagas [14]) in an effort to improve the correctness properties offered by WFMSs. It has also been demonstrated that the semantics of *some* of the ETMs can be implemented using

† In this paper, general workflows are those that integrate independently developed applications. Most commercial WFMSs have been supporting only such workflows.

workflow models [1]. Another closely related area is that of *multidatabases* or *federated databases* [6, 34] where several techniques have been developed for handling concurrent transactions whose sub-transactions access data from autonomous databases in the presence of failures. Some of these techniques have also been used for improving the correctness properties offered by transactional workflows[40]. All these developments contributed to an increase in the robustness and reliability offered by WFMSs.

This paper provides an overview of correctness issues in workflow management. Since the paper requires a general understanding of workflow management concepts, in section 2 we briefly describe the modeling and execution support available in WFMSs. A step receives data from one or more steps of a workflow, and often a program that executes on behalf of the step accesses shared data from a remote resource manager. Since there is inter- and intra- workflow sharing of data, techniques are needed to ensure data consistency. Hence we motivate the need for correctness in section 3. The correctness requirements of WFMSs can be broadly classified into two categories — execution atomicity and failure atomicity. Hence we survey techniques that have been proposed or are being used in WFMSs for handling execution and failure atomicity requirements of workflows in sections 4 and 5 respectively. Execution atomicity deals with how data is committed and how visibility of data between steps within a workflow and between workflows can be controlled. Failure atomicity determines what is to be done with the data that has already been committed by steps of a workflow before a failure occurs disrupting the workflow. We consider the effects of both system failures and logical failures. The techniques surveyed cover the areas of workflow management, extended transaction models, multidatabases and transactional workflows. Finally, in section 6 we present some open issues related to correctness of workflow execution in the presence of concurrency and failures. Section 7 concludes with a summary of the paper.

2. Basics of Workflow Management

In this section we describe the basic modeling and execution support offered by WFMSs for general workflows. We focus only on the important details needed for the discussion in the rest of the paper.

2.1. Modeling Support

WFMSs provide primitives to define workflow schemas or business processes. As shown in Figure 1, a workflow is defined as a sequence of steps. A step definition consists of what tools/programs are to be used for executing the step. Each step has a set of input and output parameters. To check that a step is started and completed correctly, a start and finish condition can be associated with it [11]. There are two types of directed arcs that connect the steps — *data flow* arcs and *control flow* arcs. A data flow arc maps an output parameter of a step to input parameters of one or more steps. This mapping can range from simple integer values to spreadsheet names and other complex objects. A control flow arc connecting two steps determines the execution dependency between the steps. Often a control flow arc has a condition attached to it. This provides the functionality for defining branching, merging, sequential/parallel execution, and alternative execution of steps. In addition they can also be used to define loops consisting of one or more steps. As shown in Figure 1, data and control flow arcs form the key components of a workflow schema. Workflows can also be nested by mapping a step to a different workflow. This is shown by the "medical evaluation" step in Figure 1. In addition, there is modeling support to define the organizational hierarchy and staff names with their designation. A step definition also contains a designation of the staff member responsible for executing an activity. This provides flexibility since any person with that designation can execute the step rather than someone specific. All the modeling activities are performed via a workflow definition tool which is often GUI based.

2.2. Execution Support

Figure 2(a) presents an architecture of a WFMS closely conforming to the reference model [21] of the *Worflow Management Coalition* (WfMC). The definitions of workflows, steps and staff designations are all stored persistently in an underlying database commonly referred to as the *workflow database*. This database also stores the states of the workflows that are in progress. Scheduling is usually performed by a *workflow engine* which refers to the workflow database to determine the state of the various workflows in

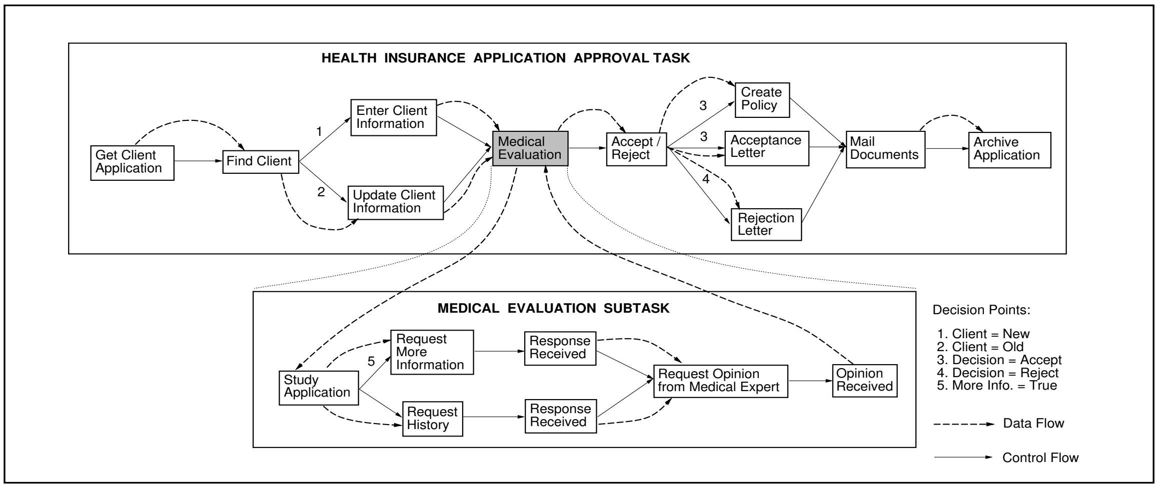

Figure 1. *Example of a Workflow*

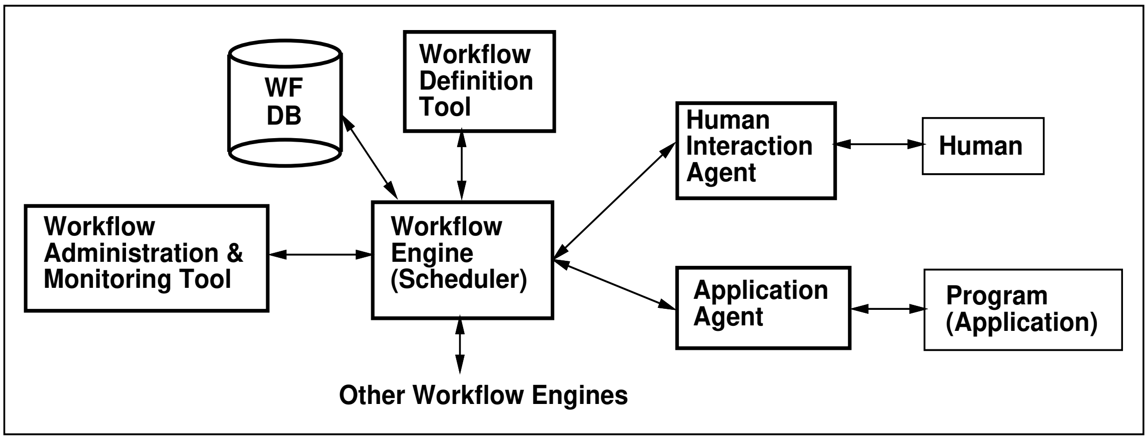

Figure 2. *Workflow Management System*

progress. Staff members interact with the WFMSs through a *human interaction agent.* The staff is presented with a work item list that lists all the steps that have been assigned to the staff. If a program is to be executed to perform a step, then the program is actually invoked by an *application agent.* The application agents are essentially daemons that run on different nodes where the programs are to be executed. The application agents interact with the workflow engine to fetch the data required to execute a step and to communicate back the output (*i.e.*, return status code and data) produced by a step. The programs in turn can access different resource managers some of which may be transactional like DBMSs and others which may be non-transactional like file systems and spreadsheets. The WFMS has no control over these resource managers since only the programs interact with them. A workflow engine can also communicate with other workflow engines for transferring control to execute a step or part of a workflow.

3. Correctness Requirements

In this section we will provide a high level description of the various correctness issues that have to be considered in workflows. These are usually associated with transactions but they are important in the context of workflows as well.

Once a workflow is invoked, the steps are executed according to the control and data flow information in the schema. A step receives data from one or more steps within a workflow, processes the data and passes them to other steps. For processing the data, a step is often associated with a program which accesses data from remote resource managers. Several other programs representing other steps from the same or different workflow can access the data from the same remote resource manager. Thus there is inter- and intra- workflow sharing of data. Whenever there is data sharing, the effect of concurrency and failures must be taken into account. Consider the following example.

When a step completes or commits, there are essentially two copies of the data items returned by the step — one at the remote resource manager where the step accessed them and the other in the workflow database as part of the workflow state information. Another program representing a different step (perhaps from another workflow) can access the same data at the remote resource manager and update it. Now the copy of the data stored in the workflow database is stale. It may be used by a subsequent step in the workflow to make a decision. Obviously the decision is made based on a invalid copy of the data and the consequences would depend on the nature of the decision.

Failures are of two types — system failures and logical failures. System failures occur when one or more of the WFMS components, *i.e.*, the workflow engine, the workflow database, or the agent fail. This can affect several steps and workflows that are in progress. Logical failures occur for example when a program associated with a step fails. This can be due to several reasons — exceptions within the program, failure of the remote resource manager, unavailability of resources and so on. In workflow management, the number of logical failures is usually high compared to system failures. In traditional transactions, the entire transaction is rolled back upon a failure. That is not acceptable for workflows.

To summarize, some of the specific questions to be addressed in the context of workflow correctness include:

(i) How can it be determined if a step in a workflow is successful ?

(ii) What is the effect of interleaving of steps from different workflows?

(iii) When one or more steps in a workflow fail, what happens to that workflow and other workflows that have accessed data produced by the failed workflow?

(iv) What happens to a workflow when one or more of the WFMS components fail?

All these questions are related to the execution and failure atomicity requirements of workflows and in the next two sections we review some of the techniques that have been proposed to address these questions.

4. Execution Atomicity of Workflows

Traditional transactions use *serializability* [4] as the correctness criterion. Hence the notion of execution atomicity is that none of the changes made by the transaction are externally visible (to other transactions) before it commits. However, this is not suitable for workflows due to two reasons (i) workflows are of long duration and (ii) the steps access heterogeneous data from autonomous local sites and complete (commit) independently. However, if steps from different workflows are allowed to interleave in an uncontrolled fashion, there can be inconsistencies. Below we survey some of the solutions that have been proposed.

The simplest form of support for controlling concurrent access to data from steps within a workflow or from different workflows is provided in WFMSs like InConcert [33] via *check-in* and *check-out*. This scheme is suitable for workflows in engineering environments such as CAD/CAM and CASE where decisions to access objects are more ad hoc. It is however not suitable for production workflows that integrate existing applications by explicitly specifying the data and control flow definitions at the workflow level. In such workflows, data items are accessed by programs representing the individual steps and the WFMS has no direct access to these data items.

To provide workflow wide concurrency for accesses to objects without allowing other workflows to observe the changes, a transactional nested process management system has been described in [7]. It provides flexibility in the way objects are committed. A step can delegate the responsibility of committing and aborting operations on certain objects to an ancestor either through an intermediate ancestor or directly. This model improves the concurrency within a workflow compared to the closed nested model [37]. For example, if a step commits its operations to a top-level step, its results are still internal to the workflow but they are accessible to all other steps within the workflow. This type of execution atomicity is ideal for workflows in engineering environments where more sophistication is required rather than the simple check-in and check-out model.

In ConTracts [44, 38], *invariant* based synchronization is used to support the executability of a ConTract (workflow). This addresses the problem we discussed earlier in section 3. Consider two steps in a workflow, the first reading an object and the second writing the same or other objects based on the value read in the first step. An example of this is a workflow that checks flight information in the first step and reserves the flight in the next step. If a seat is available in the first step, to guarantee that the seat can be reserved in the second step, the obvious scheme would be to treat the two steps as an atomic unit. However this would restrict access to other steps that need to check the flight information. Hence the invariant based approach in ConTracts establishes a constraint using a predicate after the completion of the first step, *e.g.*, keep at least one seat available. The constraint is removed after the successful completion of the second step. Thus the validity of data read in a previous step can be ensured without restricting access to data.

A few other schemes have been suggested in the context of transactional workflows. We discuss them in the rest of this section. These schemes are in some sense motivated by the concept of *relative atomicity* [13, 31] and *breakpoints* [12] discussed in the context of semantics based concurrency control in transactions for relaxing the serializability requirements by exploiting the semantics of the objects and transactions accessing the objects.

A *step compatibility* based approach is suggested in [5] to control the interleaving of steps from different workflows. They consider an example of a loan processing request where the execution order of two steps, 'risk evaluation' and 'risk update' is critical, *i.e.* given any two workflow instances, these steps from two workflows must be executed serializably. The ordering requirement is specified in a compatibility matrix where the above mentioned steps are declared to be incompatible. The workflow scheduler then uses this compatibility matrix to schedule the execution of the individual steps. Compatible steps are allowed to interleave in any manner, *i.e.*, there are no restrictions on how they are scheduled. Whenever the scheduler recognizes that two steps are incompatible the steps are scheduled such that their serializability is assured. Authors of [5] confine themselves to compatibility of steps of the same workflow. In general however, the scheduler must handle workflow instances of different workflow schemas whose steps may access the same data at a remote resource manager. The compatibility matrix must now be extended to include steps from all the workflow schemas. When this is the case, the number of step incompatibilities can be high and hence the approach may have to be refined. We return to this issue later in section 6.

The TSME system [16, 17], provides facilities for specifying workflow correctness requirements along with the workflow schemas using the Distributed Object Management (DOM) infrastructure [32]. Using the transaction specification language, dependencies can be specified between steps. Other than the state dependencies that specify a workflow structure, correctness dependencies can be specified to ensure one or more of the following: serializability, temporal correctness or cooperative correctness. The dependencies are specified in terms of the operations executed on objects. To ensure that the dependencies are satisfied, the system needs to determine the state of the objects themselves. Since DOM has mechanisms to track object accesses, dependencies can be enforced. Additional details of how TSME can be used for workflows can be found in [17, 15].

Concurrent execution of transactional workflows in discussed in [40], where a multidatabase approach is used to determine the correctness requirements for concurrent workflows. It views a workflow as a global transaction executing local transactions at different sites. Then it applies a relaxation to the global serializability requirements [18] for workflows using the correctness criterion of *M-serializability* defined in [39]. The workflow is divided into disjoint execution-atomic units, each consisting of related steps. The correctness criterion requires that steps belonging to the same execution-atomic unit of a workflow have compatible serialization orders at all local sites they access. A variation of the same scheme has been used to defined FT-serializability [24] as a correctness criterion for concurrent execution of Flex transactions [10] to implement telecommunication workflows.

5. Failure Atomicity of Workflows

Failure atomicity requirements of a workflow govern how and what changes made by the steps of a workflow are made persistent depending on the success of failure of a workflow. Traditional transactions use serializability as the correctness criterion and hence failure atomicity corresponds to the 'all-or-nothing' property, *i.e.*, if a transaction commits, all the changes made by the transaction are applied to the database and if a transaction aborts, none of the changes will be applied to the database. This criterion applies irrespective of whether a system failure or a logical failure occurs. However this notion is not suitable for workflows. In this section we survey some of the techniques that have been suggested to handle failures in workflows.

5.1. System Failures

System failures occur in a WFMS when one of the three components, *i.e.*, workflow engine, workflow database or the application agent fail. Since workflows typically contain a large number of steps, it is unacceptable to "undo" all the changes made by the workflow upto the point of the failure. Hence most WFMSs provide *forward recovery* which requires that every workflow be continued from the state of execution it was in before the failure instead of rolling back the entire workflow. Thus any useful work that has been done will not be lost. Now we survey some techniques used/suggested to ensure forward recovery in the event of the failure of each of the WFMS components.

When a workflow engine schedules a step for execution, it makes this fact persistent in the workflow database. Similarly, when a step completes, the results of the step are passed by the agent to the workflow engine which in turn records the information in the workflow database. This amounts to taking a persistent savepoint for each individual workflow. If a workflow engine fails, when it restarts after failure it obtains the state of the different workflows in progress by referring to the workflow database and continues their execution thus achieving forward recovery. It is necessary to ensure that the effect of "forward recovery" is achieved even if the workflow engine never restarts, *i.e.*, fails permanently. This implies that another workflow engine should be able to take over control of execution of all the workflows that were being handled by the failed workflow engine. This can be achieved via a clustered workflow engine architecture described in [2] where several workflow engines share a workflow database. In this scheme, if a workflow engine fails, the workflow instances controlled by it are handled by other workflow engines in the same cluster.

It can be observed that the workflow database is a crucial component in achieving forward recovery. It is also susceptible to failures, making it difficult to achieve forward recovery. Hence it is necessary to use fault tolerance techniques to replicate the state of the workflow database so that the workflows are continued from their present states even if there is a failure. Techniques and algorithms to achieve this efficiently are described in [25].

Application agents that supervise the execution of programs for performing the individual steps can also fail. This can cause problems for achieving forward recovery. Normally the program that performs the step runs on the same node as the application agent and hence if the node fails then both the agents and the program fail. However if only the agent fails and a program completes, the results returned by the program will be lost and there is no solution for this. Consider another scenario where the workflow engine fails followed by the failure of the agent. Although a program terminates successfully, the agent is unable to communicate the new state and the results produced to the workflow engine that has failed. Before the workflow engine restarts the agent fails. The workflow engine will try reconnecting to the agent. Unless proper care is taken, results of the steps that completed execution between the two failures will be lost. Essentially there is an inconsistency here since the state of the step as recorded in the workflow database will indicate that the step is still 'executing', even though the program and hence the step has completed. This problem can be ameliorated by logging significant events that happen at the agent. Hence every application agent should have logging facilities so that when a program completes, its return status code and data are logged [26]. Later the agent can pass on the results to the workflow engine. Again if the agent fails permanently, there is no solution to handle the situation. The step will be scheduled for execution by the scheduler again. There are issues of idempotency and we will discuss this scenario under logical failures.

5.2. Logical Failures

Logical failures (also termed as semantic failures in [1]) occur when a step cannot be executed successfully. This can happen for a variety of reasons. It may not be possible to successfully execute the program since an error occurred within the program or there were no resources available at the remote resource manager or the remote resource manager failed. A manual action has to be performed on behalf of the step may not be possible, *e.g*, sending a fax to a number that is incorrect. Logical failures also occur when a workflow has to be terminated due to an abnormal condition. An example is the case where a workflow that handles a customer order is terminated because the customer cancels an order.

To ensure that a workflow terminates in a proper state, it is necessary to precisely define whether the effect of a completed step should persist or be undone. Consider for example a workflow where one of several alternative paths [46] can be chosen at a decision point to achieve the same objective. After executing a few steps in the first choice, it may not be possible to complete that path due to a logical failure at a step (*i.e.*, that path cannot be used to meet the objective). Now the workflow will try to achieve the objective using the second choice and so on. However, it is important that steps that have been executed on paths that were unsuccessful be undone. This is usually achieved by *compensating* [19] the steps. Consider another scenario where a customer places an order and later cancels it. The action to be taken to handle a cancellation will very much depend on the state of the workflow at the time of cancellation. If all steps can be compensated then the entire workflow can be rolled back. However, in certain cases if vital steps of a workflow have already been executed, then it may not be possible to compensate them. An alternative action may be necessary. For example [15] describes an order cancellation scenario in a telecommunication service order provisioning workflow. Since certain facilities may already have been allocated for the customer, undoing the effect is complex and several choices are possible depending on which facilities have already been allocated. Thus, along with the workflow schema, it is necessary to explicitly state what action is to be taken (*i.e.*, what steps should be compensated and what additional steps need to be executed) if the workflow is canceled at any state (step). To facilitate this, the notion of *committed acceptable* and *aborted acceptable* termination states for a workflow have been proposed [40].

Given a transactional workflow specification, the set of acceptable states can be systematically determined using event algebra [42]. There are other notions such as *dead-path elimination* in FlowMark [11] which helps the workflow scheduler determine when a workflow is done. These techniques help the scheduler in determining when a workflow is considered complete or in an acceptable state. This is different from what we discussed in the previous paragraph that focused on pragmatic issues a workflow designer has to consider while specifying a workflow schema that can deal with logical failures.

A step can fail due to the failure of a program. However the agent may not be able to determine if the program failed before or after it met its objective. This is especially true for programs that access databases and is due to the *window* of time that exists between the actual commit of the transaction(s) by the remote resource manager and the instant when control returns from the program to the application agent. If there is a failure within this window, the transaction that executed on behalf of the program may have committed while this fact is not know to the agent. Hence one possible alternative is to have the remote database and the workflow database perform a two-phase commit to ensure that the result and the status of the activity is properly recorded in the workflow database. However, this if difficult to achieve due to the following problems. The first problem is that not all local resource managers provide the two-phase commit interface and even if they do, most do not yet conform with the XA interface standard proposed by X/Open [45]. The second problem exists because of legacy programs. Since transactions are bundled somewhere in the legacy code, it is not clear how many transactions each of these programs contain and what is the status of each when a failure occurs. Another possible alternative requires some guarantees from the program/remote resource manager. The program has to be implemented in an idempotent fashion. From the perspective of the agent, the program may be considered to have failed and the agent may execute the program again. For example, if a program that is implemented to order a part is not idempotent, then the same order could be placed twice. On the other hand if the program is implemented such that it is idempotent, then the program can be executed as many number of times until the agent has state information that the order has been placed. Thus if a program is implemented such that it guarantees idempotency, it is possible to handle failures of programs in a correct manner.

As we discussed earlier, most failures may not require the rollback of the entire workflow and a partial rollback may be sufficient. Hence, to provide more flexibility in defining the failure atomicity requirements of a workflow, the WFMS should provide the necessary modeling primitives. The execution support in the WFMS must ensure that the failure atomicity requirements are satisfied when a step fails or a workflow is terminated. One such facility has been developed for FlowMark in [30] and is based in the notion of spheres of joint compensation. A collection of steps in a workflow is grouped into a sphere S such that either all the steps of S complete successfully or all of them are compensated. Thus a sphere is basically a failure-atomic unit. Spheres can overlap and be nested. If a step fails, the sphere that immediately encloses it is compensated (sphere is backed out). Optionally, other spheres that enclose this sphere can be compensated and this can go on recursively (called cascaded backout). If a step is nested, the compensation can be deep, indicating the compensation of the individual sub-steps or shallow, indicating the compensation using a single step.

Earlier in our discussion on execution atomicity, we described how the invariant based approach is used in ConTracts to reduce the duration of locking. Now we discuss how the same approach can be used to ensure compensability [38]. The assumption that is made is that the prerequisites to execute a compensation-step are known when the corresponding step has been executed. Hence after the execution of a step, constraints can be established on shared resources such that the executability of the compensation-step is guaranteed. For example a customer may pay an advance of $1,000,000 to a company towards the processing of an order. Later if the customer cancels the order within the agreed terms, the advance has to be returned in full or part to the customer. Hence the company cannot use the advance until the order is confirmed. Hence a constraint can be established on the amount of money that can be used by the company (actual_available = total_available - $1,000,000) such that the advance can be returned to the customer (payment of advance is compensated) if needed.

Techniques for failure-handling in a nested hierarchy of workflows is discussed in [7]. The rollback of a step at a given level may or may not affect its parent step. If it does, then the parent is to be rolled back and the procedure is repeated until a parent step is reached that does not need to be rolled back. This happens when a parent is not affected by its childs' step. From that point on, a parent step may try an alternative child step. Then it discusses a two-phase remedy to handle a logical failure where the first phase called the bottom-up phase determines the highest ancestor step affected by the failure of the current step and a second phase called the top-down phase undos the changes at each level starting from that ancestor. These failure-atomicity techniques are ideal for workflows in engineering environments.

Even with all the support and specification for automatically dealing with logical failures, sometimes human intervention may be required. Hence most WFMSs support dynamic modification of workflows. Note that this modification is carried out at the workflow instance level and not at the level of the workflow schema. However the human handling the modification must follow some guidelines (similar to those we have been discussing) to ensure that the required tasks are properly executed or compensated to handle the specific scenario.

6. Discussion and Open Issues

All the techniques described to handle execution and failure atomicity are primarily implemented by the workflow scheduler with the support of the agents where the steps are performed. Hence apart from the satisfying the data and control flow requirements, the scheduler must ensure that all the correctness requirements are satisfied as well. In particular, the scheduler has to determine which steps have completed, which steps have failed and which steps have to be compensated. Thus the scheduler has to deal with enormous state information especially when several thousand instances of workflows are executing concurrently. Scheduling related issues are discussed further in [3, 40, 43, 42, 20].

In section 4, we presented the execution atomicity requirements in the presence of concurrent workflows as discussed in [44, 5, 40]. The invariant based approach of ConTracts [44] can be used to ensure executability of steps when other steps from concurrent workflows can access the same data item. However data inconsistency can be caused due to improper interleaving of two or more steps from different workflows. A different approach is needed to handle such situations. In [5], the interleaving dependency is specified using a compatibility matrix and the scheduler refers to this matrix to ensure correctness. Although the compatibility matrix is defined for only one workflow, in a real system, data sharing occurs between steps of different workflow schemas. Hence the compatibility matrix is to be defined potentially to cover all the steps of all the workflow schemas. In this situation, several steps may wait/block to be interleaved in the appropriate manner, thereby reducing the number of acceptable schedules. This situation can be ameliorated by exploiting additional semantics about the steps and the workflow. Specifically, using information about the input and output parameters of a step and utilizing data/control flow information within workflows, it is possible to reduce the number of steps that might be blocked to ensure correctness, thus allowing more schedules. In [40], the M-serializability criterion described in the context of multidatabase system [39] is used for handling interleaving of concurrent workflows. Here the system ensures that the execution order of conflicting steps belonging to the same execution-atomic units of two workflows have the same serialization order at every local site. However, steps of a general workflow can potentially be accessing non-transactional resource like flat files or spreadsheets apart from database transactions and the steps commit independently. Interleaving dependencies are also important when multiple such workflows execute concurrently and a different approach is needed.

The failure atomicity requirements of workflows focus on the correctness of individual workflows in the presence of failures. For example, in section 5, we discussed the techniques presented in [30] that ensure partial rollback requirements of individual workflows. However the effect of rollbacks in one workflow on the forward/rollback execution of other concurrent workflows has received little attention. There has been some work studying the data consistency issues when compensations are performed in the presence of concurrent transactions [28]. This work has been later extended to deal with sub-transactions from global transactions in a multidatabase environment [29, 35]. They consider steps of three types — compensatable (steps whose effects can be undone), retriable (steps that are guaranteed to be successful when tried repeatedly) and pivot (steps that are neither compensatable nor retriable). A criterion called *serializability with respect to compensation (SRC)* is defined in [35] which precludes a multidatabase transaction from observing the changes made by another transaction only at some of the sites even though they conflict at more sites. This situation occurs when some of the sub-transactions (say of transaction T1) are committed and are later undone due to the abort of other sub-transactions. In the meantime, conflicting sub-transactions (from transaction T2) could have interleaved with the sub-transactions from T1. SRC prohibits sub-transactions of T2 from seeing committed states of some sub-transactions of T1 and aborted states of other sub-transactions of T1. Although the issues are relevant to workflows, SRC may be too strict. Also compensatable steps themselves are of different types — logically compensatable and physically compensatable. Physically compensatable refers to installing the before image of the entire object. This is relevant to compensation/undo of changes on flat files and spreadsheets since they do not have a transaction manager that handles concurrency and recovery. The type of a step largely determines in some sense the effect of rollback of one workflow on another. Using alternative subtransactions and the notion of semi-atomicity (global transaction is allowed to commit different parts at different times), more resiliency can be achieved in handling the failure of sub-transactions with respect to an individual flexible transaction

[46]. In this approach, after the execution of a pivot, alternative functional paths are executed such that one of them will commit and the effects of unsuccessful paths are completely undone (compensated). But the requirements in the presence of concurrent workflows can be very complex. For example, due to data sharing between workflows, the execution of a pivot step in one workflow can affect the rollback of a concurrent workflow. The invariant based approach for ensuring compensatability (ability to rollback steps of a workflow) [44] is useful in situations where it suffices to ensure that the constraints hold irrespective of the type of step accessing the data. However, when workflows containing pivot steps execute concurrently a different approach is needed.

From the above discussion it is clear that there is a need to (i) determine the correctness requirements in the presence of concurrency and failures for the execution of general workflows whose steps commit independently, and (ii) develop suitable mechanisms for ensuring the correctness requirements. These issues are being addressed in the context of multiple workflows in [27].

7. Summary

Workflow management offers a powerful technique to integrate and automate the different tasks of an enterprise. However most commercial WFMSs provide little or no support for ensuring correctness of execution of workflows and this is a major limitation especially if WFMSs are used to run the critical business processes in an enterprise. This paper provides an overview of the correctness issues in workflow management.

A step receives data from one ore more other steps within a workflow, and often a program that executes on its behalf accesses shared data from a remote resource manager. Since there is inter- and intra- workflow sharing of data, proper techniques are needed to ensure data consistency. Hence most of the issues we investigate fall under the broad categories of execution atomicity and failure atomicity. We further differentiated failure atomicity requirement into those that arise from system failures and from logical failures. We then discussed several schemes from the literature that address these requirements. A lot of the techniques surveyed are from the domain of transactional workflows and the solutions often take the approach adopted by advanced transactions. Although several of these techniques provide insights into how correctness can be ensured, not all can be directly used in general WFMSs where steps commit independently. Also existing techniques that address the effect of rollbacks (due to logical failures) from within a workflow on other concurrent workflows have to be developed further. Hence we enumerated some open issues related to concurrent execution of workflows in the presence of failures.

It should be emphasized that the schemes studied in this paper are primarily concerned with transactional workflows, and additional research needs to be done to incorporate non-transactional objects and executions. Some of these issues are being addressed in [27].

A proper understanding of the concepts and techniques related to preserving correctness in WFMSs by both WFMS developers and workflow designers is necessary. This will help in building workflows that are flexible enough to capture the requirements of real world applications and robust enough to provide the necessary correctness and reliability properties in the presence of concurrency and failures. Through this paper we have attempted to achieve this objective.

References

[1] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Guenthoer, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1996.

[2] G. Alonso, M Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Technical Report RJ 9913(87293), IBM Almaden Research Center, 1994.

[3] P. C. Attie, M. P. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proc. Intl' Conf. on Very Large Data Bases*, page 134, Dublin, Ireland, August 1993.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1987.

[5] Y. Breitbart, A. Deacon, H.J. Schek, A. Sheth, and Weikum. G. Merging Application-Centric and Data-Centric Approaches to Support Transaction-Oriented Multi-system Workflows. *ACM SIGMOD Record*, 22(3), 1993.

[6] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 1(2):181–240, October 1992.

[7] Q. Chen and U. Dayal. A Transactional Nested Process Management System. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1996.

[8] C. T. Davies. Data Processing Spheres of Control. *IBM Systems Journal*, 17(2):179–198, 1978.

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

[10] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 507, Brisbane, Australia, August 1990.

[11] Leymann F. and Roller D. Business Process Management With FlowMark. In *Proceedings of IEEE CompCon*, pages 230–234, 1994.

[12] A. A. Farrag and M. T. Ozsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Trans. on Database Sys.*, 14(4):503, December 1989.

[13] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Trans. on Database Sys.*, 8(2):186, June 1983.

[14] H. Garcia-Molina and K. Salem. Sagas. In *Proc. 1987 SIGMOD International Conference on Management of Data*, pages 249–259, May 1987.

[15] D. Georgakopolous, M. Hornick, and A. Sheth. An Overview of Workflow Management: From

Process Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–152, 1995.

[16] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 462, Houston, TX, February 1994.

[17] D. Georgakopoulos, M. F. Hornick, and F. Manola. Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation. *IEEE Trans. on Knowledge and Data Eng.*, 1995.

[18] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 314, Kobe, Japan, April 1991.

[19] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 144, Cannes, France, September 1981.

[20] R. Günthör. The Dependency Manager. In *RIDE-NDS (Interoperability of Nontraditional Database Systems), New Orleans, Louisiana*, 1996.

[21] D. Hollingsworth. Workflow Management Reference Model, 1994. The Workflow Management Coalition, Accessible via: http://www.aiai.ed.ac.uk/WfMC/.

[22] M. Hsu. Special Issue on Workflow and Extended Transaction Systems. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 16(2), 1993.

[23] M. Hsu. Special Issue on Workflow Systems. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 18(1), 1995.

[24] W.W. Jin, M. Rusinkiewicz, L. Ness, and A. Sheth. Concurrency Control and Recovery of Multidatabase Workflows in Telecommunication Applications. In *Proc. ACM SIGMOD Conf.*, pages 456–459, Washington DC, June 1993.

[25] M. Kamath, G. Alonso, R. Günthör, and C. Mohan. Providing High Availability in Workflow Management Systems. In *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT-96)*, Avignon, France, March 1996.

[26] M. Kamath and K. Ramamritham. Modeling, Correctness & Systems Issues in Supporting Advanced Database Applications using Workflow Management Systems. Technical Report TR 95-50, University of Massachusetts, Computer Science Dept., 1995.

[27] M. Kamath and K. Ramamritham. Mechanisms for Ensuring Correctness of Workflows in the presence of Concurrency and Failures. Technical Report In preparation, University of Massachusetts, Computer Science Dept., 1996.

[28] H. F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 95, Brisbane, Australia, August 1990.

[29] E. Levy, H. Korth, and A. Silberschatz. A Theory of Relaxed Atomicity. In *Proc. of ACM SIGACTS-SIGOPS Symposium on Principles of Distributed Computing*, 1991.

[30] F. Leymann. Supporting Business Transactions via Partial Backward Recovery in Workflow Management. In *Proc. of BTW'95*, Dresden, Germany, 1995. Springer Verlag.

[31] N. A. Lynch. "multilevel atomicity: A new correctness criterion for database concurrency control". *ACM Transactions on Database Systems*, 8(4), December 1983.

[32] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. Distributed Object Management. In *Int. J.of Intelligent and Cooperative Information Systems 1, 1*, March 1992.

[33] D.R. McCarthy and S.K. Sarin. Workflow and Transactions in InConcert. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[34] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The Concurrency Control Problem in Multidatabases: Characteristics and Solutions. In *Proc. ACM SIGMOD Conf.*, page 288, San Diego, CA, June 1992.

[35] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. A Transaction Model for Multidatabase Systems. In *Proc. 12th Int'l. Conf. on Distr. Computing Sys.*, page 56, Yokohama, Japan, June 9-12 1992.

[36] C. Mohan. State of the Art in Workflow Management Systems Research and Products, 1996. Tutorial presented at ACM SIGMOD International Conference on Management of Data, 1996.

[37] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, PhD Thesis, MIT, Cambridge, MA, April 1981.

[38] A. Reuter and F. Schwenkreis. ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems. *Bulletin of the Technical Committee on Data Engineering*, 18(1), March 1995. IEEE Computer Society.

[39] M. Rusinkiewicz, A. Cichocki, and P. Krychniak. Towards a Model for Multidatabase Transactions. *International Journal of Intelligent and Cooperative Information Systems*, 1(3):579–617, 1992.

[40] M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond. ACM Press*, 1994.

[41] A. Sheth and M. Rusinkiewicz. On Transactional Workflows. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[42] M. P. Singh. Synthesizing Distributed Constrained Events from Transactional Workflow Specifications. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1996.

[43] M.P. Singh, L.G. Meredith, C. Tomlison, and P.C. Attie. An Algebraic Approach for Workflow Scheduling. Technical Report Carnot-049-94, MCC, 1994.

[44] H. Waechter and A. Reuter. The ConTract Model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, San Mateo, 1992.

[45] X/Open, editor. *Distributed Transaction Processing: The XA Specification.* X/Open Company Limited, UK, 1992.

[46] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proc. 1994 SIGMOD International Conference on Management of Data*, pages 67–78, 1994.