Failure Handling and Coordinated Execution of Concurrent Workflows*

Mohan Kamath† and *Krithi Ramamritham*
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
(kamath,krithi)@cs.umass.edu

Abstract

Workflow management systems (WFMSs) coordinate the execution of applications distributed over networks. In WFMSs, data inconsistencies can arise due to (i) the interaction between steps of concurrent threads within a workflow (intra-workflow coordination) (ii) the interaction between steps of concurrent workflows (inter-workflow coordination) and (iii) the presence of failures. Since these problems have not received adequate attention, this paper focuses on developing the necessary concepts and infrastructure to handle them.

First, to deal with inter- and intra- workflow coordination requirements we have identified a set of high level building blocks. Secondly, to handle failures we propose a novel and pragmatic approach called opportunistic compensation and re-execution that allows a workflow designer to customize workflow recovery from correctness as well as performance perspectives. Thirdly, based on these concepts we have designed a workflow specification language that expresses new requirements for workflow executions and implemented a run-time system for managing workflow executions while satisfying the new requirements. These ideas are geared towards improving the modeling and correctness properties offered by WFMSs and making them more robust and flexible.

1 Introduction

Workflow management systems (WFMSs) coordinate the execution of applications distributed over large networks. *Workflow management* has emerged as the technology to automate the execution of the many steps (applications) that comprise a business process [13,17,22]. Business processes often consist of applications that access shared resources like databases. Hence dependencies arise due to the interaction between steps of concurrent workflows. Events happening in one workflow can potentially cause data inconsistencies in other concurrent workflows. Thus a workflow designer should have the capability to express coordinated execution requirements across workflows. Also, due to the complicated nature of business processes, several threads can be executing concurrently within a workflow. It is essential to handle data inconsistencies arising due to such concurrent executions and due to failures. Recovering from failures by compensating (reversing the effect of) all the completed steps of a workflow and aborting the workflow may be very conservative and sometimes impractical. Strategies such as spheres of joint compensation [21] are geared specifically towards workflow rollbacks but do not deal with the effects of the rollbacks on the re-execution of workflows. Hence an integrated approach is needed to handle workflow rollback and re-execution. Such an approach will allow a workflow designer to customize workflow recovery with respect to both *correctness* and *performance*. So far, adequate attention has not been paid to many issues related to correctness [20].

To support failure handling and coordinated execution of workflows, as part of the CREW (Correct & Reliable Execution of Workflows) Project at UMass, we have developed the necessary concepts and infrastructure. To handle coordinated execution requirements, we have identified high level building blocks that express *mutual-exclusion* and complex *ordering* requirements across workflow steps, and *rollback dependency* across workflow instances. These building blocks can be used to specify both inter- and intra- workflow coordination. To handle failures within a workflow, we have proposed a new scheme called *opportunistic compensation and re-execution* which eliminates unnecessary recovery overheads when workflows are rolledback partially and re-executed. To customize the order in which the steps are compensated in this scheme, workflow designers can also specify *compensation dependent sets.*

Incorporating the above concepts, we have developed a workflow specification language called LAWS which allows the specification of failure handling and coordinated execution requirements. We use a rule-driven approach to execute the steps in a workflow. Hence, we have developed a compiler to translate all the high level requirements expressed in LAWS into a uniform set of rules. These rules can be dynamically modified, as steps execute (and failures occur). To this end, we have identified a small set of implementation level primitives *AddRule(), AddEvent() and AddPrecondition()*, sufficient enough to realize the high level coordinated execution and failure handling requirements. Finally, to demonstrate the usefulness and practicality of our approach, we have implemented a prototype system.

The rest of the paper is organized as follows. Essentials of workflow management are discussed in Section 2. Section 3 provides the motivation for this work. An overview of our approach for failure handling and coordinated execution is outlined in section 4. Sections 5 through 7 form the crux of the paper. We describe the LAWS language in Section 5. Compile-time actions are discussed in Section 6 and run-time support is discussed in Section 7. Related work is discussed in Section 8 and Section 9 summarizes the paper and discusses future work.

2 Workflow Management Systems

As described by the Workflow Management Coalition (WfMC) [16], a WFMS consists mainly of a workflow engine and application agents. In addition, there are other components for modeling, administration and monitoring. The workflow engine and the tools communicate with a workflow database (WFDB) to store and update workflow state data. The components are connected as shown in *Figure 1.* A workflow schema (workflow definition) is created by a workflow designer with the aid of the modeling tool. A workflow schema is essentially a directed graph with nodes representing the steps to be performed. A step is performed by typically executing a program that accesses a database/document or enacting some form of human action. The arcs connecting the steps are of two types: data and control arcs. A data arc denotes the flow of data between steps while a control arc specifies the ordering sequence between two steps. The latter may also have a condition associated with it to indicate that the succeeding step is

*This material is based upon work supported by the National Science Foundation under grant IRI - 9619588.

† *Current Address:* Oracle Corporation, 500 Oracle Pkwy, Redwood Shores CA 94065 email: mkamath@us.oracle.com

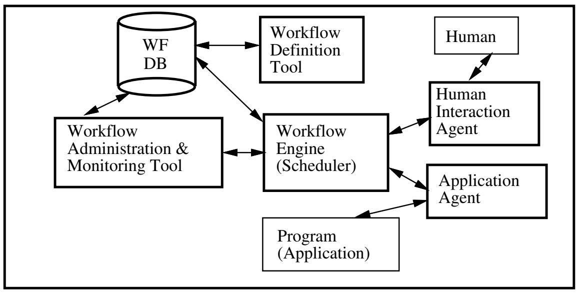

Figure 1: *Components of a Workflow System*

to be executed only if the condition evaluates to true.

The program associated with a step (a transaction, in transactional resource managers) and the data that is accessed by the step are not known to the WFMS and hence the steps themselves are 'black boxes' as far as the WFMS is concerned. A WFMS does not have access to step related information embedded in resource managers accessed by a program executed to perform a step. Hence, a WFMS needs additional information to determine for example if two steps from different workflows could have accessed the same resources in a conflicting manner.

Using a workflow schema as a template, several instances of the workflow can be executed. The workflow engine is responsible for managing all the instances of all schemas, *i.e.*, it has to schedule the steps of the instances when they are eligible for execution. The workflow engine provides the agent with the information required to execute a step. The agent is responsible for executing the step and communicating back the results of the step to the engine. The WFDB provides the persistence necessary to facilitate forward recovery in case of failure of the workflow engine.

Failures can occur in workflow system components and they have to be handled efficiently to improve WFMS availability. Strategies for handling failures of the workflow server are discussed in [3] and for the failure of the workflow database are discussed in [19]. In the rest of the paper we will be dealing with logical failures, *i.e.*, failures (unsuccessful execution) of the steps in a workflow.

3 Motivation

In this section we will motivate some of the failure handling and coordinated execution problems that require special attention.

3.1 Coordinated Execution of Workflows

The need for coordinated execution of workflow steps arises from *application* requirements as well as *data consistency* requirements.

For example, steps from two different workflows may need to be executed in a *mutually exclusive* fashion if the resource managers accessed by the applications performing the respective steps do not have concurrency control (e.g., non-transactional resource managers) and the applications access some resources in a conflicting mode. This can happen for example if applications are reading/updating file(s) served by a NFS file system. While one option is to enhance the lower layers of the system to add a layer that manages concurrent accesses, another is to specify a mutual exclusion requirement at the workflow specification level and let the workflow system handle it. Note that this requirement arises from the perspective of data consistency. Although the tradeoff between the two alternatives may depend on the specific setting it is definitely desirable to have a general purpose facility for supporting mutual exclusion of steps.

Consider another example as shown in *Figure 2(a)* where steps (S12/S23 and S14/S25 respectively) from concurrent workflows (WF1 and WF2) that access common resources are to be executed in the same *relative order*. Such situations

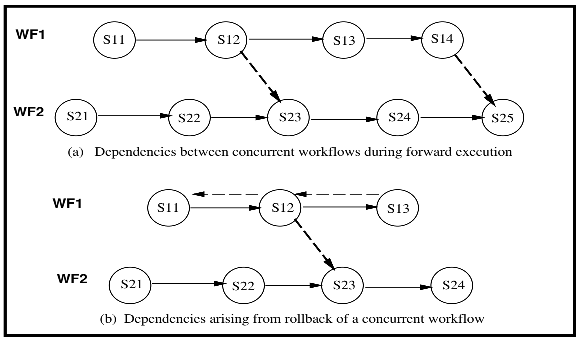

Figure 2: *Dependencies Across Workflows*

arise for example in order processing workflows where orders are to be fulfilled in the sequence in which they were received. This implies that steps from two order processing workflows that conflict (*e.g.*, ordering the same parts or requiring the same machines) are to be executed in the same relative order. If the steps are allowed to interleave freely, a workflow processing an earlier order may not be able to continue due to lack of resources (which were used by a concurrent workflow processing a later order). Note that this is an *application* requirement since interleaving of steps will not cause data inconsistency. Relative ordering of conflicting steps from a pair of workflows is analogous to serialization of conflicting object accesses from a pair of transactions.

Figure 2(b) shows an example where two steps S12 and S23 access common (conflicting) resources. Consider a rollback of workflow WF1 to handle the failure of step S13. This failure may require rollback of steps S12 and S11. Workflow WF2 now continues execution using data produced by S23 that may be incorrect since it was originally produced after the execution of step S12 which has subsequently been compensated. Hence a corrective action such as a rollback and re-execution may be needed in WF2. This is what we refer to as a *rollback dependency* between steps of concurrent workflows. The same figure can be used to illustrate another scenario. Workflows WF1 and WF2 are used for order processing and steps S12 and S23 are ordering the same components but only step S12 is able to fulfill its order since it executed first. Execution of step S23 can be delayed until the final outcome of step S12 is known. If S12 is rolledback S23 may be able to complete. Otherwise, if WF1 completes and S12 is not rolledback, WF2 may have to be aborted.

It should be possible to specify these correctness requirements at the workflow schema level. To the best of our knowledge, current commercial and prototype WFMSs neither have the provision to specify nor to enforce coordinated execution requirements across multiple workflow steps.

3.2 Failure handling in Workflows

In the last section, we alluded to some implications of coordinated execution on failure handling. Now we focus on the effect of failures on steps within a workflow.

Consider the example shown in *Figure 3* where a step fails in a workflow with if-then-else branching. Suppose the top branch in taken after step S2 during the first execution thread and step S4 fails. The failure handling specification may require the workflow to partially rollback to step S2 and re-execute step S2. During the next execution because of changes in the data produced by step S2 (or from a feedback of failure at S4) the bottom branch may be taken. Essentially a branch that is different from the previous execution is taken. Now the effect of step S3 has to be undone. *When* S3 will be compensated depends on S3's data dependency with respect to steps S2 and S5. If there is no data dependency with either of the steps then S3 can be compensated anytime before the workflow completes, including compensating it

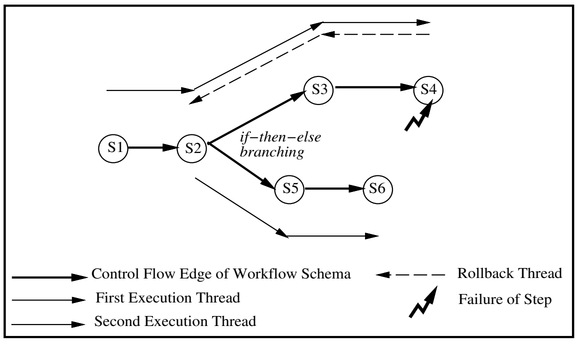

Figure 3: *Rollback in a Workflow*

in parallel with step S5's execution. If it has a dependency with respect to step S5 then S3 has to be compensated before the execution of step S5. These are complex rollback and compensation requirements and providing a default semantics is not enough. Hence the extended transaction model (say, Sagas [12]) based approach requiring the compensation of all previously executed steps and aborting the workflow may be impractical in several scenarios. Also since the steps of workflows are often loosely related it is not necessary to always compensate the steps in the reverse execution order. Instead a workflow designer should be able to specify such rollback requirements based on dependencies and the business logic. What we just discussed is an example of workflow recovery customization based on *correctness* requirements. *Performance* is also an important issue in handling workflow recovery. Often in the context of partial rollback and re-execution it may not be necessary to compensate and re-execute some of the steps since they do not produce any new results or the previous results may prove to be sufficient. In such cases results from the previous execution of a step can be used rather than compensating and re-executing the step again. This can result in substantial savings especially if the steps involve the use of expensive human or computer resources. Current WFMSs do not have provision to support such requirements.

Another important factor to consider is the interaction between forward execution of steps and rollbacks within a single workflow. Inconsistencies can result from these interactions. For example, consider *Figure 4* which shows the different threads of forward execution and rollback in a workflow and illustrates a race condition that develops at *confluence steps* (steps where two or more branches meet) due to failure. After step S2 is executed, control is passed along both the branches (AND branching), resulting in two concurrent threads of execution. If the execution of the different threads within a workflow is not tracked, data inconsistencies can arise. In the example, one of the two threads of forward execution (top thread in the figure) fails at step S4. The failure specification requires that

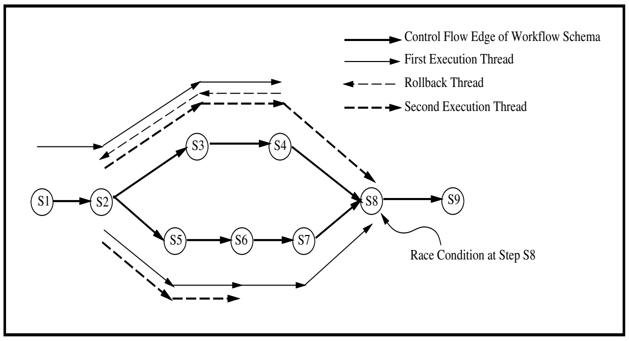

Figure 4: *Race Condition arising in a Workflow*

the thread be rolledback upto step S2 (compensating steps S3, S2) and restarted from step S2. Meanwhile, control in the other thread (bottom thread in the figure) has already reached the confluence step S8. After step S2 is executed now, two new concurrent threads of execution start along both the branches. The new thread of control along the top path may reach step S8 before the bottom thread. In this case, step S8 will be triggered using information from *different* threads of execution — the old thread from the bottom path and the new thread from the top path. The data input to step S8 may be incorrect since the two threads flowing into step S8 used information from *different executions* of step S2. Hence, the interaction of forward execution and rollbacks within a workflow can cause data inconsistencies. This problem is relevant to customized transaction models [11] as well.

4 Overview of Our Approach

In this section we briefly describe the CREW approach to handle the different problems we motivated in the previous section. Details of our approach are presented in the rest of the sections of the paper.

Support for workflow applications has been addressed by researchers focusing on workflow systems and transaction systems. *Extended transaction systems* structure a large transaction into sub-transactions and execute them with additional precedence requirements between start/commit/abort of the individual sub-transactions. In practice they have found limited applicability since not all steps that make up a workflow are transactions. On the other hand, traditional WFMSs just allow the specification of step level details and data/control flow requirements in a workflow schema. Our approach falls in the category of *transactional workflows* [23] where additional correctness requirements can be specified on top of traditional workflows specifications. These requirements specify additional constraints on workflow execution schedules. To specify *how* failure handling and coordinated execution is to be performed we provide additional capability in our workflow specification language.

To handle coordinated execution requirements, we have identified high level building blocks that express *mutual-exclusion* and complex *relative-ordering* requirements across workflow steps, and *rollback-dependent execution* across workflow instances. Details of how these primitives are used can be found in section 5. Step executions in workflows are typically triggered by the completion of one or more previous steps or the occurrence of specific events and conditions. Hence, we use a rule based approach to manage the execution of workflows at run-time. The high level building blocks are dynamically translated into low level implementation primitives. We have identified a small set of implementation level primitives *AddRule(), AddEvent() and AddPrecondition()*, that affect the rule sets dynamically. These low level primitives are often used to exchange information across rule sets belonging to different workflow instances. Note that the messages exchanged are *logical* and not physical. The message is passed from a workflow instance to the engine which in turn passes it to the relevant instances.

To handle failures within a workflow, we propose a new scheme called *opportunistic compensation and re-execution* which eliminates unnecessary compensation and re-execution of steps when workflows are rolledback partially and re-executed. Two types of compensation are possible — *complete* and *partial* and two types of re-executions are possible — *complete* and *incremental*. If the previous execution of a step is useless in the current context, then a complete compensation and complete re-execution may be necessary. On the contrary, in cases where the previous execution of the step is useful an incremental re-execution or a partial compensation of the step may be sufficient to produce an effect equivalent to the complete compensation and re-execution of the step with

```
if (step has been compensated)
   execute step ;
elseif (incremental re-execution condition = TRUE)
   modify inputs ; execute step ;
elseif (partial compensation condition = TRUE)
   modify inputs ; compensate step ;
elseif (normal re-execution condition = TRUE)
   if (step ∈ compensation dependent set)
      compensate CD set ; execute step ;
   else
      compensate step ; execute step ;
else
   proceed to next step ;
```

Figure 5: *Opportunistic Compensation and Re-execution*

the new inputs. The partial compensation and incremental re-execution strategy is particularly useful in order processing workflows, for example, where the quantity of items required in a step has changed. To customize the order in which the steps are compensated in this scheme workflow designers can also specify *compensation dependent (CD) sets.* A CD set specifies a group of steps in which the compensation of one affects the others. A CD set is to be compensated only in the reverse execution order of its member steps. Note that if a step's execution has already been compensated (based on the requirements specification or the steps are part of a compensation dependent set) and if the reexecution thread passes through that step then it has to be re-executed – irrespective of the evaluation of the re-execution condition. The notion of CD sets is different from the notion of spheres of joint compensation [21]. The former is used to specify the order of step compensation in the context of our opportunistic compensation and re-execution strategy. The semantics of the latter is that when a step belonging to a sphere fails, immediately all other steps that belong to the sphere are compensated. Thus there is no integrated treatment of rollback and re-execution in the context of the latter. The algorithm for supporting opportunistic compensation and re-execution is shown in figure 5.

To handle the race-condition described in *Figure 4* the following approach is used within a workflow. The *rollback dependency* building block is translated in such a way that one execution thread notifies another in case of a failure. Hence, if a step fails in one thread and its rollback requires that the branching step be also compensated then other concurrent threads that originated from the branching point are halted. If the branching point is not compensated, execution continues along the threads. If a thread has already reached a confluence step its effects are re-evaluated and modified by our opportunistic compensation and re-execution strategy as a new re-execution thread traces that path.

Details of our workflow specification language are provided in section 5. In that section we discuss how the higher level building blocks for coordination are used and also how opportunistic compensation and reexecution as well as compensation dependent sets are specified. The compilation process that translates these requirements into lower level implementation primitives is described in section 6. Additional details of implementation of the low level primitives is provided in section 7. Additional details of the techniques presented in this paper can be found in [18].

5 Language Support

In this section, we describe the different components of LAWS (LAnguage for Workflow Specification). Apart from the ability to specify traditional workflow schemas, LAWS has the capability to capture/express the different failure handling and coordinated execution requirements of workflows at a high level.

5.1 Mandatory Specification

The following components provide the ability to specify traditional workflow schemas. These are *mandatory*.

1. *Workflow Specification*: This specifies the name of the workflow and the input/outputs to the workflow. An instance of a workflow is identified by the WFMS using the workflow name and the instance number.
2. *Step Specification*: This specifies the details of the individual steps including the name of the node(s) (an agent should be running on each specified node) where the step is to be executed, the name of the program that is to be executed to perform the step,the inputs and outputs to the step. Optionally, the name of the program that is to be executed to compensate (where possible) the effects of that step can also be specified.
3. *Data-Flow Specification*: This specifies the data flow between workflow inputs/outputs and step inputs/outputs. Data for a step is obtained either from the inputs to the workflow or the outputs of preceding steps of the workflow.
4. *Control-Flow Specification*: This provides the navigational specification of workflows, *i.e.*, how control flows within a workflow. The data flow specification places a partial ordering on the execution sequence of the steps, the control flow specification adds further restrictions on this partial ordering.

5.2 Failure Handling and Coordinated Execution Specification

The following components are *optional* and are used to specify the failure handling and coordinated execution requirements of workflows. Items 5, 6 and 7 correspond to the different specifications required to realize our opportunistic compensation and re-execution strategy. Items 8 and 9 correspond to the specifications required to realize coordinated execution of workflows.

5. *Failure Handling Specification*: This specifies how failures of individual steps of a workflow are to be handled. It indicates the step to which the workflow is to be partially rolledback and whether the workflow is to be *re-started* from there or *aborted*. The following states that *on the failure of step* S7, *compensate step* S2 and *re-start the workflow beginning at step* S2:
   S7: COMPENSATE S2 RE-START S2 ;.
   If a step does not have a failure handling specification, a default action of workflow abort is assumed, *i.e.*, if that step fails then the workflow is aborted.
6. *Step Re-Execution Specification*: This specifies if a step is to be re-executed in the case of a workflow being re-started after partial rollback. The decision to completely compensate and re-execute, incrementally re-execute or partially compensate a step is made based on *conditions*. The conditions are predicates on the current workflow state which includes the step outputs of the preceding steps in the workflow and the old inputs (step input used for the previous successful execution of the step) to the step. Let us consider a step S2 that has two inputs I1 and I2. The following states that step S2 is to be incrementally re-executed if the *new* value of input I1 exceeds its *old* value (from previous execution). Note that the new input is modified based on the old input. S2 is to be partially compensated if the *new* value of input I1 is less than its *old* value. Here again the input value is changed so that partial compensation is performed to reflect the differences in the previous and new input. Finally, the step is to be completely compensated and re-executed only if the new

value of input I2 is different from the old one.

```
S2: IF (I1.NEW > I1.OLD)
      THEN MODIFY (I1.NEW = I1.NEW - I1.OLD) RE_EXECUTE
   ELSEIF (I1.OLD > I1.NEW)
      THEN MODIFY (I1.NEW = I1.OLD - I1.NEW) COMPENSATE
   ELSEIF (I2.NEW != I2.OLD) COMPENSATE RE_EXECUTE
```

Note that if a step re-execution specification does not exist for a step then it is assumed that the step need not be compensated and re-executed if the workflow is partially rolled back and re-started. Alternatively, if the specification explicitly states that the condition for re-execution is 'ALWAYS', then the step is re-executed *every time* the workflow restarts and the execution thread passes through that step. When an explicit condition exists, then the step is compensated/re-executed only if the condition evaluates to TRUE. Also, if the condition is 'INPUTS_CHANGED', it evaluates to true if *any* of the new inputs to the step have values different from their old values. Note that the step specification contains the name of the program to be used for compensating a step.

7. *Compensation Dependent Set*: This specifies a set of steps that have compensation dependency with reference to one another. A compensation dependent set is to be compensated only in the reverse execution order of its member steps. The following states that steps S3 and S6 are compensation dependent which means that S6 is to be compensated (if it has been executed) before the compensation of step S3.

```
CD1: S3 S6 ;
```

If no compensation dependent sets are specified, a step is compensated if required just before its re-execution using the opportunistic compensation and re-execution strategy.

8. *Step Conflict Specification*: This specifies conflicts between steps of one workflow with steps from the same or other workflows. Since the workflow designers have more information about the programs and their role in a specific workflow, they can specify the conflict conditions for a pair of steps within or across workflows. Two steps are considered to conflict if a given condition evaluates to TRUE. Step conflict specification need be given only for those that may access the same resource. For example, step conflicts can be defined independently for (i) steps accessing the design database (ii) step accessing the inventory database etc. An example of step conflict specification is:

```
IF (S12.I1 == WF2.S23.I2)
   THEN S12 CONFLICTS_WITH WF2.S23 ;
```

which states that step S12 of the current workflow conflicts with step S23 of workflow WF2 is the condition (S12.I1 == WF2.S23.I2) evaluates to TRUE. The workflow under consideration and workflow WF2 are referred to as conflicting workflows.

9. *Coordinated Execution Specification*: This specification defines inter-workflow coordinated execution requirements. Note that the same primitives can be used to specify intra-workflow coordinated execution requirements as well. There are two types of specification (both types of specifications can have complex conditions attached to them):

The first specifies how actions in a workflow should be performed with respect to actions in a conflicting concurrent workflow. This helps a designer for example to specify if steps from concurrent workflows are to be executed in a *mutually exclusive fashion* or if the steps should satisfy the *relative ordering* requirement. Event comparison predicates compare the ordering of events from the concerned workflows using the HAPPENED_BEFORE, HAPPENED_AFTER and RELATIVE_ORDER operators. An example of this type of specification (corresponding to *Figure 2(a)*) is:

```
IF ((S12 CONFLICTS_WITH WF2.S23) AND S12 RELATIVE_ORDER WF2.S23
   AND (S14 CONFLICTS_WITH WF2.S25))
 THEN S14 RELATIVE_ORDER WF2.S25 ;
```

The second specifies the action to be taken in the current workflow based on the occurrence of specific event(s) in conflicting workflows. This helps a designer to specify *rollback dependency* between steps of conflicting workflows. An example of this type of specification (corresponding to *Figure 2(b)*) is:

```
ON WF1.S12.COMPENSATE
IF (S23 CONFLICTS_WITH WF1.S12) AND (S23 HAPPENED_AFTER WF1.S12 )
   THEN COMPENSATE S23 RE-START S23 ;
```

Note that the conflict definitions specified in the step conflict specification are used in the condition part of these specifications.

5.3 Discussion

As discussed in section 4 the goal of our work is to provide modeling capabilities at the workflow level to specify constraints on step execution based on dependencies between steps and business requirements. The WFMS has no knowledge of the internals of the programs representing the steps. Database transactions can automatically ensure consistency using concurrency control schemes since they know the records/objects being accessed but it is not possible for a WFMS to do so because of the presence of legacy components and autonomous resources. Hence, it is upto the designer to take advantage of the modeling capabilities to specify failure handling and coordinated execution requirements in a correct manner.

Let us specifically focus on the requirements of coordinated execution of workflows. Our approach has similarities with the compatibility matrix based approach for concurrency control in object-oriented database systems. Just as an object designer specifies how the methods of the different objects conflict, the workflow designer has to specify a condition based on which the WFMS can determine if two steps conflict. In a enterprise where there are perhaps 100 workflow schemas, the number of workflow pairs that conflict is likely to be very small since they are designed for different business objectives (customer order processing, administrative workflows, etc.) and access different data objects. Even if it seems that a conflict specification has to be created for all steps of all pairs of workflows, in reality the "global matrix view" of conflict specification would be sparse. For example, steps of order processing workflows are not likely to conflict with those of administrative workflows. Even amongst order processing workflows conflicts are likely only between workflows related to similar products or products that use same/similar components.

In summary, we have a practicable approach for providing additional correctness properties at the workflow level. The conflict specification process is slightly involved but it is not something that is beyond the reach of a workflow designer (since the designer already has a good knowledge of the data and control flow mapping and the semantics associated with each steps' data input and output), especially since these additional specifications enhance functionality, performance and failure handling capabilities of the system. Techniques to automate the specification of conflict dependencies and coordinated execution requirements based on step level information have been suggested elsewhere [18]. Note that our strategies have been developed for production (or repetitive) workflows and may need enhancements to handle ad-hoc requirements/changes in workflows.

6 Compile-Time Actions

The compiler for LAWS works in two phases. During the first phase, it parses the schema and performs extensive syntax and semantic error checking. In the second phase, it generates step related information (node-name, program-name etc.) and composite rules from all the different specifications provided in the workflow schema. These rules are then used by the run-time environment for navigating through the workflow instances, handling failures and performing coordinated execution. In this section we will discuss the details of the two phases.

6.1 Syntax and Semantic Checking

The first phase of the compiler performs the following: (i) for each step specification check if all the required inputs (like nodename, program name, etc.) are specified, (ii) for data flow specification, check if the inputs sources to each step are valid (iii) for control flow specification, check if the condition and step(s) to which control flows are valid (iv) for failure handling specification check that the steps to be compensated do exist (v) for re-execution specification check if the condition is valid and (vi) for conflict and coordinated execution specification check if the steps referred to in the other workflows are valid. If any required information is not available, a default value/action is assigned if possible. For example, if a failure handling specification does not exist for a step, the default action of workflow abort is assigned to the step.

6.2 Translating Mandatory Specifications

The second phase of the compilation is more involved. The rules generated in this phase are the conventional Event-Condition-Action rules used in active database systems. All rules generated are stored in a $RuleSet$. During the rule generation phase, existing rules in the $RuleSet$ are modified or new rules are added to the $RuleSet$. The compiled output finally contains the step related information and the rules generated in the second phase of compilation (stored in the $RuleSet$).

The first set of rules are generated based on the data flow dependencies, where the completion of the individual steps that feed data to a step becomes an event of the rule associated with the step and the start of the step forms the action of that rule. An example rule is (S=Start, D=Done)

E: S1.D AND WF.S AND S2.D AND S9.D
A: S11.S

This rule states that S11 is to be started after the start of the workflow and the completion of steps S1, S2 and S9. This requirement could have arisen from the data flow specification (indicating that S11 receives data from the workflow input and steps S1, S2 and S9). Existing rules are modified based on the control flow dependencies. More events are added to the rules that start the steps and conditions specified in the control flow specification are also added to the rules. Due to the control flow from the if-then-else branch after step S10, the above rule for example could look like

E: S1.D AND WF.S AND S2.D AND S9.D AND S10.D
C: (S10.O1 $>$ 50)
A: S5.S

where S10.O1 represents the first output of step S10.

6.3 Translating Failure Handling Specifications

These correspond to the specifications that realize opportunistic compensation and re-execution. An example rule is (F=Fail, C=Compensate, A=Abort)

E: S4.F
A: S1.C WF.A

This rule indicates that on the failure of step S4, step S1 is to be compensated and the workflow aborted. Another example is

E: S2.F
A: S1.C S1.S

This rule indicates that on the failure of step S2, step S1 is to be compensated and the workflow is to be re-executed from step S1. Information available in the step re-execution specification is added to the condition part of the existing rules whose actions correspond to the start of the steps. The rule condition is formed such that the re-execution condition is evaluated only if the execution thread was created due to the re-start of the workflow rather than the original thread (normal execution). Whether a thread was created as part of normal execution or re-start can be determined from the workflow state information at run-time. If there are compensation dependent sets, the closure of compensation dependent sets is first determined since they can overlap. Then the action specified on the failure of a step corresponds to the members of this closure set.

6.4 Translating Coordinated Execution Specifications

The conflict specification and the coordinated execution specification are processed in an integrated manner. From these specification the required pre-conditions are extracted and added to the action part of the relevant rules in the $RuleSet$. Thus, when a workflow is instantiated, the run-time RuleSet (all rules related to the workflow schema at run-time) has only those rules related to the normal navigation and failure-handling of the workflow. Later depending on the state of the workflow data, coordinated execution rules are added to the run-time RuleSet *dynamically*.

In the following discussion, workflows whose events are to be synchronized are referred to as WF1 and WF2. $SynchEvent$ can be $IMMEDIATE$ or a normal event. IMMEDIATE implies that the rule is to be fired immediately after it is received in a workflow instance without waiting for any event(s) to happen. $Condition$ is a comparison involving the state of both the workflows. $SynchAction$ can be one of:

- $FinishOrBlock$ Step — complete the step if it has already started otherwise block the step
- $Block$ Step — block the step
- $Finish$ Step — complete the step

The lower level implementation primitives are as follows:

- *AddRule(WorkflowName.InstanceID, SynchEvent, Condition, SynchAction)*
  which indicates that a rule consisting of the tuple *(SynchEvent, Condition, SynchAction)* is to be delivered to an instance of WorkflowName with an identity of InstanceID. If the InstanceID part is skipped then the rule is delivered to all instances of WorkflowName
- *AddEvent(WorkflowName.InstanceID, Event)*
  which indicates that $Event$ is to be delivered to an instance of $WorkflowName$ with an identity $InstanceID$
- *AddPrecondition(WorkflowName.InstanceID, Event, Action)*
  which indicates that the *(Event, Action)* tuple is to be delivered to an instance of $WorkflowName$ with an identity $InstanceID$ and the $Event$ is to be added as a pre-condition to the rule that fires $Action$

We now show how the semantics of higher level building blocks are realized using lower level primitives. As an example, we discuss the translation of *relative ordering* into low level primitives. In the discussion below, WF1: $< line >$ indicates that WF1 enacts $< line >$. Also, since we are only considering one instance each of WF1 and WF2 in this discussion, the instanceID has not been specified in the parameters parts of the lower level primitives. In practice, when there are several instances, the instanceID has to be specified. Section 7 discusses this issue further. Enactment of the primitives alone is shown to complement the figure.

An example user specification for relative oredring is

```
IF S1.I1 > WF2.S2.I2 THEN S1 CONFLICTS_WITH WF2.S2 ;
IF S1 CONFLICTS_WITH WF2.S2 AND S1.START RELATIVE_ORDER WF2.S2.START
   AND S8 CONFLICTS_WITH WF2.S10
   THEN S8.START RELATIVE_ORDER WF2.S10.START ;
```

Three different cases are possible — (i) the conflict condition evaluates to FALSE or (ii) the conflict condition evalutes to

TRUE and the conflicting step has not been executed or (iii) the conflict condition evalutes to TRUE and the conflicting step has been executed. Due to space limitations we will just discuss the last case.

WF1: *AddRule(WF2, S1.DONE, S1.I1 > WF2.S2.I2, FinishOrBlock S2)*

WF2: *AddEvent (WF1, S2.ConflictDone)*

WF1: *AddPrecondition (Self, WF2.S10.Done, S8)*

WF1: *AddRule (WF2, IMMEDIATE, NULL, Notify S10.Done)*

WF2: *AddEvent (WF1, S10.Done)*

Thus WF1 executes step S8 Only after the event *S10.Done* is received from WF2.

6.5 Discussion

In the above approach for coordinated execution we discussed message exchange between workflow instances. Strictly speaking, an instance is not an active entity but basically contains state information. However, we use the notion of message passing between instances for ease of explanation. Note that these are *logical* messages and not physical messages. Logical messages do not have the overheads of physical messages and are comparable to that of a call to the engine (like a lock manager call in databases). Physical messages are exchanged when the instances under consideration are under the control of different engines or agents. A discussion on this can be found in [18].

An alternative way to manage rollback dependency is using the *spheres of control* approach where the database system tracks the dependency between the transactions and determines which other transactions have to be rolledback if a certain transaction rolls back. But building a system for maintaining spheres of control is not an easy task [15]. In contrast we have defined an approach where rollback dependencies are tracked *selectively* and the action to be taken is specified in the workflow schema and hence it can be automatically enacted should a rollback occur.

7 Run-Time Actions

In this section we focus on the run-time workflow control mechanisms to satisfy the specified requirement. When a workflow is instantiated, the RuleSet and the step information corresponding to that workflow schema are loaded (if it does not exist already). Then a *workflow.start* event is generated which triggers several rules that have *workflow.start* as one of their events. Typically for the *start* steps of the workflow, this is the only event needed to start the step. Then the step is executed by sending the required information (apart from the rules, step level information and data mapping information is available as part of the compiled schema) to the appropriate agent and getting back the results. If the step succeeds a *step.done* event is generated else a *step.fail* event is generated. This in turn triggers the appropriate rules and the execution continues till a *workflow.complete* or *workflow.abort* event is generated. When a rule is triggered, only after all the required events have been generated is its action fired.

All the rule and step information is grouped and maintained on a per instance basis and if no coordinated execution is required there is no need for message exchange between workflow instances. However, when coordinated execution is required between workflows, messages are to be exchanged between the workflow instances to handle our primitives like *AddRule()* and *AddEvent()*. The workflow engine has complete knowledge of all the workflow schemas and the number of instances of each currently in progress along with their respective instance IDs. Indexes are maintained so that state information of any specific instance of a workflow can be easily accessed or updated.

Efficiency is as important concern in rule-based systems. Performance can degrade in systems where composite events have to detected based on the occurrence of primitive events. Although the events in our rules are not primitive, they are not very complex to cause inefficient execution. In addition we also maintain indexes on the primitive events listed in the event part of the rules. On the occurrence of a primitive event like a step completion event, the indexes help us quickly determine which new rules and pending rules are to be considered for firing. Thus, the core infrastructure we have developed for scheduling steps is efficient. Rule-based approaches to handle workflow executions have also been used by others [5].

The CREW environment contains nodes of heterogeneous architectures. To achieve total portability the LAWS compiler and a subset of CREW run-time environment have been implemented using the *Java* programming language. Additional details of the implementation can be found in [18].

8 Related Work

Sagas [12] was one of the first proposals to integrate several transactional activities into a large application. Later several extended transaction models were proposed [11] and formalisms/languages/environments developed to realized extended transaction models. Notably, work on long running activities [9], ConTracts [24], inter-task (inter-step) dependencies [7,4], and customized transaction models (TSME) [14] have all focussed primarily on coordinating sub-transactions based on the significant events like *begin, commit, abort* of sub-transactions. They all assume access to the various transaction managers to get transaction related information. This is not the case in workflow environments which attempt to integrate independently developed applications. ConTracts deals with issues related to coordinated execution across long computations using the notion of *invariants*. However they do not address the type of coordinated execution requirements discussed in this paper. They provide a high-level discussion on how rollback dependency can be handled if spheres of control [8] are available in a system. Also, none of the above discuss the race conditions created due to failures along parallel threads of execution within a workflow. The need for mutual exclusion of steps from concurrent workflow processes to ensure correct interleavings has also been discussed in [1] where a history based protocol is proposed to dynamically impose access restrictions. In contrast, we have developed a uniform set of low level mechanisms to handle different type of coordinated execution requirements by dynamically modifying workflow RuleSets.

Handling of logical failures of steps in workflows has received more importance recently. Failure handling of steps is discussed in [21] using the notion of spheres of joint compensation. Although it provides a systematic way of partially rolling back a workflow using cascaded rollback of spheres, it does not deal with the interactions between rollback and forward execution of other steps in a workflow. In contrast, our opportunistic compensation and re-execution technique takes an integrated look at workflow rollback and re-execution both from correctness and performance perspective. There has also been some recent work on failure handling in transaction hierarchies [6]. While our work focuses on modeling and system support for custom handling of workflow failures, other general strategies particularly those discussed in advanced transaction models can be used to handle workflow step failures. A detailed discussion on how concepts from advanced transaction models such as sagas and flex transactions can be used to handle workflow step failures is provided in [2]. Another transaction oriented approach for handling workflow exceptions is presented in [10]. A three layer error model for exception handling in the METEOR system is presented in [25] for handling exceptions where different corrective action is taken depending on the specific reason of failure. In contrast, our focus has been on generic problems that can occur at the workflow (application) level, focusing on the effect of a step failure on other concurrent steps within the same workflow and across workflows. Hence, the solutions we have developed and in [25] complement each other.

9 Conclusions

Commercial/prototype WFMSs and current research literature do not adequately address problems arising from interaction between steps executed concurrently both within and across workflows and the presence of failures. In this paper we described the concepts and infrastructure that we have developed in the CREW project at UMass to handle these problems. To support coordinated execution we identified a set of high level building blocks which a workflow designer could use to specify coordinated execution requirements. Also, to handle failures we proposed a novel and pragmatic approach called *opportunistic compensation and re-execution* that allows a workflow designer to customize workflow recovery both from correctness and performance perspective. The infrastructure we have developed consists of two main components — a language to specify the requirements and a run-time environment that satisfies these requirements while executing the workflows.

An important issue related to our opportunistic compensation and re-execution strategy is the overheads involved in implementing the strategy and the related performance benefits. To implement this strategy, the overheads include maintaining additional data that correspond to the previous execution of the steps and checking the appropriate conditions before the execution of a step. Since these overheads are usually small, it is not expensive to use this strategy. The benefits from the strategy vary depending on the specific nature of the step. If a step involves executing simple programs that do not access many resources then the savings are not likely to be substantial. However, in most real-world workflows a step involves either accessing a number of records from a database or human oriented tasks like moving inventory where the savings are considerable. In general we believe that the benefits from the opportunistic compensation and re-execution scheme will be considerable while incurring a small overhead.

While WFMSs initially offered only minimal modeling features like data and control flow, additional features are being added or developed to improve the modeling and correctness properties offered by WFMSs and to make them more robust and reliable. Our work is an important step in this direction. Since centralized workflow control can be a performance bottleneck, we have extended our work to support parallel and distributed workflow control architectures [18]. Also, the CREW run-time environment has been extended to implement WorldFlow, a WFMS that can be used to develop Web based electronic commerce workflows. In these workflows, one or more of the steps read/update data directly communicating with Web servers. Since the Web is not reliable (*i.e.*, the network/servers can fail), some of the failure handling infrastructure discussed in this paper will be very useful.

References

[1] G. Alonso, D. Agrawal, and A. El Abbadi. Process synchronization in workflow management systems. In *8th IEEE Symposium on Parallel and Distributed Processing (SPDS'97)*, New Orleans, Louisiana, October 1996.

[2] 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.

[3] 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.

[4] 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.

[5] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In *Proceedings of OO-ER Conference*, Cold Coast, Australia, 1995.

[6] Q. Chen and U. Dayal. Failure Handling for Transaction Hierarchies. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1997.

[7] Panos Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Trans. on Database Sys.*, 19(3):450–491, 1994.

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

[9] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-running Activities with Triggers and Transactions. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, pages 204–214, June 1990.

[10] J. Eder and W. Liebhart. The workflow activity model wamo. In *Proceedings of 3rd Intl Conference on Cooperative Information Systems*, Vienna, Austria, September 1995.

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

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

[13] 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.

[14] 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.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

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

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

[18] M. Kamath. *Improving Correctness And Failure Handling In Workflow Management Systems*. PhD thesis, Computer Science Department, University of Massachusetts, February 1998.

[19] 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.

[20] M. Kamath and K. Ramamritham. Correctness Issues in Workflow Management. *Distributed Systems Engineering Journal*, 3(4):213–221, December 1996.

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

[22] 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.

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

[24] 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.

[25] D. Worah. Error Handling and Recovery For The ORBWork Workflow Enactment Service in METEOR. Master's project, University of Georgia, Computer Science Department, 1997.