IMPROVING CORRECTNESS AND FAILURE HANDLING IN WORKFLOW MANAGEMENT SYSTEMS

A Dissertation Presented

by

MOHAN UMESH KAMATH

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 1998

Department of Computer Science

IMPROVING CORRECTNESS AND FAILURE HANDLING IN WORKFLOW MANAGEMENT SYSTEMS

A Dissertation Presented

by

MOHAN UMESH KAMATH

Approved as to style and content by:

Krithivasan Ramamritham, Chair

Lori Clarke, Member

Don Towsley, Member

C. M. Krishna, Member

C. Mohan, Member

David W. Stemple, Department Chair
Computer Science

To My Dear Wife and Loving Parents

ACKNOWLEDGMENTS

I am very grateful to Prof. Krithi Ramamritham for being my advisor and mentor. His inspiring thoughts and general support has assisted me in my personal and professional growth. The technical guidance and constructive criticisms he provided has helped me greatly in conducting my research and developing this thesis.

I wish to thank Dr. C. Mohan at the IBM Almaden Research Center for giving me an opportunity to be a part of the Exotica Project on Workflow Management. My discussions with him and other colleagues of the project have provided me a lot of insights into the area of workflow management. I also wish to thank Dr. Narain Gehani and Dr. Dan Lieuwen for helping me explore Web related issues in the context of workflow management and electronic commerce during my visit to Lucent Technologies' Bell Labs.

My thanks are due to Prof. Don Towsley for providing me with valuable suggestions and guidance in this and other areas of research I have worked with him. Comments provided by Profs. Lori Clarke and Mani Krishna have been very helpful in preparing this thesis.

My thanks also go to all my officemates, and several other friends both from within and outside the department for their help and the interesting times we have shared during the course of my stay at UMass. My special thanks to Betty Hardy, Sharon Mallory and other members of the department front office for promptly and immensely helping me in several administrative activities. Last, but not the least, I am indebted to my wife Rekha for offering me tremendous support and help through my graduate studies.

ABSTRACT

IMPROVING CORRECTNESS AND FAILURE HANDLING IN WORKFLOW MANAGEMENT SYSTEMS

MAY 1998

MOHAN UMESH KAMATH

B.S., BHARATHIYAR UNIVERSITY, INDIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

PH.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Krithivasan Ramamritham

A workflow management system (WFMS) facilitates the coordinated execution of applications (steps) that comprise a business process (workflow) across distributed nodes. State-of-the-art WFMSs do not have adequate support for handling various correctness and failure handling requirements of workflows. Different correctness requirements arise due to the dependencies between steps that access shared resources. Failure of steps in a workflow and system failures can cause data inconsistencies if handled improperly. Scalability is also a problem in current WFMSs since they use a centralized workflow control architecture that can easily become a performance bottleneck.

In this thesis, we have developed the concepts and infrastructure necessary to address the above issues. To handle dependencies across workflows we have developed techniques for expressing and handling *mutual-exclusion*, *relative ordering* and *rollback dependency* requirements across workflow instances. To handle failure of steps within a workflow, we have proposed a new *opportunistic* scheme that avoids unnecessary compensations and re-executions when workflows are rolledback partially and re-executed. To handle system failures we have designed suitable logging schemes and protocols. To achieve scalability while satisfying the different

correctness and failure handling requirements, we have enhanced our techniques to work on parallel and distributed workflow control architectures.

To realize the above concepts, we have designed a workflow specification language, a two stage compiler and a rule-based run-time system. A workflow designer specifies the workflow schema and the resources accessed by the steps from a global database of resources. The two stage workflow compiler determines data dependencies and translates the high level schema into a uniform set of rules. The run-time system interprets these rules and executes the workflows in accordance with their requirements under central, parallel and distributed workflow control. To demonstrate the usefulness and practicality of our approach, we have implemented a prototype system that can offer better correctness, performance and functionality than state-of-the-art WFMSs.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

To efficiently utilize the resources available and to gain an edge over their competitors, organizations are increasingly relying on enterprise-wide integration of information using advanced technologies like database systems, client-server computing and the World Wide Web. Amidst this trend, *workflow management* has emerged as a practical technique to automate the execution of *business processes* of organizations. It has attracted attention from both industry and research since banks, insurance companies, manufacturing companies, healthcare organizations and telecommunication companies have shown interest in adopting the technology for conducting their daily business. Workflow management is also considered to be an essential technology for enabling electronic commerce.

Workflow management facilitates the coordinated execution of applications that comprise a business process across distributed nodes. A business process or a *workflow* is defined by the Workflow Management Coalition [64] as "a procedure where documents, information or tasks are passed between participants according to defined set of rules to achieve, or contribute to, an overall business goal". Alternatively, a workflow can be viewed as a "collections of steps" that have to be performed in a certain partial order. Usually an application has to be executed to perform a step. The application may access data from one or more resource managers (*e.g.*, databases, file systems) and make its changes persistent before it completes.

As shown in figure 1.1, a workflow can be viewed as a "collections of steps" that have to be performed in a certain partial order. A *workflow schema* specifies the

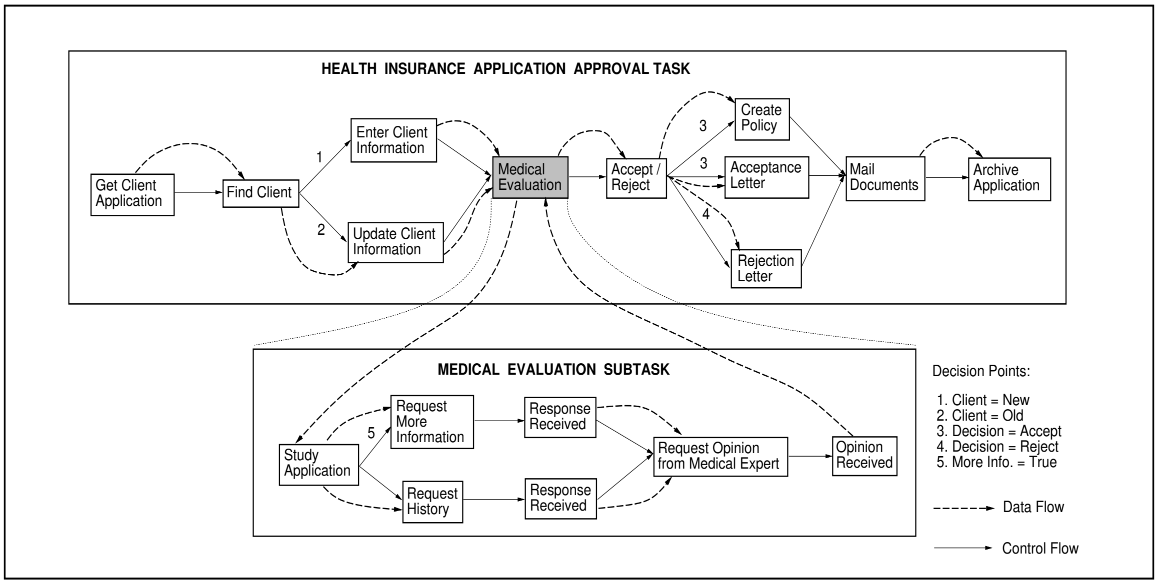

Figure 1.1 Example of a Workflow Application

set of steps that comprise a workflow and the data and control flow between the steps. The data flow specification provides the mapping of data between steps and control flow specifies the execution order of the steps. Several types of branching can be specified in the control flow requirements — sequential, conditional branching, parallel branching and loops. Thus a workflow schema provides the definition of a business process. To perform a business process, a *workflow instance* is initiated. In an object-oriented sense, a workflow schema is a template or a class definition and a workflow instance is an instance of the class definition, *i.e.*, a workflow instance represents a copy of the schema template filled with specific values. Every workflow instance is associated with a *state* which reflects the values of the various data items associated with the workflow and the state of the steps of the workflow, *i.e.*, which of the steps have been performed and which have not been. In the rest of the thesis the term 'workflow schema' is used to explicitly refer to the template or the workflow definition and 'workflow' to represent an instance.

Figure 1.2 presents an architecture of a WFMS as suggested by the reference model [64] of the Workflow Management Coalition. A *Workflow Management*

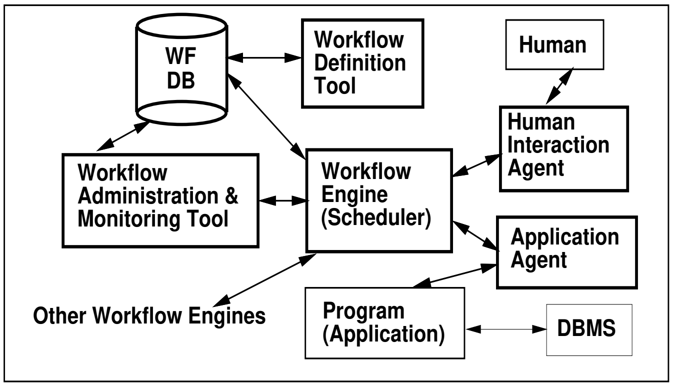

Figure 1.2 WFMS Architecture

System, WFMS, primarily consists of a modeling tool to define workflow schemas, a workflow engine that provides navigational support to control the execution of the workflow instances in accordance with the workflow schemas and application agents which supervise the execution of the application programs that are performed on behalf of the steps. The workflow engine manages the state information of all the workflow instances in progress and communicates with the application agents to perform the steps and obtain the results of step execution. The modeling tool and the workflow engine also communicate with a workflow database. The workflow database stores the workflow schema information and state information of the workflow instances. The workflow database is not accessible external to the WFMS and the WFMS has no knowledge of the databases and other *external* resources (resources outside the WFMS) accessed by the applications performed on behalf of the steps. A typical WFMS may store several workflow schemas. Multiple instances of each of the workflow schemas can be in execution at any given time.

1.1 Motivation

Although WFMSs have been maturing, there are a number of correctness and failure related issues which state-of-the-art WFMSs do not adequately address. This is a major drawback, especially since WFMSs are to be used in banks, insurance companies and healthcare organizations where data inconsistencies can lead to undesirable consequences.

Current prototype and commercial WFMSs have good support only for handling execution order (control flow) dependencies between steps of an individual workflow. They have limited capability for failure handling (both exception handling and recovery). Most of the WFMSs use a centralized workflow control architecture. Also, workflow agents in WFMSs are stateless. All of these deficiencies have an adverse impact on the correctness and performance of workflow execution. After analyzing different workflow execution scenarios several issues that need to be addressed have been identified. These issues will be motivated in the rest of the section.

Applications that execute to perform the steps of workflows can access shared resources like databases and files. When several workflows execute concurrently, conflicts and dependencies can arise between steps of different workflows, *i.e.*, events happening in one workflow can potentially have an effect on other concurrent workflows. For example, if a step from a workflow changes the value of a data item in a database and is subsequently compensated due to a failure in that workflow, steps from other workflows that accessed the data item after the initial update may observe the data in an inconsistent state. Such data dependencies have to be handled correctly. Consider another scenario where multiple *threads* (flow of control along a path of the workflow) of execution in a single workflow interact with each other. Data dependencies can arise due to the run-time interactions between threads. Thus, there is a *need to handle conflicts and dependencies within and across workflows.*

Due to the complex nature of business processes, failure of a step (unsuccessful execution) in a workflow can influence the execution of other steps in the same workflow. Consider a scenario where several threads are executing concurrently within a workflow. Failure of steps in one or more of the threads of a parallel workflow can cause race conditions since steps can be executed based on inconsistent data values received from different threads. Also, on the failure of a step, compensating (undoing the effects of the step) *all* of the completed steps of a workflow and aborting the workflow may be very conservative and sometimes impractical. Strategies such as spheres of joint compensation [86] are geared specifically towards workflow rollback but do not deal with the effect of the rollback on the re-execution of workflows. Hence, there is a *need for good failure handling techniques at the workflow level.*

Since workflow agents are stateless, failures at the workflow agent can cause idempotency problems. Consider a scenario where a workflow engine failure is followed by the failure of the agent. The agent may be supervising the execution of a program that performs a step. After the program terminates successfully, the agent may be unable to communicate the completion of the step and the program results to the workflow engine that has failed. Before the workflow engine restarts the agent could also fail. The workflow engine will try reconnecting to the agent. Unless proper care is taken, results of the execution of steps that completed between the two failures will be lost. This results in an inconsistency since the state of the step as recorded in the workflow database will indicate that the step is still 'running', even though the program has executed and the step has been actually completed. If the engine times out, it may start the step again although it has already been executed. This can result in duplicate executions which are to be avoided. The same phenomenon can be observed in Web based workflows where the unreliability of the Web can cause idempotency problems while handling steps that update Web information sources, specifically after a step is scheduled if a response is not received within the timeout period from a remote Web server, without additional mechanisms it is not possible to determine whether the request was processed or not. If the step is scheduled again, there is a possibility of duplicate execution. Hence, there is a *need for mechanisms to achieve idempotency in the event of agent failures.*

Centralized workflow systems have an advantage since tracking workflow progress is very simple. However they provide little scalability. Large corporations typically have a few hundred thousand instances of workflows executing concurrently. In such environments, a centralized architecture can become a performance bottleneck [97]. Hence, there is a *need for using non-centralized workflow control architectures to achieve scalability.*

These are the significant problems that were discovered while analyzing several workflow execution scenarios. Other problems such as failure of the workflow engine and the workflow database have already been addressed [14, 72]. To summarize, this thesis addresses the following key issues:

1. coordinated execution of concurrent workflows,

2. workflow-specific failure handling techniques,

3. achieving idempotency in the event of agent failures.

4. avoiding centralized control of workflows, and

Lack of support for ensuring data consistency within and across workflows is a key requirement to improve applicability of WFMSs. Hence, the issues addressed in this thesis revolve around the theme of improving data consistency while still maintaining a high level of performance. This will bring WFMSs a step closer towards transaction processing systems while still maintaining the flexibility offered by them.

1.2 Contributions of this Thesis

This thesis discusses the concepts and infrastructure that have been developed in the CREW (Correct and Reliable Execution of Workflows) Project at UMass to address all the issues highlighted in the previous section. The solutions presented offer better correctness, performance and functionality than state-of-the-art WFMSs.

1.2.1 Conceptual Contributions

We have developed the following concepts and techniques to address the various problems:

- **Techniques for coordinated execution of workflows:** We have identified high level building blocks that express *mutual-exclusion*, *relative ordering* and *rollback dependency* requirements across workflow instances. These building blocks have been developed to directly address the various problems encountered during concurrent execution of workflows. Since a WFMS has no knowledge of the resources accessed by the applications, resources used by a step are identified by the workflow designer. A global resource database (populated by the workflow designer) stores the resource identifiers of the resources accessed by the various steps. The resource database is a key component since the data dependency extraction techniques use it to determine various types of dependencies that can be expressed using the high level building blocks. The resource database is used only at compile time to extract potential conflicts. Irrespective of the workflow control architecture this resource database is needed only at nodes where workflow schemas are compiled. The dependencies themselves are handled by the run-time system.

- **Workflow-specific failure handling techniques:** We have proposed an integrated approach to handle partial rollback and subsequent re-execution of workflows in the event of a logical failure of a step. The integrated approach uses an *opportunistic compensation and re-execution strategy* that minimizes the amount of lost work and the amount of work that is redone. This strategy compensates and re-executes a step *if* and *when* it is needed. In addition, the use of *compensation dependent sets* is also proposed to customize failure handling requirements of a workflow. When a compensation dependent set is rolled back, the member steps of that set are compensated in the reverse execution order. These techniques can be used to get better correctness and performance at the workflow level. They have also been enhanced to handle race conditions in workflows.

- **Techniques for achieving idempotency in the event of an agent failure:** This problem can be ameliorated by having logging facilities at the application agents. Every application agent logs significant events, *e.g*, when a program completes, the return status code and data produced by the program. Using such a mechanism even if an agent fails, after it has recovered it can communicate the results to the workflow engine without causing inconsistencies in the state information about the step. To handle idempotency of steps in Web workflows, a simple *exactly once semantics* protocol has been developed. In this protocol, the WFMS generates a unique ID for every remote step to be performed over the Web and stores this ID persistently. The WFMS uses this unique ID and a set of standard interfaces at the remote servers to achieve idempotency even in the presence of failures.

- **Architectures for avoiding centralized control of workflows:** To address the scalability issue *two alternative workflow control architectures have been considered — parallel and distributed.* In a parallel architecture several workflow engines perform the navigation of workflow instances in parallel (still communicating with the same set of agents as in a centralized architecture). In a distributed architecture the navigation is performed by the agents themselves without the use of a workflow engine. In both architectures, steps from concurrent workflows each of which is managed by a different workflow engine/agent can conflict. Hence, in comparison with a centralized workflow control, additional exchange of messages is required to synchronize/notify events across workflows and thus coordinate the execution of workflows. The necessary protocols for parallel and distributed workflow control have also been developed. These protocols use piggybacking and other techniques that exploit available information in order to *reduce the number of additional messages exchanged.*

1.2.2 Infrastructure Contributions

Based on these concepts and techniques, we have designed and implemented appropriate system support. The following are the salient aspects of the system support developed in this thesis:

- **Design of a workflow specification language and a compiler:** As part of the solution, a *workflow specification language* has been developed that has primitives for specifying traditional workflow requirements as well as enhanced requirements for failure handling and coordinated execution. This language is also extensible in the sense that more primitives can be added to the language to handle the requirement of new workflow execution scenarios. As part of the language infrastructure, a *two stage compiler* to process workflow schemas specified in the workflow language has also been developed. The first stage extracts data dependencies and creates an enhanced intermediate schema. Since a rule-based run-time system (explained below) is used to handle workflow execution, the second stage of the compiler processes the intermediate schema and generates appropriate rules.

- **Design of a rule-based run-time system for central, parallel and distributed workflow control architecture:** 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, a rule based approach is more suited to manage the execution of workflows at run-time. The rule-based run-time system interprets the rules at run-time and ensure that the workflows are executed in accordance with the specification. A small set of *implementation level primitives* have been identified that dynamically modify the rules and are sufficient to realize the high level coordinated execution. Mechanisms needed for correctly implementing the various failure handling requirements have also been developed. The run-time system support has been designed to satisfy specified correctness properties under centralized, parallel and distributed workflow control architectures. Using a simple formal model the *correctness* of the low level implementation mechanisms has also been proved.

- **Implementation of a prototype system to demonstrate practicality:** A prototype WFMS has been implemented to realize a subset of the techniques developed. The prototype has been built using the Java programming language for achieving portability across heterogeneous architectures. The prototype includes a workflow compiler and a rule-based run-time system for central, parallel and distributed workflow control architectures. Applications were simulated with different failure probabilities and traces obtained to verify that the executions are in accordance with the schema specifications. The prototype system has also been extended to implement WorldFlow, a system for reliably handling Web workflows.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows.

- *Problem Statement and Analysis:* Chapter 2 first provides an overview of workflow management. It describes the essential modeling features and system support provided by current WFMSs. The rest of the chapter focuses on correctness and performance issues in workflow execution. Different workflow execution scenarios are analyzed to motivate the various problems addressed in this thesis.

- *Approach to solve the problems:* Chapter 3 starts with a broad picture of the approach to solve the various problems. It first presents the various concepts and algorithms developed for handling failures including the opportunistic compensation and re-execution scheme that reduces the amount of work lost due to unnecessary step compensations. Then it presents the techniques for handling coordinated execution of concurrent workflows and the interactions between them.

- *Language and Compiler Support:* Using the concepts embedded in the approach we first describe the modeling support developed for workflow management. Chapter 4 presents details of the workflow modeling language LAWS (LAnguage for Workflow Specification) and the modeling primitives it offers. The semantics of the LAWS primitives have been captured using a simple language called LITES (Language for Implementation TEchniques Specification). Chapter 5 first discusses the low level implementation primitives including those used in rule-based system. The chapter then describes the two stage CREW compiler that processes the workflow schema and generate appropriate rules. Then the chapter presents proofs of correctness of low level mechanisms for realizing the semantics of the LAWS primitives using LITES. The grammar for LAWS is provided in Appendix A and a sample specification using LAWS is provided in Appendix B

- *System Support:* The final discussion is on system support for workflow management based on the concepts embedded in the approach. Chapter 6 describes the CREW run-time system. It presents techniques for managing workflow execution using rules in the presence of concurrency and failures. The chapter discusses schemes for managing workflow execution under different workflow control architectures and the implementation of the prototype system. Protocols that reduce the number of messages exchanged in parallel and distributed workflow control architectures are presented along with the details of the various data structures used and other systems related issues. This is followed by an explanation of the prototype system that has been implemented using the Java programming language. Finally, we discuss and compare other approaches (whenever applicable) that could be used to handle the various problems addressed in this thesis. In chapter 7, enhancements to the prototype system are discussed for the implementation of WorldFlow, a system for handling Web workflows. It also describes how WorldFlow can be used to build new applications.

- *Related Work and Conclusions:* Chapter 8 discussed related work. Chapter 9 revisits the contributions of this thesis and summarizes the directions for future research.

CHAPTER 2

Workflow Execution Correctness and Performance Issues

To build the context for our discussion, a brief overview of workflow management is provided as it is accomplished today. Based on this overview, the inadequacies in the context of workflow execution correctness and performance are then analyzed.

2.1 State-of-The-Art Support for Workflow Management

Workflow can be categorized into three types as done in [92] — *ad hoc, administrative* and *production.* In ad hoc workflows, the workflow structures are not prespecified and executions are not automated. Communication typically takes place through electronic mail or document exchange. Contingencies are handled manually. Administrative workflows are better structured than adhoc workflows and, hence, their execution can be automated. They typically handle simple tasks (*e.g.* decision approval) and failure of workflows are tolerated since they do not hamper the regular functioning of the organization. The third and the most interesting category of workflows are production workflows. Workflow schemas are usually well formed and pre-specified. These workflows handle the regular business of an organization and are usually executed repetitively. Maintaining accuracy of information is critical for production workflows and hence this thesis will be focusing only on production WFMSs.

Organizations capture the details of their business processes and map them to workflow schemas (templates) based on the modeling capabilities provided by a WFMS. The schemas are then instantiated as needed to run the actual business processes. Several instances of a schema may be executing at any given time. The rest of the overview will focus mainly on the modeling and system support in state-of-the-art WFMSs. This will also describe the semantics of a workflow. Additional details on the WFMSs can be found in [65, 56, 66].

2.1.1 Modeling Support

Modeling support determines the flexibility and precision available for designing workflows according to the business process requirements and the facilities available for specifying how deviations from these requirements are to be handled. Based on the WfMC's *reference model* and features provided by other state-of-the-art WFMSs, the rest of the section discusses the prominent modeling features.

A step definition specifies how the step is to be executed (*i.e.*, automatically or by a human role) and the name of the program to be used for executing the step. Each step has a set of *input* and *output* parameters. Input and output parameters can be integers, floats, strings, array and even complicated structures. *Data flow* in a workflow is defined by mapping the output parameters of one or more steps to the input parameter of a step. This mapping can range from simple integer values to file names and other complicated objects. *Control flow* determines the path taken during the execution of the workflow. This is specified by defining a directed edge between the steps. Several types of branching can be specified in the control flow requirements — sequential, conditional branching, parallel branching and loops. For conditional branching, a predicate is associated with a control flow edge. This predicate is evaluated at run-time to determine which branch is to be taken. In the case of parallel branching control flows through all the branches succeeding a step. Loops are used for repeated execution of a set of steps. After the set of steps have been executed for the desired number of times (as determined by the loop condition), control flow exits the loop and proceeds with the rest of the workflow. Data and control flow specifies data flow and execution dependencies between the steps. Note that notion of *data dependency* is different from the notion of *data flow dependency*. Data dependencies between steps from same or different workflows are caused due to access patterns of steps on external resources like the application database. Alternately, a pair of steps have a data flow dependency if one step requires data produced by the other. Steps from different workflows that access the same external resource in conflicting modes have a data dependency between them. A pair of steps have a control flow dependency if one step requires the completion of the other. Workflows can also be nested by having a different workflow perform the work of a step. In nested workflows, instead of specifying the program to execute a step, a workflow is specified. This is shown by the "medical evaluation" step in Figure 1.1. A workflow schema can also contain loops. Loops allow a set of steps to be repeatedly executed. Flow of control into a loop is not permitted. Thus, loops cannot overlap although they can be contained (or nested) inside one another. The least numbered step (closest from the start step) is considered the start step for a loop.

In addition to the ability to specify workflow schemas, there is modeling support for defining the organizational hierarchy and staff names with their designation. If a step is to be performed under human supervision, the step definition also contains a designation of the staff member responsible for executing that step. This provides flexibility since any person with that designation can execute the step rather than someone specific. In the rest of this thesis staff modeling issues are ignored as it is tangential to the main focus. All the modeling activities are performed via a workflow definition tool which is often GUI based.

2.1.2 System Support

2.1.2.1 Normal Execution

Referring to figure 1.2, the definitions of workflows, steps and human roles are all stored persistently in an underlying database commonly referred to as the *workflow database*. This database also stores the state information of the workflow instances that are in progress. Scheduling (navigation through the workflow instances) is performed by a *workflow engine*. The workflow engine communicates with the workflow database to query/update the state of workflow instances.

As shown in figure 1.2 workflow agents are of two types — *human interaction agents* that interact with humans and *application agents* that handle the automatic execution of the applications (program invoked to accomplish the goals of the step). The application agents interact with the workflow engine to fetch the workflow data required by the step and to communicate back the outcome of the step. The applications themselves access different resource managers some of which may be transactional, like DBMSs, and others which are non-transactional, like file systems. An abstract view of data exchange between the various components is shown in figure 1.2. Note that the applications have no knowledge about the workflow database and the workflow engine has no knowledge of the application databases and files. This is a crucial issue in WFMSs since the workflow engine just acts as a scheduler of steps from various workflow instances and *not* as a scheduler of resources accessed by the steps.

A significant issue in execution support is how the system coordinates (schedules) the steps at run-time based on their definitions. Most WFMS use workflow state based schedulers. The workflow database maintains the state of steps and workflows in progress. A step can be in one of the following states — *inactive, waiting, ready, executing* or *completed.* When a workflow is instantiated all the steps are in the inactive state. Once events that are responsible for executing a step start occurring, the step is in the waiting state when it waits for the other remaining events to occur. A step leaves the waiting state when it is determined based on the workflow definition and workflow state that the step is eligible for execution. A step becomes eligible when all the data items are available and control flow passes through that step. From the waiting state the step may enter an optional ready state (if the step is to be performed by human assistance) after it is added to the worklist (a list containing the steps that are to be performed by an individual) of the role responsible for that step. Once the role selects the step for execution then the WFMS schedules the step. At this point the step enters the executing state. If the step is to be executed automatically, it enters the executing state directly from the waiting state. Once the execution of the program (or task) representing a step completes the step enters the completed state.

After the completion of a step, the engine updates the workflow database with the information generated by the step. This action is also referred to as a *step commit*. If there are more steps to be executed in the workflow, the engine determines the steps that are eligible for execution and their inputs by querying the workflow database. After the execution of all the required steps of a workflow, the *workflow commits* and the instance is marked as committed in the workflow database. A workflow commit checks if all the required steps have been executed and unwanted but previously executed steps are compensated. WFMSs in general do not coordinate accesses to individual resources accessed by the steps. Although some document based WFMSs provide rudimentary concurrency control in the form of check-in and check-out, production WFMSs do not offer any kind of support for coordinating steps based on resources accessed by them.

In this thesis, the term 'thread' does not correspond to the notion of a thread as defined in the operating system sense. Note that the workflow engine itself may be multi-threaded indicating that the engine can handle multiple control flow along the same or different workflows.

2.1.2.2 Occurrence of Failures

Failures interrupt the normal execution of workflows and corrective action is necessary to execute the workflow from the point of failure.

Failures are of two types — *system* and *logical*. System failures occur in a WFMS when one of the three main components, *i.e.*, workflow engine, workflow database or the application agent fail. Logical failures (also termed as semantic failures in [11]) occur when a step cannot be executed successfully. Logical failures in turn can occur in different ways — (i) occurrence of error within the program or lack of resources needed by the program (*application failure*) or (ii) successful execution of the application which does not produce expected results (*exceptions*). Application failures are difficult to handle and there is ongoing research in this area [89]. Hence, application failures are not discussed further in this thesis. We however assume that on the failure of an application, the application returns the proper return code based on which corrective action can be taken in a workflow. This will allows application failures and application exceptions to be treated in the same manner. A step failure corresponds to on of the two cases.

The rest of this section discusses existing solutions for system and logical failures.

System Failures

Since workflows typically contain a large number of steps, unlike transaction processing systems in the event of an engine failure it is unacceptable to "undo" or compensate [82] the changes made by the workflow. Hence most WFMSs provide *forward recovery*, which requires that every workflow be continued from its state of execution prior to the failure instead of rolling back the entire workflow. Thus any useful work that has been done will not be lost due to unnecessary compensations.

Often to recover from the failure of a step it may be necessary to perform a partial recovery by compensating/undoing the effects of some of the steps of a workflow. An explicit program is required to compensate or undo a step. The compensation is actually performed using the state information stored in the workflow database. The WFMS has no knowledge of the working or semantics of the compensation programs which are provided by the workflow designers. The compensation program may do the exact inverse or a logical inverse. In this thesis, it is assumed that the compensation programs always succeeds. Finer mechanisms can be designed to compensate a step by allowing multiple retries until the compensation is successful.

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 stores the information in the workflow database. This amounts to taking an step-consistent persistent savepoint for each individual workflow. If a workflow engine fails, when it restarts after a failure it obtains the state of the different workflows in progress (by referring to the workflow database) and continues their execution from the respective most recently completed step before the failure thus achieving forward recovery. This will prevent the unnecessary re-execution of previously completed steps and thus, minimize lost work. This is in contrast to transaction systems where transactions that were executing at the time of failure are rolled back. 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 [14] where several workflow engines share a workflow database. In this scheme, if a workflow engine fails, the workflow instances controlled by the failed engine are handled by alternative workflow engines in the same cluster.

It can be observed that the workflow database is also a crucial component in achieving forward recovery. It is also susceptible to failures, thus making it difficult to achieve forward recovery at times. Hence, it is necessary to use fault tolerant techniques to replicate the state of the workflow database so that the workflows are continued from their respective states even if there is a failure. Techniques and algorithms to achieve this efficiently are described in [72]. These issues are not discussed further in this thesis since they are orthogonal to the main focus.

Logical Failures

Most failures may not require the rollback of the entire workflow and a partial rollback may be sufficient. A partial rollback involves rolling back the workflow to some previously executed step which we refer to as a *rollback step*. Then the workflow is re-executed from this rollback step. To provide 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 [86] 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. Note that to compensate or perform an "undo" of the step, WFMSs use compensating programs provided by the workflow designer. 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).

Inputs to a workflow that has been already instantiated can be changed dynamically. This would be done for example in an order processing workflow where the order quantity or type is changed. Although this scenario does not correspond to a failure, it is to be handled in a similar manner, *i.e.*, the workflow is to be partially rolledback and then re-executed.

2.1.2.3 Summary

To summarize, this thesis considers the following modeling and system support features:

- Step Level Specification — Program (application) for (i) executing a step, and (ii) compensating a step
- Control flow — (i) sequential, (ii) conditional branching, (iii) parallel, (iv) loops, and (v) nesting
- Data flow between steps
- Failure Handling — step failure and workflow cancelation (abort)
- Dynamic workflow input changes (when workflow instances are in progress)

The problems addressed and the solutions developed in this thesis considers the impact of the all the above mentioned features.

2.2 Correctness and Performance Issues

Based on the above discussion we now introduce some of the failure handling and coordinated execution problems in WFMSs that require special attention.

2.2.1 Need for coordinated execution of steps across workflows

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

One of the requirements of coordinated execution is the *mutually exclusive* execution of steps from concurrent workflows. Consider steps from two different workflows that access the same resource in *conflicting* modes. The resource manager serving the resource may not have support for concurrency control (e.g., non-transactional resource managers) causing data inconsistency. This can happen for example if applications are reading/updating file(s) served by a NFS file system which does not guarantee consistency of concurrent accesses to a file. While one option to solve this problem is to enhance the lower layers of the resource manager to add a layer that manages concurrent accesses, another is to specify a mutual exclusion requirement at the workflow level and let the workflow system handle it. Although the tradeoff between the two alternatives may depend on the specific scenario where the WFMS is used it is definitely desirable to have a general purpose facility within the WFMS for supporting mutual exclusion of steps. This view is supported with further examples. One of the main reasons for implementing WFMSs in organizations is to increase efficiency. Hence, to reduce queuing at resources that are shared by steps from several different workflows it would be beneficial to schedule the steps that share the resources in a mutually exclusive manner. Here mutual exclusion is needed due to application requirements. Mutual exclusion across workflows may be needed not just for a *single* step but a *group* of steps. Consider for example a workflow containing two steps that access a specific spreadsheet where the second last step verifies changes made during the first step (performed by a different staff member). Suppose a requirement is imposed where unverified changes made to the spreadsheet (between the two steps) should not be revealed to steps of other workflows. Then the two steps are to be executed as a mutually exclusive unit. Thus, steps from other workflows have to be executed either before the changes are made or after the changes are verified — conflicting steps from other workflows cannot freely

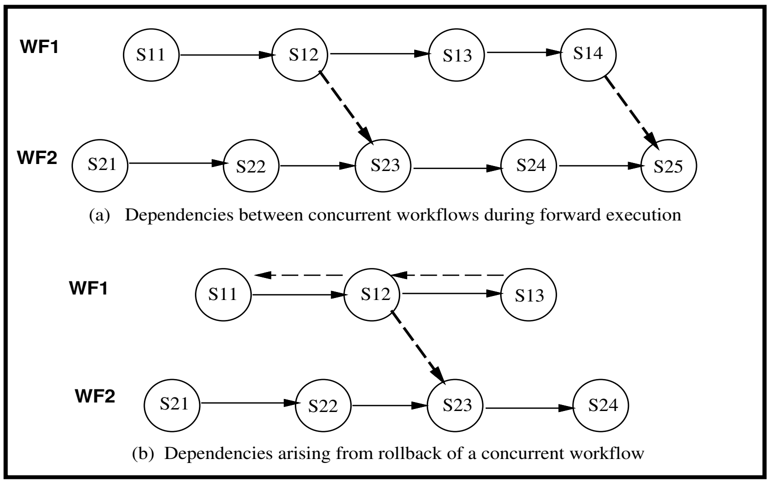

Figure 2.1 Dependencies Across Workflows

interleave between the two steps. Hence, from the perspective of mutual exclusion the two steps have to be treated as a group.

A second requirement of coordinated execution is the *relative ordering* of steps from concurrent workflows. Relative ordering requires that, if a pair of steps from concurrent workflows conflict, then the steps from all other conflicting pairs (between the respective workflows) have to be executed in the same order as that of the first conflicting pair. Hence, relative ordering is analogous to serialization of conflicting steps from a pair of workflows. An example is shown in *Figure 2.1(a)* where steps (S12/S23 and S14/S25 respectively) from concurrent workflows WF1 and WF2 access common resources and hence are to be executed in the same relative order. Such situations 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.*, steps from different workflows 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.

A third requirement of coordinated execution is *rollback dependency* between steps of concurrent workflows. *Figure 2.1(b)* shows an example where two steps S12 and S23 access common (conflicting) resources. Consider the rollback of workflow WF1 to handle the failure of step S13. This 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. Here step S23 is rollback dependent on step S12. The same figure can be used to illustrate another scenario. Let WF1 and WF2 be two order processing workflows and let steps S12 and S23 order the same components. Since step S12 executed first only it may be successful in fulfilling its order. Now, if S23 if not able to fulfill its order, its execution can be delayed until the fate of step S12 is known. If S12 is rolled back S23 may be able to complete. Otherwise, if WF1 completes and S12 is not rolled back, WF2 may have to be cancelled. Thus, steps S12 and S23 are rollback dependent. Note that there need not be rollback dependency between steps of concurrent workflows everytime steps from concurrent workflows conflict and one of them is rolled back. For example, in the last example if step S23 is successful, it need not be rolled back even if step S12 is rolled back in WF1.

These coordinated execution problems were deduced after (i) studying different workflow execution scenarios and (ii) comparing them with the concurrency control problems that are solved by database transactions. The important concurrency control problems that database transactions address include lost update, incorrect summary and dirty read. The coordinated execution requirements address these problems in the workflow context. For example, mutual exclusion (both single step and group of steps) addresses lost updates and the incorrect summary problems. Also, rollback dependency addresses the dirty read problem. However, as WFMSs are used in new application domains, new approaches may be needed to meet their requirements.

To summarize, three different types of coordinated execution requirements between steps of concurrent workflows have been identified. It should be possible to achieve these requirements using a WFMS. To the best of our knowledge, state-of-the-art commercial and prototype WFMSs neither have the provision to specify nor to enforce coordinated execution requirements across multiple workflow steps. Hence, it is necessary to develop a general purpose infrastructure for flexibly supporting coordinated execution of workflows.

2.2.2 Need for workflow-specific failure handling techniques

In the discussion of coordinated execution, the effect of failures in a workflow on other concurrent workflows was illustrated. Now the focus is on the effect of step failures on the execution of other steps within the *same* workflow.

Consider the example shown in *Figure 2.2* where a step fails in a workflow with if-then-else branching. Suppose the top branch is taken after the completion of

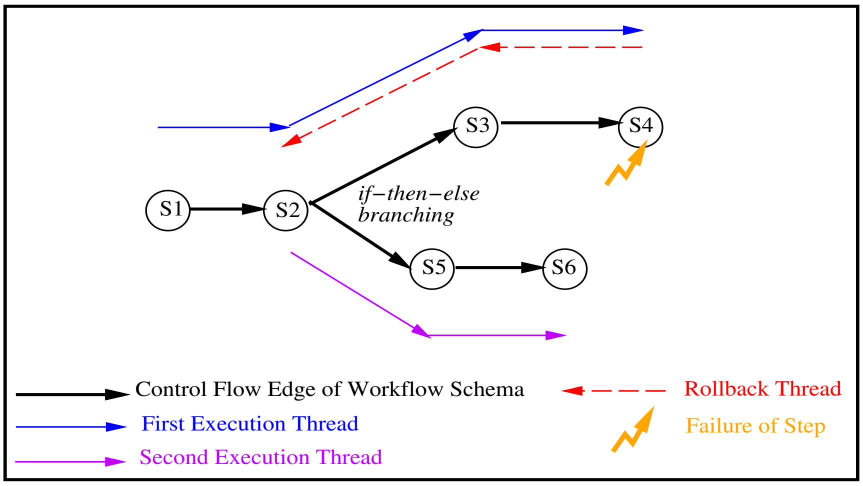

Figure 2.2 Rollback in a Workflow with if-then-else branching

step S2 and step S4 fails. The failure handling specification for S4 may require a partial rollback of the workflow to step S2 and re-execution from step S2. During the subsequent execution, because of changes in the data produced by step S2 (or from the feedback provided by the failure of 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 if it has made persistent changes. *When* S3 will be compensated depends on S3's data dependency between S3 and other steps in the workflow. If there is no data dependency with any of the steps then S3 can be compensated anytime before the workflow completes, including compensating it in parallel with the execution of other steps. 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 re-execution requirements.

Several extended transaction models have been proposed for handling long duration applications such as workflows. The Sagas model [55] in particular has been developed for workflow type of applications. In this model, a transaction consists of a number of sub-transactions to be executed in sequence and committed independently. In the event of a failure of a sub-transaction, all previously executed sub-transactions are compensated (undone) in the reverse execution order and the transaction is aborted. Such an approach may be very conservative in several practical scenarios resulting in a loss of useful work. In most workflow applications, the workflow can be rolled back partially to some previous state and forward execution can continue from there. Also, since the steps of workflows are often loosely related it is not necessary to always compensate the steps in the reverse execution order. Thus, it should be possible to achieve more flexibility in handling rollback requirements.

What was 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 the steps 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 workflow systems do not have provision to account for such possibilities. The concept of spheres of joint compensation that was discussed earlier in section 2.1.2.2 does not consider interaction between the steps of the sphere and other steps outside of the sphere. To achieve better performance while achieving the required correctness, more emphasis has to be placed on the interaction between the steps.

Another form of interaction between the forward executions of concurrent steps in a single workflow can cause inconsistencies. For example, consider *Figure 2.3* which shows the different threads of forward execution and rollback in a workflow and illustrates a race condition that develops at a *confluence step* (step where two or more branches meet) due to a failure. After step S2 is executed, control is passed along both branches (AND branching), resulting in two concurrent threads of execution. If the execution of the different threads within a workflow are not tracked, a race condition can occur. Thus, in the context of workflows, a race condition is a situation where a step is triggered incorrectly after receiving improper control flow signals. Such race conditions lead to data inconsistencies. 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 the thread be rolled back to step 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 new thread along the bottom path. In this case, step S8 will be triggered using information from *different*

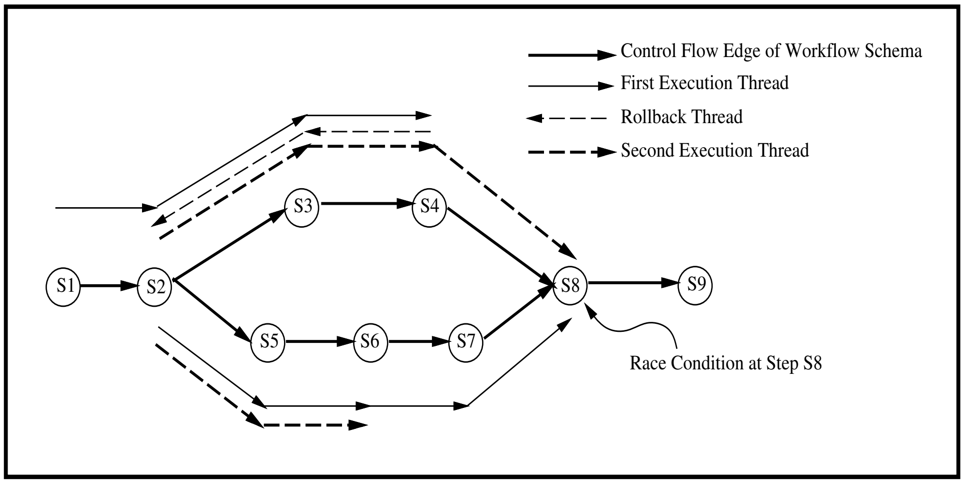

Figure 2.3 Race Condition arising from a Rollback in a Workflow

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 use information from *different executions* of step S2. Hence, uncontrolled interaction of forward execution and rollbacks within a workflow can cause data inconsistencies. Such race conditions can be problematic and have not been considered in current WFMSs. Note that such race conditions are relevant to customized transaction models [48] as well.

A WFMS must provide support at the system level to automatically handle race conditions. On the other hand, to customize rollback and reduce unnecessary compensation and re-execution, support is needed both at the modeling level and the system level.

2.2.3 Need for achieving idempotency when agents fail

In the earlier discussion on system failures, solutions were proposed to handle failures of the workflow engine and the workflow database. In this section the focus will be on the failure of the remaining WFMS component, the application agent.

Since workflow agents are stateless (they do not store history information), failures at the workflow agent can cause idempotency problems. Consider a scenario

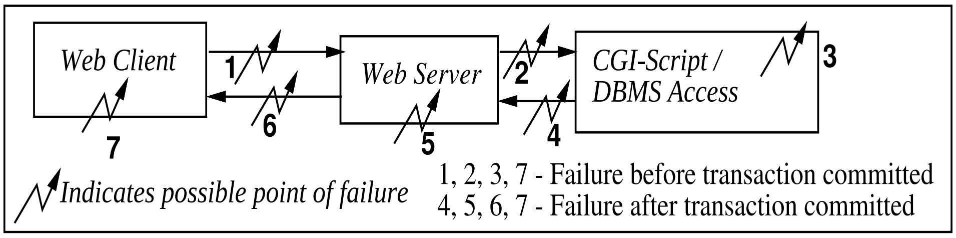

Figure 2.4 Failure of Update request on the Web

where the failure of the workflow engine is followed by the failure of the agent. The agent may be supervising the execution of a program that performs a step. After the program terminates successfully, the agent may be unable to communicate the completion of the step and the results produced to the workflow engine that has failed. Before the workflow engine restarts the agent could also fail. The workflow engine will try reconnecting to the agent. Unless proper care is taken, the results of the steps that completed execution between the two failures will be lost. This results in an inconsistency since the state of the step as recorded in the workflow database will indicate that the step is still 'executing', even though the program has executed and the step has been actually completed. If the engine times out, it may start the step again although it has already been executed. This can result in a duplicate execution which is to be avoided. As shown in figure 2.4, the same phenomenon can be observed in Web based workflows where the unreliability of the Web can cause idempotency problems. While handling steps that update Web information sources, *i.e.*, after a step is scheduled, if a response is not received within the timeout period from a remote Web server, without additional mechanisms it is not possible to determine whether the request was processed or not. If the step is scheduled again, there is a possibility of duplicate execution. Hence, there is a *need for mechanisms to achieve idempotency in the event of agent failures.*

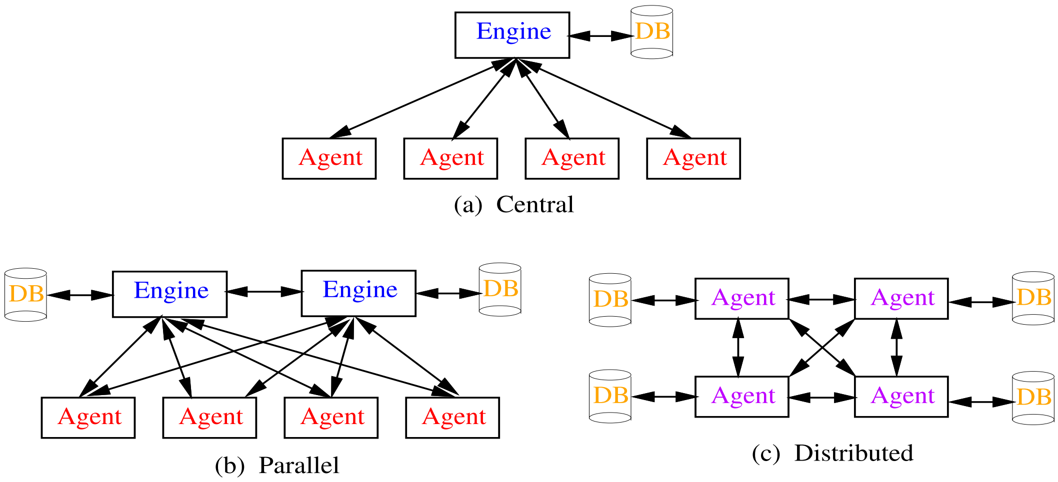

Figure 2.5 Central, Parallel and Distributed Workflow Environments

2.2.4 Need for avoiding centralized control of workflows

In a WFMS with a centralized control, all workflow instances are controlled from a single node. Hence, the state information of all workflow instances (global state) is on a single node. This provides a number of benefits. For example, tracking the state of a workflow is easy. Also, it is easy to determine whether there are conflicts/dependencies between steps of different workflows since the step inputs and states can be easily determined. However, a central workflow engine has scalability problems. In large organizations, usually hundreds of thousands of workflow instances may be executing concurrently [72]. A WFMS where all application agents communicate with a central engine can be a performance bottleneck when the workflow engine has to handle such large workloads. As shown in *Figure 2.5*, two other alternatives are possible — *parallel* and *distributed* workflow environments.

A parallel workflow environment has several workflow engines. A workflow instance is controlled by only one engine, chosen based on the load at the different workflow engines. Workflow instances do not migrate from one engine to another. An engine communicates with several agents to execute the steps of a workflow instance. An agent performs steps for multiple engines. Each workflow engine has its own workflow database to maintain scalability. The state of a single workflow is available at the engine. Hence, tracking the state of a workflow is easy. However, the global state of all the workflow instances is distributed across the engines. This makes it difficult to determine whether there are conflicts/dependencies between steps of concurrent workflows managed by different workflow engines. Hence, inter-engine communication is required to determine if there are conflicts/dependencies between steps of such concurrent workflows. As shown in figure 2.6, the format of the messages exchanged by the workflow engines with the agents is the same as for a central engine. It contains the workflow identifier, instance identifier, step identifier and the corresponding action and the inputs to that step. As in a central workflow environment, the entire state of a workflow is available at a single node (the engine) and hence workflow commits merely require the checking the states of the individual steps as stored in the workflow database associated with the engine.

In a distributed workflow environment, the workflow instances are scheduled and coordinated by the agents themselves. Hence, a distributed workflow control architecture has good scalability. Since these agents not only execute the step but also handle the scheduling, they store the workflow state information in an agent database. To execute a workflow instance, agents that are responsible for executing the steps of that workflow instance have to communicate with each other. After the execution of a step, an agent has to communicate the entire state information of the workflow that it is aware of to the agent responsible for executing the next step. This information is communicated via a *workflow packet*. As shown in figure 2.6, the format of the messages exchanged between agents is more complex. It contains the workflow identifier, instance identifier, and the state information of that instance. This state information accumulates as the workflow progresses from one step to another (one agent to another). Thus the contents of a workflow packet includes the contents of the workflow packet received by the agent (request for performing that step) and the output produced by the execution of the step at

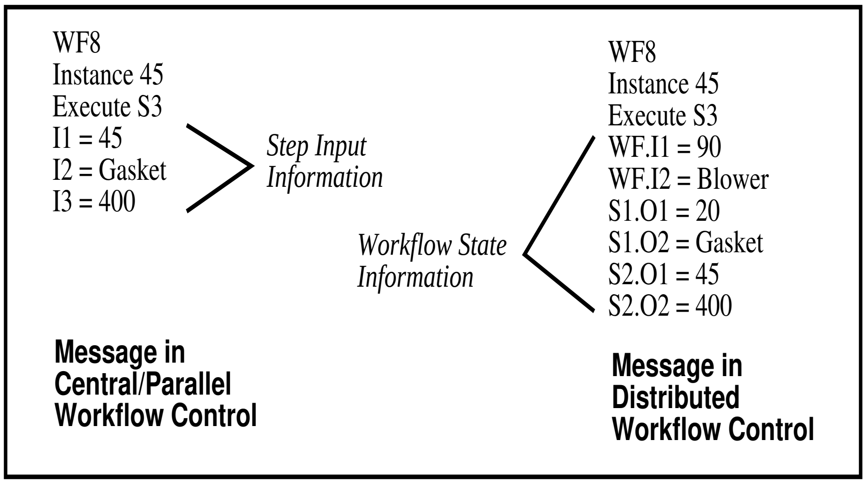

Figure 2.6 Message Formats in Different Workflow Environments

the agent. Each workflow has a coordinator agent which is typically the agent responsible for executing the first step of the workflow. This coordinator agent is not responsible for navigating the workflow, but only for handling workflow commit and abort. In a distributed environment, the state information of a *single* workflow is distributed across agents. Hence, tracking the state of a workflow is complicated. To determine which step of a workflow is being performed at a given instant, a chain of probe messages has to be sent starting from the agent responsible for performing the first step until the message reaches the agent that is performing the current step. Since the global state of workflow instances is also distributed across agents, it is complicated to determine whether there are conflicts/dependencies between steps of concurrent workflows managed by different workflow engines. Other than workflow packets, additional inter-agent messages are required to determine whether there are conflicts/dependencies between steps of concurrent workflows. Also, a workflow commit requires more work since the coordinator agent for a workflow has to communicate with agents responsible for executing the terminal steps of each thread of execution.

To summarize, inter-engine and inter-agent communication is required for coordinated execution of workflows in parallel and distributed environments respectively.

This communication is required to query/update state information of workflow instances. The messages are exchanged in addition to messages flowing in the network for routine execution of steps. To maintain scalability, it is necessary to device schemes that reduce the number of additional messages exchanged for coordinated execution.

2.3 Summary

In this chapter we provided a brief overview of WFMSs by focusing on the workflow and system model and presented the various problems that can arise in state-of-the-art workflow systems. The following are our key objectives:

- handling coordinated execution of steps from concurrent workflows — mutual exclusion, relative ordering and rollback dependency.

- handling failures within a workflow — controlling interactions between rollback/forward execution of steps and preventing race conditions.

- achieving scalability in workflow execution — designing scalable mechanisms for handling normal execution, coordinated execution and failures under parallel and distributed workflow control architectures.

- achieving idempotency of steps in the event of an agent failure — both in traditional WFMSs and in Web WFMSs.

To summarize, the following are the correctness and performance requirements:

2.3.1 Correctness

We classify correctness requirements into two categories based on the occurrence of failures.

2.3.1.1 Normal Execution

In the absence of step failures, it is necessary to enforce (i) mutual exclusion and (ii) relative ordering of steps from concurrent workflows to ensure consistency of accesses to shared resources.

2.3.1.2 Failure Cases

Failure of a step can affect other steps in the same and other workflows. Also, agents in current WFMSs do not store state information that can cause idempotency problems. Thus it is necessary to ensure that (i) steps are compensated and re-executed in the proper order by satisfying the data dependencies, (ii) race conditions are avoided during workflow rollback and re-execution, (iii) rollback dependency requirements are to be satisfied by steps from concurrent workflows and (iv) idempotency of steps is to be ensured in the presence of agent failures.

2.3.2 Performance

We classify performance issues into two categories based on the granularity at which it is examined. Both these are needed to improve efficiency and achieve scalability.

2.3.2.1 Workflow Level

Minimizing the amount of lost work is crucial to achieving better efficiency. This is particularly important in the context of compensation and re-execution of steps in a workflow on the failure of a step. To reduce resource consumption unnecessary step compensation and re-execution is to be avoided.

2.3.2.2 System Level

Scalability is very important since it affects the progress of not just one but all the workflows in the system. Hence, to avoid the performance bottleneck caused by a centralized control, parallel or distributed control architectures have to be used. Under these new architectures, new protocols are needed to enforce the correctness requirements discussed earlier. In addition, to maintain scalability the number of additional messages entailed by these protocols have to be reduced.

The rest of the thesis describes the concepts and infrastructure developed to address these problems.

CHAPTER 3

Approach — Concepts and Algorithms

In this chapter the CREW (Correct and Reliable Execution of Workflows) approach [74] is described to handle the different problems motivated in the previous chapter. First, we state our assumptions and then provide a broad overview of the CREW approach. Then specific concepts and techniques involved in coordinated execution, failure handling, achieving idempotency and decentralized control of workflows are presented. The run-time interactions between coordinated execution and failure handling techniques are also analyzed. Detailed implementation of these techniques are presented in chapter 6.

3.1 Assumptions

The workflow model assumed is summarized in section 2.1.2.3. It is a superset of the modeling features provided by most state-of-the-art commercial and research WFMSs. Hence, it is necessary to consider the impact of any additional features added to the existing modeling features. It is also assumed that the workflow/application designer has knowledge of the data resources accessed by the individual steps/applications. This information is necessary to automatically deduce conflict conditions between steps of workflows.

For the run-time system, it is assumed that the network connecting the workflow engine and the agents is reliable. Hence, the algorithms and mechanisms developed in this thesis will focus on logical failures within a workflow and engine/agent failures. It is also assumed that multicasting is supported by the network connecting

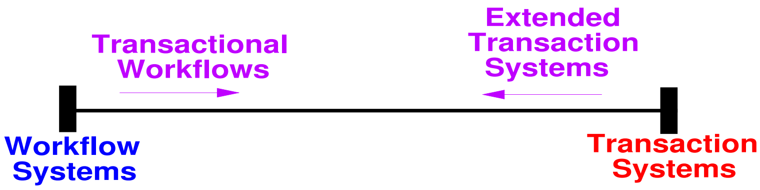

Figure 3.1 Support for Workflow Applications

the engines and the agents. This will useful in the design of efficient mechanisms to handle engine/agent failures.

3.2 Overview of CREW Approach

Workflow applications are *long-duration* applications since the duration of a workflow can range from a few hours to a few months. As shown in *Figure 3.1*, support for *long-duration* applications has been independently developed by researchers focusing on workflow systems and transaction systems. *Extended transaction systems* structure a large transaction into sub-transactions and execute them with additional constraints on the individual sub-transactions, *e.g.*, *precedence* requirements between start/commit/abort of the sub-transactions. However, in practice not all steps that make up an application are transactions. Researchers in workflow systems have proposed the notion of *transactional workflows* [120] where additional correctness requirements can be specified on top of traditional workflow specifications. The CREW approach is based on this philosophy. Workflow steps are not restricted to be mere database transactions nor is the entire workflow treated as a single transaction. Instead, additional correctness requirements are specified that constrain the workflow executions in a desired manner and enhance the correctness properties of workflows. The additional constraints will produce fewer schedules than unconstrained execution of workflows. This chapter discusses concepts that address correctness and performance requirements of workflow execution to support transactional workflows.

We first address failure handling issues in the context of individual workflows. By considering the interaction between the rollback of steps and the forward execution of steps from the perspective of both correctness and performance, an integrated approach is suggested. WFMSs have no knowledge of resources accessed by the applications. Hence, workflow designers have to directly specify failure handling and coordination execution requirements. However, this can be a cumbersome process. To simplify the process, the proposed scheme allows workflow designers to specify resources accessed by each workflow step. From this specification and workflow schema information, the system determines the type of failure handling and coordinated execution requirements for a given workflow. Later at run-time, failure handling requirements are enforced within a workflow while coordinated execution requirements are enforced across workflows.

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 a rule based approach is ideally suited for managing workflows in WFMSs. Rules are of the standard E-C-A (Event-Condition-Action triplet) format. Upon the completion of a step, a step completion event is generated. A step may be eligible for execution on the occurrence of one or more required step completion events (*composite event*). Once the composite event occurs, optionally a condition has to be evaluated. The condition typically comprises of predicates based on the outputs produced by the steps whose step completion events are needed for the occurrence of the composite event. If the specified condition evaluates to TRUE, then the step is eligible for execution. The actual execution of the step is performed by the agent responsible for executing the step.

We have developed a workflow specification language LAWS (LAnguage for Workflow Specification), that allows a workflow designer to not only state traditional workflow requirements like control and data flow but also enhanced requirements for

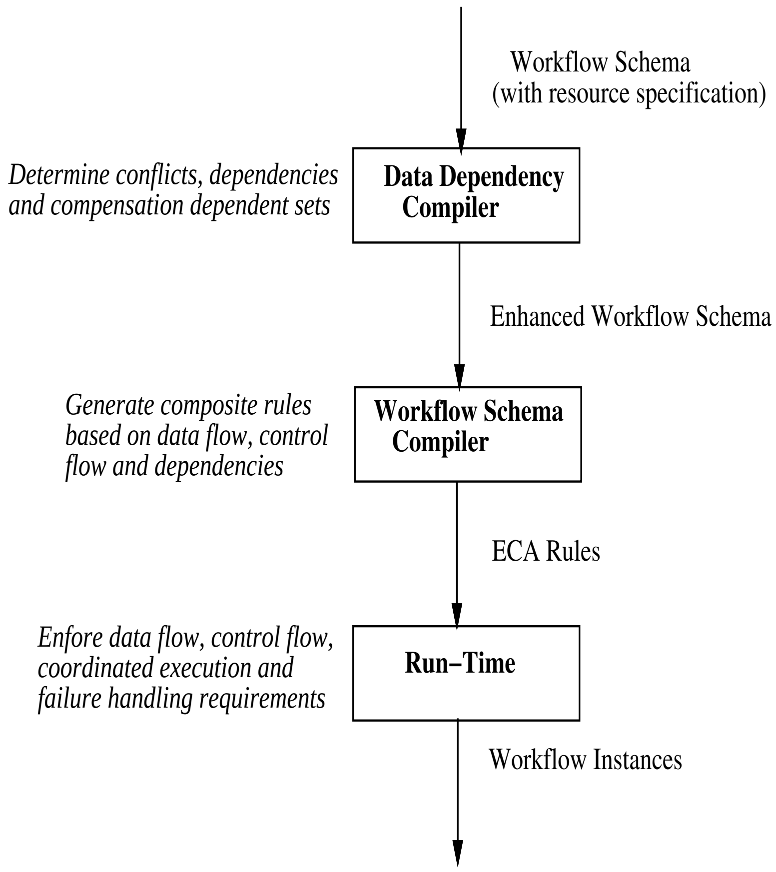

Figure 3.2 Workflow Compilation and Execution

failure handling and coordinated execution. Details of LAWS are provided in chapter 4. We have developed a two stage compiler for translating a user provided workflow schema into a set of rules. As shown in *Figure 3.2*, the first stage of the compiler is responsible for determining conflicts and dependencies between steps within a workflow and across workflows. The compiler refers to the compiled definitions of existing workflow schemas while determining dependencies. The first stage produces an enhanced workflow schema that contains the original schema information and additional specifications for failure handling and coordinated execution. During the second stage the enhanced workflow schema is translated into rules. Details of the compiler are provided in Chapter 5. At run-time, the rules are interpreted by the workflow engine in the case of central/parallel workflow control and the workflow agents in the case of distributed control to handle workflow executions in accordance with their requirements. Details of the run time mechanisms are provided in Chapter 6.

The techniques developed for failure handling and coordinated execution can have an effect on each other as well as other existing modeling features. Hence, it is important to study the potential interactions caused by these techniques. Later in the chapter, we discuss the possibility of such interactions and analyze their effects.

3.3 Handling Coordinated Execution

This section presents a high level discussion of how the different coordinated execution requirements are handled.

3.3.1 Mutual Exclusion

Mutual exclusion requires that steps which conflict based on the resources they access be executed in a non-overlapping fashion. As we discussed earlier in chapter 2, mutual exclusion across workflows may be needed not just for a *single* step but a *group* of steps. The algorithm for handling mutual-exclusion is shown in figure 3.4. Handling mutual exclusion for a single step is fairly straight forward. Before the execution of that step, it is verified that no other steps that are currently in execution while accessing the same resource(s). If there are no conflicting steps, then the step can execute. If there are other conflicting steps that are executing then the step has to wait for the conflicting steps to complete. Essentially, the engine holds a lock on behalf of the workflow instance that contains the step that requires mutual exclusion. This technique can be extended to the case where mutual exclusion is required for a group of steps. The workflow engine keeps track of the resources accessed by the steps of the group and prevents the execution of any other step that needs resources accessed by the group. All such steps are delayed until the completion of the group.

```
if (there are conflicting steps in execution)
    wait for conflicting steps to complete;
    execute step ;
else
    block other conflicting steps from executing;
    execute step ;
```

Figure 3.3 Algorithm for mutual exclusion of step execution

```
enforce mutual exclusion of first pair of conflicting steps;
foreach (subsequent conflicting pair in workflow)
    wait for completion of conflicting member from other workflows;
    execute step;
    notify completion of step to other waiting workflows;
```

Figure 3.4 Algorithm for relative ordering of steps

3.3.2 Relative Ordering

Relative ordering requires that steps belonging to conflicting pairs from concurrent workflows have the same precedence order. If the workflow designer has relaxed (dropped) some of the ordering requirements, then only some of the conflicting pairs need to execute in the same precedence order. To enforce this requirement the workflow engine has to maintain state information about the precedence order between the first pair of conflicting steps. This precedence order becomes the precedence order between the respective workflow instances. Steps belonging to subsequent conflicting pairs then have to be scheduled in the same precedence order. To enforce relative ordering it is necessary to know the order in which steps from conflicting pairs have to be scheduled. Hence, to determine the precedence order between the first pair of conflicting steps it is necessary to execute them in a mutually exclusive fashion such that a definite order is established between the steps.

Relative ordering requires more coordination between workflow instances than mutual exclusion does. Consider two workflow instances whose first set of conflicting

if (step has rollback dependency)
    if (step from other workflow that caused the dependency is compensated)
        perform rollback/re-execution in current workflow;

Figure 3.5 Algorithm for rollback dependency of steps

steps have been executed in a certain order. The relative order between the subsequent steps depends on the branching taken and the presence of failures in either of the workflow instances. For instance, step X in workflow A could be waiting for the completion of a conflicting step Y of workflow B. However, if a branching path is taken in workflow B such that Y will never be executed then step X in workflow A need not wait anymore. Thus execution of steps other than those in the conflict pair can have an effect on relative ordering. Details of this technique are provided later in section 6.1.2.

3.3.3 Rollback Dependency

A rollback dependency between steps of concurrent workflows is established when two steps from concurrent workflows access resources in conflicting modes and a rollback of the step executed first (leading step) affects the workflow instance that contains the step that executed later (lagging step). Once a rollback dependency has been established, if the step that executed first is compensated the other workflow must be notified about the compensation. This allows the other workflow to take appropriate action. Upon the receipt of the notification, the current thread of execution is halted just as in the case of failure handling. Then, appropriate rollback action has to be taken. Hence, an additional message exchange is needed to track the end point of the current thread of execution in parallel and distributed workflow environments. The technique described earlier to halt threads for handling race condition is applicable here. Details of the mechanisms for handling rollback dependency in central workflow control are provided in section 6.1.2.

3.4 Failure Handling Techniques

3.4.1 Integrated Approach for Partial Rollback and Re-Execution

As discussed in chapter 2 an integrated approach to handling partial rollback and re-execution is necessary in terms of minimizing the amount of lost work. As part of the CREW approach, we propose a new scheme called *opportunistic compensation and re-execution* which eliminates unnecessary compensation and re-execution of steps when workflows are rolled back partially and re-executed. The basic idea is to decide whether to compensate and re-execute a step dynamically during workflow re-execution. This decision is based on the dynamic evaluation of a compensation and re-execution condition associated with that step.

Two types of compensation are possible — *normal* and *partial* and two types of re-executions are possible — *normal* and *incremental*. A normal compensation totally annuls the effect of the previous execution of the step. Similarly a normal re-execution corresponds to re-executing a step with its new inputs. A partial compensation does not totally annul the effect of the previous execution of the step. Instead only some part of the step is undone. The reasons for having these different types of compensations and re-executions will become clear in the ensuing discussion on their use with simple examples.

The same techniques are also useful in situation where workflow inputs are changed while a workflow is in progress. An example of such a situation is an order processing workflow where the customer changes the details of the order after it has been placed. Steps in the workflow that have been completed have to be revisited to check if they have to be compensated and re-executed. While these situations do not correspond to failures, the actions required are very similar to those of a partial rollback and re-execution.

In this thesis, partial compensation and incremental re-execution are discussed primarily based on numerical inputs to steps. However, the concept can be generalized to include non-numerical inputs. Based on the evaluation of the re-execution condition, one of several different compensation programs can be chosen to compensate a step. Similarly one of several programs can be chosen to re-execute the step. The basic idea is to provide flexibility in determining how a step is to be compensated and re-executed such that work achieved by the previous execution of the step can be used to the best extent.

3.4.1.1 Normal Compensation and Re-execution

A normal compensation and re-execution corresponds to a normal compensation followed by a normal re-execution. This techniques is useful in cases where the previous execution of the step is useless in the current context. One example of such a scenario is an order processing workflow where, due to the failure of a step (in completing an order for a component) other previously executed steps (that fulfilled orders for related components) are undone. Another example is a change made by a customer when a order has been partially fulfilled by the order processing workflow. Depending on the nature of the change, the previously executed steps have to be completely undone and new ones have to be executed.

3.4.1.2 Incremental Re-execution

In the case where the previous execution of the step is useful, an incremental re-execution of the step may be sufficient to produce an effect equivalent to the normal compensation and re-execution of the step with the new inputs. Note that the effect of the previous execution of the step is not compensated since it is still useful. An example of such a scenario is an order processing workflow where due to a change requested by a customer an incremental re-execution of the step is necessary, *i.e.*, if a step was executed successfully to order 50 items of component X and if the change in the order requires 75 items of component X, the step can be incrementally re-executed to order 25 items of component X. If intervening orders are present, the additional 25 components may not be available. Note that the re-execution is based on a value that is the difference of the new and previous inputs. If a traditional failure handling approach is taken in such a scenario, compensating the previous step (cancelling the previous order) and executing the step with the new inputs will result in the waste of resources.

3.4.1.3 Partial Compensation

Similarly, there are cases where the previous execution of the step is useful and a partial compensation of the step is sufficient to produce an effect equivalent to the normal compensation and re-execution of the step with the new inputs. In some sense, this is the inverse of the incremental re-execution discussed earlier. As in the case of incremental re-execution, if a traditional failure handling approach is taken, compensating the previous step (cancelling the previous order) and executing the step with the new inputs will result in wastage of resources.

Note that partial compensation is different from incremental re-execution. In partial compensation, a portion of the effect of the previous execution of the step is compensated such that the cumulative effect of the partial compensation and the previous execution is the same as executing the step with a new set of inputs. In contrast, in incremental re-execution, the step is executed again such that the cumulative effect of the incremental re-execution and the previous execution is the same as executing the step with a new set of inputs. Both these techniques benefit from the fact that the invocation of one program is sufficient enough to produce the desired results rather than executing two programs, one for compensation and another for forward execution. In an abstract sense, partial compensation and incremental re-execution are similar in the sense that the new inputs are determined before the step is executed and a single program is to be executed. Hence, the run-time support needed will be the same. This issue is discussed further in chapter 6.

3.4.1.4 Compensation Dependent Sets

Under opportunistic compensation and re-execution, steps are not always compensated in the reverse execution order so as to delay the compensation of a step until it is decided that the step has to be compensated and re-executed. However, in certain cases there may be data dependencies between steps which require that the steps be compensated in the reverse execution order to maintain consistency. To customize the order in which the steps are compensated, *compensation dependent sets* can be specified. A compensation dependent set is one in which the compensation of one step affects the others. A compensation dependent set is to be compensated only in the reverse execution order of its member steps. If the steps are executed in some partial order then the compensation is also performed in the reverse partial order.

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 re-execution thread passes through that step, then it has to be re-executed – irrespective of the evaluation of the re-execution condition. The notion of compensation dependent sets is different from the notion of spheres of joint compensation [86]. A compensation dependent set is used to specify the order of step compensation in the context of the opportunistic compensation and re-execution strategy. The semantics of spheres of joint compensation is that when a step belonging to a sphere fails, all other steps that belong to the sphere must be immediately compensated. Thus there is no integrated treatment of rollback and re-execution in the context of

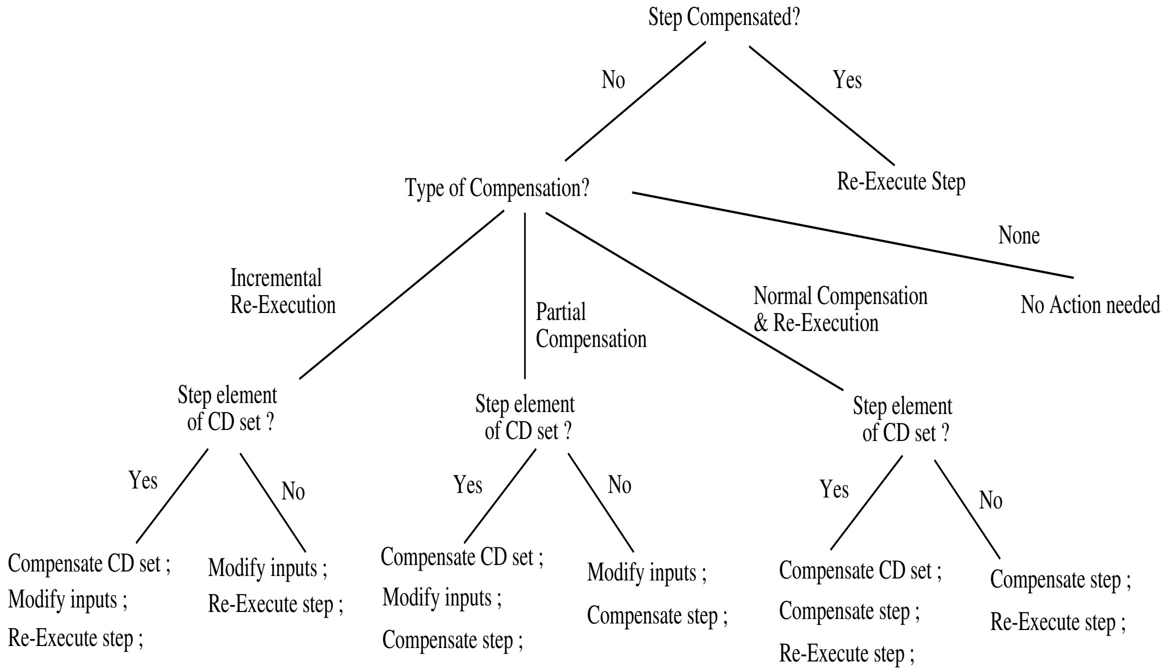

Figure 3.6 Algorithm for Opportunistic Compensation and Re-execution

the latter. By defining compensation dependent sets only the necessary steps need be compensated in the reverse execution order.

3.4.1.5 Opportunistic Compensation and Re-Execution

The algorithm for supporting opportunistic compensation and re-execution (OCR) is shown in figure 3.6. Each step is associated with a compensation and re-execution condition. This condition is a statically specified predicate based on the current and previous inputs to the step. Using the workflow state information this condition is dynamically evaluated before a decision is made about how the step is to be compensated and re-executed. The algorithm considers different cases depending on whether a step has been compensated or not. If a step has not been compensated then it is to be compensated and then re-executed. The appropriate action is taken depending on which of the three conditions (*i.e.*, partial compensation or incremental re-execution or the normal compensation and re-execution condition) evaluates to TRUE. Alternatively, if a step has already been compensated since it is part of a compensation dependent set that was rolled back, it is re-executed. The conditions are evaluated from left to the right as shown in in the algorithm and the compensation and re-execution decision is made based on whichever evaluates to true first.

To illustrate the concepts of opportunistic compensation and re-execution and the notion of compensation dependent sets, some simple examples are provided below. The right side of the implications ($\Rightarrow$) denote histories H where E, C, F refer to execution, compensation and failure respectively of the indicated step. Also, P refers to partial compensation and I refers to incremental re-execution.

We will first discuss cases where normal compensation and re-execution are needed. Consider a sequential workflow with 4 steps – S1, S2, S3, and S4. Let us say a failure occurs at step 4 which will be indicated by F4 in the history. Also the failure handling requirement for step 4 requires the workflow to be rolled back to step 2. Step 2 will be compensated and re-executed after which the rest of the workflow execution continues. In the absence of compensation dependent sets the following two histories illustrate the use of the compensation and re-execution condition:

Normal Compensation and Re-execution condition for Step S3 = TRUE

$\Rightarrow$ H = E1 E2 E3 F4 C2 E2 C3 E3 E4

Normal Compensation and Re-execution condition for Step S3 = FALSE

$\Rightarrow$ H = E1 E2 E3 F4 C2 E2 E4

In both these cases, it is assumed that the incremental re-execution and partial compensation is not possible (examples of these follow). Note that if the re-execution condition for step S3 is false, it is not compensated and re-executed again. Thus the strategy reduces resource consumption thereby improving performance at the workflow level. The benefits will actually depend on the nature of the step. They will be considerable for steps requiring human action. For example, if the step involves transferring material from one warehouse to another, then a considerable amount of savings can be realized by reducing the loss of completed work. Notice that the benefits are obtained while the correctness is unaffected, *i.e.*, steps that require compensation in the reverse execution order are still compensated in the reverse order. Essentially, the benefits are achieved by delaying the decision to compensate and re-execute some of the steps until it is known how and what is to be compensated and re-executed.

Let us revisit the example when compensation dependent sets are specified. Consider steps S2 and S3 to be part of a compensation dependent set. A failure occurs at step 4 which will be indicated by F4 in the history. Also the failure handling requirement for step 4 requires the workflow to be rolled back to step 2. Step 2 will be compensated and re-executed after which the rest of the workflow execution continues. Then the execution of the same sequential workflow would be as follows:

$\{S2, S3 \in CDS\}$

$$\Rightarrow H = E1\ E2\ E3\ F4\ C3\ C2\ E2\ E3\ E4$$

Compensation dependent sets are specified if data access dependencies exist between the steps. In the example note that step S3 is compensated before step S2 due to the compensation dependency.

The next discussion considers cases where partial compensation and incremental re-execution is needed for steps. Consider a sequential workflow with 4 steps – S1, S2, S3, and S4. Also, let steps S2 and S3 be part of a compensation dependent set. Let us say a failure occurs at step 4 which will be indicated by F4 in the history. Also the failure handling requirement for step 4 requires the workflow to be rolled back to step 2. Step 2 will be compensated and re-executed after which the rest of the workflow execution continues. Then the execution of the same sequential workflow would be as follows:

$\{S2, S3 \in CDS\}$

Incremental Compensation condition for Step S3 = TRUE

$\Rightarrow$ H = E1 E2 E3 F4 C2 E2 I3 E4

Note here that C3 never happened and I3 happened instead of E3. Partial Compensation condition for Step S3 = TRUE

$\Rightarrow$ H = E1 E2 E3 F4 P3 C2 E2 E4

Note here that E3 never happened (in re-execution phase) and P3 happened instead of C3.

These specific examples from a sequential workflow give an idea of how opportunistic compensation and re-execution works. The technique is general enough to handle arbitrary set of steps. Consider the failure of an arbitrary step j in a workflow. The failure handling specification for step j may require the workflow to be rolled back to some previous step i and re-executed from there. All intermediate steps that executed between the time i was executed and j was attempted are compensated and re-executed individually based on their compensation and re-execution conditions. This is irrespective of whether the workflow has sequential flow, parallel flow or conditional flow. If such intermediate steps are members of compensation dependent sets then the appropriate steps are compensated in the reverse execution order.

3.4.2 Handling Race Conditions

As discussed earlier in Chapter 2, race-conditions are possible when there are parallel branches of execution in a workflow. The workflow designer does not need to explicitly specify any additional information for handling race conditions.

We use the algorithm shown in Figure 3.7 to handle failure along a parallel branch in a workflow. This is one part of the solution for handling race conditions. The key factor is determining if one or more parallel branches will be affected by a failure in a branch. This is directly dependent upon whether the failure requirement of a step in a branch requires rollback to the branching point or a step that precedes the branching point. If the failure of the step requires a workflow abort then the specified steps are compensated and the workflow aborted. However in most cases, the workflow will be rolled back to some previous step and re-executed from there. Referring to *Figure 2.3*, if the failure requirement of step S4 requires a rollback only to step S3 then the other branch will not be affected. If this is the case, execution continues along the other branches without any interference. Instead, if the rollback is to step S1 or S2 then other parallel branches will be affected. In this case, other concurrent threads that originated from the branching point are halted, *i.e.*, no more steps in the affected branches are scheduled for execution. If control flows along other branches has already reached the confluence step then there is no need to halt the threads. After the branching step has been compensated and re-executed, new execution threads are created for the various parallel branches. The opportunistic compensation and re-execution scheme is applied along the branches to determine which of the steps are to be compensated and re-executed. Thus logical failure of a step is handled correctly without causing inconsistencies in the execution of other steps.

Race conditions are avoided by ensuring that the results of the steps that have been executed but are affected by a rollback are invalidated. A step will be affected by a rollback if it is a successor (not just immediate but transitively as well) of the rollback step. The successor relationship between the steps is determined using a matrix that is produced statically during the compilation of a workflow schema. Once the results of the affected step is invalidated, the events and results produced by that step cannot be used to trigger the execution of other steps. Since race conditions are caused by the use of events and data produced by steps affected by a rollback, race conditions can be avoided by invalidating the results of the affected steps.

```
if (rollback does not reach branching point)
    Use OCR strategy along the failed branch ;
else
    foreach (branch containing successors of rollback step)
        halt execution thread along the branch ;
    wait for compensation/re-execution of branching step ;
    foreach (branch affected by rollback)
        Use OCR strategy along the branch ;
```

Figure 3.7 Handling failure along a parallel branch using OCR

3.5 Achieving Idempotency of Steps in the event of Agent Failures:

This problem of ensuring idempotency of a step in traditional WFMS can be ameliorated by having logging facilities at the application agents and using the EONS (Exactly ONce Semantics) protocols for idempotency 3.8. In the EONS protocol a unique ID is associated with each step execution request. Before a request is sent from the engine a unique ID is generated for it and is persistently stored. Then the request is tagged with the unique ID and sent to the agent. In traditional WFMS, the workflow name, instance number and step name forms the unique ID. For Web WFMSs, a unique ID is generated explicitly using the host address information. Every agent persistently stores step execution information along with the outcome of the step. The agents can be enhanced to store the persistent information in a database. When the agent comes up after a failure, if it receives a repeat of a request that it has already processed then it can pass on the results of the completed execution of the step to the workflow engine without re-executing the step and causing data inconsistencies. If the client does not get a reply in the specified timeout period, the engine resends the request with the same unique ID. Details of the mechanisms for handling agent failures in central workflow control are provided

```
At the workflow engine:
    generate unique ID for request;
    make ID persistent;
    schedule step for execution;
    mark step as scheduled in workflow database;
    while response has not been received {
        resend request to the agent with step ID;
    }
At the agent:
    using step ID check persistent store if step has been processed;
    if step has not been processed {
        log step inputs;
        execute step;
        log step outcome;
        send reply with outcome;
    }
    elseif step execution is in progress {
        allow step to complete;
        log step outcome;
        send reply with outcome;
    }
    else {
        get details of processed step and send reply;
    }
```

Figure 3.8 EONS Protocol to ensure idempotency of Step Execution

in section 6.1.4, for distributed workflow control in section 6.4.5 and for Web WFMSs in section 7.2.

The EONS protocol is quite similar to networking retransmission schemes like *Stop-and-Wait* and *Go-Back-n* [24]. These protocols associate a counter with each data frame. Using the counters, the networking software ensures that each packet is released to higher levels of software exactly once, just as the algorithm ensures that each update is performed exactly once. The main difference is that the EONS protocol requires that this exactly-once guarantee hold even if the connection between the two parties is broken for an arbitrarily long period of time. Consequently, it requires persistent storage of request information, while the networking protocols do not.

3.6 Decentralized Control of Workflows

In this section we discuss how the concepts used for coordinated execution and failure handling need to be extended for decentralized workflow control.

Ensuring mutual exclusion for a single step or a group of steps is easy when all instances are controlled by a single workflow engine, because all state information is available on a single node. However, enforcing mutual exclusion in parallel and distributed workflow control environments is more involved. In parallel workflow control, the global state is split amongst the various workflow engines. Hence before a step is executed it is necessary to ensure that none of the other workflow engines/agents have scheduled conflicting steps. If conflicting steps are in progress at one or more sites, then the step has to be delayed until all such conflicting steps complete. Hence there are additional exchange of information between the workflow engines. In distributed workflow control, a single workflow's state is distributed amongst various agents. Hence, when mutual exclusion is needed for a group of steps in distributed workflow control, it is necessary to pass additional information between agents involved in executing the group. This will help the agent executing the last step of the group to notify the agent(s) waiting for a notification. Details of the mechanisms for handling mutual exclusion in central workflow control are provided in section 6.1.2, for parallel workflow control in section 6.3.1 and for distributed workflow control in section 6.4.3.

As in the case of mutual exclusion, relative ordering in parallel and distributed workflow control environments is more involved. Mutual exclusion of the first pair is achieved as described in the previous section. A single workflow can have several conflicting workflow instances from the perspective of relative ordering. A step in a workflow may be a member of several conflicting pairs. In some of the pairs the step might be the leading member (other member has to follow this step) and in the other pairs the step might be a lagging member (this step has to follow the other step). For all conflicting pairs where the step is a lagging member, the step has to wait for the other member of the conflicting pair to complete. Workflows containing the other member may be controlled by other workflow engines. Thus determining which of the steps from concurrent workflows form conflicting pairs and scheduling those steps in the correct order requires message exchange protocols between the engines. In distributed workflow control, since a single workflow's state is distributed amongst various agents additional communication is required between agents for relative ordering. In addition to message exchange between agents controlling steps from different workflows, additional information is to be passed between agents handling individual workflows. This additional information communicates the relative ordering information about other conflicting workflow instances with respect to a given workflow. This information is used by an agent to wait for messages from agents for completion of member steps of conflicting pairs or notify the appropriate agents on the completion of steps. Details of the mechanisms for handling relative ordering in central workflow control are provided in section 6.1.2, for parallel workflow control in section 6.3.1 and for distributed workflow control in section 6.4.3.

To handle rollback dependency in decentralized workflow control similar enhancements are needed for the mechanisms. Specifically, additional exchange of messages is needed when the steps that are rollback dependent belong to workflows controlled by different workflow engines in the case of parallel workflow control. Also, in distributed workflow control, additional messages are to be exchanged between agents responsible for executing the rollback dependent steps. Details of the mechanisms are presented for parallel workflow control in section 6.3.1 and for distributed workflow control in section 6.4.3.

Enforcing the opportunistic compensation and re-execution strategy is simpler if all the workflow state information is available at a single node as in central and parallel workflow control. However, it is more complicated in the case of distributed workflow control since the state information of an individual workflow is split amongst nodes. To handle compensation dependent sets for example, control has to pass through several nodes for compensating the steps that executed after the step that is being considered for compensation. Details of mechanisms for opportunistic compensation and re-execution are provided in section 6.1.4 for central workflow control and section 6.4.5 for distributed workflow control.

Handling race conditions in a central or a parallel workflow control environment is easier since all state information is available at a single node. In a distributed workflow control environment however, handling race conditions is more complicated since the end point (step that is currently being executed) of the various threads of control have to be located. This is necessary to halt the different threads of execution. It requires starting separate *probe* threads along the the various paths to inform the agents to stop scheduling the subsequent steps. Note that the probe threads can run in parallel with the steps that are to be executed before the branching point. A probe thread travels from agent to agent just as in the normal execution of workflows. Details of the mechanisms for handling race conditions are provided in section 6.1.4 for central workflow control and section 6.4.5 for distributed workflow control.

3.7 Run-Time Interaction between Various Features of the CREW WFMS

We can broadly classify the workflow features into three categories:

1. Traditional Features: Conditional branching, looping and nesting

2.Coordinated Execution Features: Mutual exclusion, relative ordering and rollback dependency

3.Failure Handling Features: Opportunistic compensation & re-execution, handling race conditions

We will now consider the interaction between each of these categories.

Traditional features and failure handling features: The OCR scheme is designed to work with the different types of conditional branching. The workflow is rolledback to some previously executed step irrespective of whether the failure occurred along a sequential path, an if-then-else branch or in parallel branch. Also, the OCR strategy for parallel branches where a probe thread is used to quiesce the flow of control on other affected branches handles race conditions as well. In the case of looping and nesting the set of steps that form the loop or the nested workflow is referred to as a *block*. In programming languages statements within a 'while' loop is considered a block and control cannot enter the block at any random statement although control can exit at any of the statement. A similar approach is used to handle loops and nested workflows, *i.e.*, control can exit from any step within the block to some previously executed step outside the block but a rollback cannot enter a block other than its entry point (first step in the block).

A nested workflow can have multiple levels of nesting. In case of a rollback in a nested workflow, the OCR strategy is applied only to the steps at the same level (same workflow) and not to steps at a higher (parent workflow) or lower level (child workflow). If a step is actually performed by a nested workflow, the step can be partially compensated and re-executed by executing the nested workflow with the modified inputs. In the case of loops, the OCR strategy is applied in a limited manner. More specifically, the steps within a loop in a workflow cannot be used to specify compensation dependent sets due to the ambiguity caused in the semantics. For example, one possible alternative is for the dependent steps of the loop to be compensated in the reverse execution order. Another alternative is to compensate all executions of a step in the reverse order before considering the next step in the set. The former is a more practial alternative and is discussed further in chapter 6.

Traditional features and coordinated execution features: Our techniques for mutual exclusion, relative ordering and rollback dependency are designed to work with different types of conditional branching within a workflow. For example, the techniques can be used to handle mutual exclusion, relative ordering or rollback dependency requirements between steps along parallel branches (threads) within a workflow. Another interesting scenario arises when a relative ordering requirement exists between two workflows and the lagging step exists in an if-then-else branch which is never taken. This can cause the waiting workflow to stall unless events are tracked and notified properly between workflows. This scenario is discussed in detail in chapter 6. In the context of looping and nesting, enforcing mutual exclusion and rollback dependency is straight forward since only one step is involved per workflow. In the case of loops, the run-time system identifies the specific conflicts using a combination of the step name and the loop index.

Failure Handling features and coordinated execution features: The main issue addressed here is how OCR interacts with the coordinated execution requirements. Again, mutual exclusion is simple to handle since it merely involves checking for conflicting steps in the compensation and re-execution phase. In the case of relative ordering, the lagging workflow can stall while waiting for a leading step if the leading step has been compensated as part of the OCR scheme. To avoid such a situation, events related to compensation of steps have to be notified between workflows. Interactions between rollback dependency and OCR is also interesting. A lagging step may be required to roll back since the leading step was rolled back. The lagging step in turn can have its own failure handling specification, causing the

OCR scheme to be triggered in the lagging workflow. These issues are discussed further in chapter 6.

3.8 Summary

This chapter presented concepts and algorithms used to develop solutions to the various problems motivated in the context of failure handling and coordinated execution. Starting with an overview of the CREW approach, we presented solutions for handling coordinated execution of workflows where techniques were discussed for mutual exclusion, relative ordering and rollback dependency. The next focus was on an integrated approach for partial rollback and re-execution where various compensation and re-execution strategies were considered. This was followed by a discussion on the opportunistic compensation and re-execution scheme. Solutions were also described for handling race conditions arising from the failure of steps in workflows and for achieving idempotency of steps in the event of agent failures.

The solutions presented for centralized workflow control were also extended for parallel and distributed workflow control architectures. Recall that, in a parallel workflow control architecture, the state of an individual workflow is on a single node while the global state is distributed across nodes. Hence, the extended solutions require additional message passing of state and event information across nodes. Distributed workflow control requires additional message since the state of an individual workflow is distributed across nodes. The final discussion focused on the run-time interactions between the various features supported by the CREW WFMS.

CHAPTER 4

Language Support — Specification and Semantics

This chapter first describes the details of LAWS, the workflow specification language developed in the CREW approach. It presents the different modeling primitives provided in the language. This is followed by a formal discussion on the semantics of the specification. Finally, the use of the LAWS primitives is illustrated in specifying a sample workflow schema.

4.1 LAWS: Language for Specifying Workflow Requirements

To precisely express the execution requirements of workflows in the CREW approach, a language called LAWS (LAnguage for Workflow Specification) has been developed. Apart from the ability to specify traditional workflow schemas, it also has the capability to capture/express the different failure handling and coordinated execution requirements of workflows at a high level. In the rest of this section, the different components of LAWS are described and its use is illustrated for specifying high level requirements. Note that some of the constructs are used directly by the workflow designer to specify workflow schemas while the rest are used by the data dependency compiler for generating coordinated execution requirements.

4.1.1 Mandatory Specification

The following primitives 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. An example specification is:

WF1 : I1 I2 I3 I4 I5 : O1 O2 ;

which states that workflow with schema name WF1 has 5 inputs and 2 outputs.

2.**(Step Specification)**: This specifies the details of the individual steps including the name of the node (an agent should be running on this node) where the step is to be executed, the name of the program that is to be executed to perform the step, the name of the program that is to be executed to undo (where possible) the effects of that step, the inputs and outputs to the step. An example specification is:

S3 : eternity : prod comp3 : I1 I2 : O1 ;

which states that step S3 is to be performed on agent named 'eternity' and the program to be executed for performing the step is 'prod' and the program to be executed for compensating the step is 'comp3'. The step has two inputs and one output.

Nesting of workflows is also supported. Instead of specifying the node and the programs to be executed, the name of the workflow that is to be executed to perform the step is specified. To compensate the step, a different workflow can be specified or a single compensating program can be specified. An example specification is:

S4 : WF7 WF8 : I1 I2 I3 : O1 O2 ;

which states that step S3 is to be performed by executing WF7. Also, by executing WF8 the effects of step S3 can be rolledback at a later time. The step has three inputs and two outputs which correspond to the inputs and outputs of the nested workflow.

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. Examples of specification are:

S3.I1 = WF.I1 ;

S3.I2 = S2.O1 ;

which states that the first input is obtained from the first input to the workflow and the second input is obtained from the first output of step S2. Note that there is no difference in the way data flow is specified for nested workflows. The WFMS maps the step data to the workflow data and hence data flow is specified in terms of the step inputs and outputs.

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. Examples of specification are:

```
S2: IF S2.O1 > 100 AND S1.O1 > 50 THEN S3
    ELSE IF S2.O1 < 50 THEN S4 ELSE S5 ;
S3: S7 ;
```

The first statement had conditional control branching after step S2 and the next statement has direct control flow from S3 to S7.

Since loops represent another form of control flow, their specification is discussed here. The attributes associated with a loop are the start step, the exit step, loop initialization statements, loop end statements and an exit condition. An example specification is as follows:

```
LOOP S2 S6
    INIT I = 0
    END I = I+1
```

EXIT I > 30 ;

which states that the loop starts at step S2 and ends at step S6. At initialization time I is set to 0 and at the end of the loop it is incremented by 1 and if the value of I exceeds 30 then control flow exits the loop to the successor step of S6.

4.1.2 Failure Handling and Coordinated Execution Specification

The following primitives 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 the opportunistic compensation and re-execution strategy. Items 8 and 9 correspond to the specifications required to realize coordinated execution of workflows. Some of the examples given along with the specifications correspond to figure 4.1 unless a different figure is referred to explicitly.

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 rolled back 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 that 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. The following specification 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 *everytime* the workflow restarts and the execution thread passes through that step. If a re-execution 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.

7. **(Compensation Dependent Set)**: This specifies a set of steps that have compensation dependency with reference to one another. A compensation dependent set (CDS) is to be compensated only in the reverse execution order of its member steps. The following states that steps S3 and S6 form compensation dependent set number 1 which means that S6 is to be compensated (if it has been executed) before

the compensation of step S3.

CDS1: 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 conflict condition evaluates to TRUE. Step conflict specification need be given only for those steps that access the same resource. For example, consider workflows in a company that access design, inventory and manufacturing databases. Step conflicts can be defined independently for (i) steps accessing the design database (ii) step accessing the inventory database and so forth. 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 a workflow conflicts with step S23 of workflow WF2 if the condition (S12.I1 == WF2.S23.I2) evaluates to TRUE. The workflow under consideration and workflow WF2 are referred to as conflicting workflows. Later in section 4.2 techniques are described for automatically generating step conflict specification based on the information about the resources accessed by the steps of a workflow.

9. **(Coordinated Execution Specification)**: This specification defines inter-workflow coordinated execution requirements. There are two types of specification. Both these types of specifications can have have complex conditions attached to them.

These specifications are automatically generated by the data dependency compiler. A workflow designer specifies the need for relative ordering along with the workflow specifications. A workflow designer specifies rollback dependencies along with the step specification.

(1) The first type specifies how actions in a workflow should be performed with respect to actions in a conflicting concurrent workflow. This allows a designer to specify for example 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. Some specific examples are provided below. First Consider the specification for mutual exclusion of steps.

```
IF ((S12 CONFLICTS_WITH WF2.S23)
THEN S12 MUTUALLY_EXCLUDE WF2.S23 ;
```

which indicates that if step S12 of the current workflow conflicted with step S23 of workflow WF2 then step S12 should be executed in a mutually exclusive manner with respect to step S23 of workflow WF2.

For mutual exclusion of a group of steps the specification is as follows:

```
IF ((S12 CONFLICTS_WITH WF2.S23 AND WF2.S24)
THEN S12 MUTUALLY_EXCLUDE WF2.S23 AND WF2.S24;
```

which indicates that if step S12 of the current workflow conflicted with steps S23 and S24 of workflow WF2 then step S12 should be executed in a mutually exclusive manner (as a group) with respect to steps S23 and S24 of workflow WF2.

Next consider the specification for relative ordering of steps. This specification corresponds to *Figure 2.1(a)*):

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

THEN S14 RELATIVE_ORDER WF2.S25 ;

which indicates that if step S12 of the current workflow conflicted with step S23 of workflow WF2 and if step S14 of the current workflow conflicted with step S25 of workflow WF2, then step S14 of the current workflow and step S25 of workflow WF2 should be executed in the same relative order as that of step S12 of the current workflow and step S23 of workflow WF2.

(2) The second type 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 dependencies* between steps of conflicting workflows. Thus, decisions to execute and rollback steps in one workflow can be made based on events in other concurrent workflows. An example of this type of specification (corresponding to *Figure 2.1(b)*) is:

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

which states that if step S12 of workflow WF1 is compensated and step S23 of the current workflow conflicted with and happened after step S12 of workflow WF1, then step S23 of the current workflow should be compensated and the current workflow should be re-started from step S23. Also, note that the conflict definitions specified in the step conflict specification are used in the condition part of the workflow coordinated execution specification.

The syntax for LAWS is included in Appendix A.

4.2 Specifying Step Resources

Since the workflow system has no knowledge of the resources accessed by the programs, additional semantic information is provided by the workflow designer

Table 4.1 Resource Meta-database

Resource ID	**Resource Name**	**Access Constraints**
X	database-name/table-name	—
Y	file-name	ME
Z	machine-name	ME

about the individual steps. The resources used by various steps of the workflows are first registered in a resource meta-database as shown in *Table 4.1*. It stores the resource identifier and access constraints on this resource. The resource identifier could be a file name, machine name or a database table name. Access constraints on a resource specifies whether mutual exclusion is needed for steps accessing that resource.

For each step the resources accessed by the step and the mode of access (wherever applicable) is specified by the workflow designer as shown in *Table 4.2*. The identity of the resource accessed is selected from the resource meta-database. If the resource database does not have an entry in the resource meta-database then it is created. The resource accessed can be further qualified based on the inputs to the step. This has a significant impact on concurrency since conflicts can be considered at a finer granularity. For example, the resource name can be specified as: DatabaseName/TableName/Key or the resource name can be specified as DatabaseName/TableName/Key1-Key2. The former specifies a specific object while the latter specifies a range of objects. The values for the keys are step inputs. Note that the value of the key may not be known statically when the workflow is specified and could be bound to the specific input or output value of a workflow/step.

The resource information provided to the workflow compiler frees the the workflow designer from specifying a conflict for a step with respect to steps from different workflows, *i.e.*, the designer just focuses on the step level details and the rest of the dependencies are taken care of by the workflow compiler and the run-time system.

If key value is not known statically, conflict between steps based on the semantic information may have to be determined dynamically based on the workflow state information. Using the workflow schema information the compiler can also deduce relative ordering, rollback dependency requirements and compensation dependent sets.

4.3 Semantics of Specification – Correctness Requirements

In this section the semantics of the various specifications are described using a simple formal notation for normal execution, failure handling and coordinated execution.

4.3.1 LITES: Notation for Specifying LAWS

We have developed a language called LITES (Language for Implementation TEchniques Specification) to abstract the semantics of the implementation techniques and various workflow correctness requirements. LITES contains symbols for denoting workflow specification, events in history and connectors for connecting the specification and events. This subsection introduces the constructs of LITES and illustrate its use with simple examples. Table 4.3 denotes the notation used for the various events. The set of events include significant events of a workflow. The *step number* within a workflow is denoted by i. Arguments in upper case are used

Table 4.2 Step Level Resource Specification

Step Name	Resource ID	Access Mode
WF1.S1	X/I1	R
WF1.S1	Z	R
WF1.S2	Y	W
WF2.S1	X/I2	R
WF2.S2	Y/I1-I2	W

Table 4.3 Notation Used for Various Events

Event	Notation
Start of Step Execution	$s(i)$
Finish of Step Execution	$f(i)$
Failure of Step Execution	$fa(i)$
Start of Step Compensation	$cs(i)$
Finish of Step Compensation	$cf(i)$
Abort of workflow X	$a(X)$

Table 4.4 Notation Used for Event Connectors

Connector Purpose	Symbol
Implication	$\Rightarrow$
Equivalence	$\equiv$
Happens Before (at run time)	$\rightarrow$
Precedes (in schema definition)	$\prec$
Membership connectors	ϵ , $\forall$, $\ni$
Logical connectors	$\wedge$, $\vee$
Parallel Execution of Steps	$\Vert$
Alternative Execution of Steps	$\sqcup$

to denote workflows and lower case to denote steps. Later, as the need arises the notation will be enhanced.

We also use a set of symbols (*connectors*) to connect events in histories or specify requirements of workflows. Table 4.4 denotes the symbols used for connectors. Note that in an implication, the left hand side denotes a specification in the workflow schema or the occurrence of certain events in history and the right hand side denotes the ordering of appropriate events in history.

Conditions are also used to decide if and when certain events should occur. A condition is denoted by $\mathcal{C}$. Its arguments vary depending on the context in which it is used. For example, if it is associated with only one step (a) as in the case of *re-execution* condition then it is denoted by $\mathcal{C}(a)$. However, if it is used to denote a *conflict* condition between two steps (a of workflow X and b of workflow Y) then

Table 4.5 Notation Used for Composite Events

Event Description	**Original Notation**	**New Notation**
Execution of Step	$s(a) \rightarrow f(a)$	$e(a)$
Compensation of Step	$sc(a) \rightarrow fc(a)$	$c(a)$
Attempt (try) of Step	$s(a) \rightarrow fa(a)$	$t(a)$
Start of i^{th} execution of step		$s_i(a)$
Finish of i^{th} execution of step		$f_i(a)$
Modification of inputs to i^{th} execution		$m_i(a)$
i^{th} *Execution* of Step	$s_i(a) \rightarrow f_i(a)$	$e_i(a)$
i^{th} *Compensation* of Step	$sc_i(a) \rightarrow fc_i(a)$	$c_i(a)$
i^{th} *Attempt (try)* of Step	$s_i(a) \rightarrow fa_i(a)$	$t_i(a)$
i^{th} *Re-execution Condition Evaluation*		$RC_i(a)$

it is denoted by $\mathcal{C}(a,b)$. A condition is evaluated purely based on the value of step inputs/outputs and hence is *stable*, *i.e.*, once a condition has been evaluated its value does not change over time.

Since start and finish of steps happen very commonly in histories, they are replaced with a higher level event that represents step *execution*. Other high level events for compensation and failure of steps are also introduced as shown in *Table 4.5*. Composite events are introduced purely for simplifying history representation. For step related events, another argument i is introduced which indicates that the event is related to the i^{th} attempt of step execution. This is necessary since failures and partial rollbacks can happen multiple times in a workflow and it is necessary to differentiate each of the attempts of step execution.
We represent re-execution condition for i^{th} execution of step a by $\mathcal{RC}_i(a)$.

4.3.2 Normal Execution

The following are some basic properties (semantics of steps) and requirements (orderings that must be satisfied) that should hold for any workflow.

Property 1 *In any history, start and finish events of a step form a total order.*

$$\forall\ a,\ s(a)\ \rightarrow\ f(a)$$

4.3.2.1 Sequential Flow:

Requirement 1*If one step is a successor of another in a workflow specification, (sequentially with zero or more steps in between), the steps execute in the same order.*

$$\forall\ (a,b) \in X,\ a\ \prec\ b\ \Rightarrow\ e(a)\ \rightarrow\ e(b)$$

Requirement 2*If two steps of a workflow are to be executed in an order (sequentially with zero or more step in between), their finish and start events are totally ordered.*

$$\forall\ (a,b),\ a\ \prec\ b\ \Rightarrow\ f(a)\ \rightarrow\ s(b)$$

4.3.2.2 If-Then-Else Branching:

Requirement 3*In the case of an if-then-else branching in a workflow, the first step of the selected branch will be executed sequentially after the last step executed before the branch.*

$$\forall\ (a,b,c),\ a\ \prec\ (b\ \sqcup\ c)\ \Rightarrow\ (f(a)\ \rightarrow\ s(b))\ \vee\ (f(a)\ \rightarrow\ s(c))$$

Alternatively,

$$\forall\ (a,b,c) \in X,\ a\ \prec\ (b\ \sqcup\ c)\ \Rightarrow\ (e(a)\ \rightarrow\ e(b))\ \vee\ (e(a)\ \rightarrow\ e(c))$$

Requirement 4*An 'or' confluence step (where two or more alternative executions of threads join to become one) is executed only after the finish of the last step in the active branch.*

$$\forall\ (a,b,c),\ (a\ \sqcup\ b)\ \prec\ c\ \Rightarrow\ (f(a)\ \rightarrow\ s(c))\ \vee\ (f(b)\ \rightarrow\ s(c))$$

Alternatively,

$$\forall\ (a,b,c) \in X,\ (a\ \sqcup\ b)\ \prec\ c\ \Rightarrow\ (e(a)\ \rightarrow\ e(c))\ \vee\ (e(b)\ \rightarrow\ e(c))$$

4.3.2.3 Parallel Branching:

Requirement 5 *In the case of a parallel branching in a workflow, the first step of each of the parallel branches will be executed sequentially after the last step executed before the branch.*

$$\forall\ (a,b,c),\ a\ \prec\ (b \parallel c)\ \Rightarrow\ (f(a)\ \rightarrow\ s(b))\ \wedge\ (f(a)\ \rightarrow\ s(c))$$

Alternatively,

$$\forall\ (a,b,c) \in X,\ a\ \prec\ (b \parallel c)\ \Rightarrow\ (e(a)\ \rightarrow\ e(b))\ \wedge\ (e(a)\ \rightarrow\ e(c))$$

Requirement 6 *An 'and' confluence step (where two or more parallely executing threads join to become one) is executed only after the finish of the last step in each of its incoming branches.*

$$\forall\ (a,b,c),\ (a \parallel b)\ \prec\ c\ \Rightarrow\ (f(a)\ \rightarrow\ s(c))\ \wedge\ (f(b)\ \rightarrow\ s(c))$$

Alternatively,

$$\forall\ (a,b,c) \in X,\ (a \parallel b)\ \prec\ c\ \Rightarrow\ (e(a)\ \rightarrow\ e(c))\ \wedge\ (e(b)\ \rightarrow\ e(c))$$

Based on the above properties and requirements, the precedence ordering of events for some sample workflow specifications is presented below.

Example 1 $\forall\ (a,b,c,d),\ a\ \prec\ b\ \prec\ c\ \prec\ d\ \Rightarrow$

$$f(a)\ \rightarrow\ s(b)\ \rightarrow\ f(b)\ \rightarrow\ s(c)\ \rightarrow\ f(c)\ \rightarrow\ s(d)$$

Example 2 $\forall\ (a,b,c,d),\ a\ \prec\ (b\ \sqcup\ c)\ \prec\ d\ \Rightarrow$

$$(f(a)\ \rightarrow\ s(b)\ \rightarrow\ f(b)\ \rightarrow\ s(d))\ \vee\ (f(a)\ \rightarrow\ s(c)\ \rightarrow\ f(c)\ \rightarrow\ s(d))$$

Example 3 $\forall\ (a,b,c,d),\ a\ \prec\ (b \parallel c)\ \prec\ d\ \Rightarrow$

$$(f(a)\ \rightarrow\ s(b))\ \wedge\ (f(a)\ \rightarrow\ s(c))\ \wedge\ (f(b)\ \rightarrow\ s(d))\ \wedge\ (f(c)\ \rightarrow\ s(d))$$

Example 4 $\forall\ (a,b,c,d) \in X,\ a\ \prec\ b\ \prec\ c\ \prec\ d\ \Rightarrow\ e(a)\ \rightarrow\ e(b)\ \rightarrow\ e(c)\ \rightarrow\ e(d)$

Example 5 $\forall\ (a,b,c,d)\ \epsilon\ X,\ a\ \prec\ (b\ \sqcup\ c)\ \prec\ d\ \Rightarrow\ (e(a)\ \rightarrow\ e(b)\ \rightarrow\ e(d))\ \vee$
$$(e(a)\ \rightarrow\ e(c)\ \rightarrow\ e(d))$$

Example 6 $\forall\ (a,b,c,d)\ \epsilon\ X,\ a\ \prec\ (b\ \|\ c)\ \prec\ d\ \Rightarrow$
$$(e(a)\ \rightarrow\ e(b))\ \wedge\ (e(a)\ \rightarrow\ e(c))\ \wedge$$
$$(e(b)\ \rightarrow\ e(d))\ \wedge\ (e(c)\ \rightarrow\ e(d))$$

4.3.3 Failure Handling

Requirement 7 *The $(i-1)^{th}$ execution of a step has to be compensated before the i^{th} execution of that step.*

$$\forall\ a\ \epsilon\ X, i > 1, e_i(a)\ \Rightarrow$$
$$e_{i-1}(a)\ \rightarrow\ c_{i-1}(a)\ \rightarrow\ e_i(a)$$

The next few rules discuss the technique of opportunistic compensation and re-execution.

Requirement 8 *Steps of a workflow are to be compensated in reverse order if they form part of a compensation dependent set (CDS).*

$$\forall\ a,b,c\ \epsilon\ X, a\ \prec\ b, a,b\ \epsilon\ CDS1\ \Rightarrow$$
$$c_{i-1}(b)\ \rightarrow\ c_{i-1}(a)\ \rightarrow\ e_i(a)\ \rightarrow\ e_i(b)$$

Requirement 9 *If a different option is taken during re-execution at a if-then-else branching, the steps executed along the other branches are eventually compensated before execution completes if they have not already been compensated*

$$\forall\ a,b,c\ \epsilon\ X, a\ \prec\ ((b\ \prec\ c)\ \sqcup\ d), t_{i-1}(c)\ \rightarrow\ e_i(d)\ \Rightarrow$$
$$e_i(d)\ \rightarrow\ c_{i-1}(c)$$

Requirement 10 *If a step on a execution thread has been already been compensated (because it belongs to a compensation dependent set), then it has to be re-executed if control flow passes through that step*

$$\forall\ a, b \in X, a \prec b, c_{i-1}(a) \rightarrow e_i(b) \Rightarrow$$
$$c_{i-1}(a) \rightarrow e_i(a) \rightarrow e_i(b)$$

Requirement 11 *If a step fails, either the WF is rolled back and restarted from a previously executed step (by evaluating its re-execution condition) or aborted.*

$$\forall\ a_m, a_n \in X, m \leq n, t_i(a_n) \Rightarrow RC_{i+1}(a_m) \vee a(X)$$

Requirement 12 *During the re-execution phase of a workflow, the re-execution conditions are evaluated in the same order as that of the step execution order.*

$$\forall\ a, b \in X, a \prec b \Rightarrow$$
$$\mathcal{R}C_i(a) \rightarrow \mathcal{R}C_i(b)$$

Requirement 13 *If re-execution condition of a step is true, either the step is compensated and re-executed or the step inputs are modified and the step is executed with the new inputs*

$$\forall\ a \in X, \mathcal{R}C_i(a) \Rightarrow$$
$$(c_{i-1}(a) \rightarrow e_i(a)) \vee (m_i(a) \rightarrow e_i(a))$$

4.3.4 Coordinated Execution

4.3.4.1 Mutual Exclusion

Requirement 14 *If a conflict condition $\mathcal{C}$ defined between step a of workflow X and step b of workflow Y is true then the two steps are to be executed in a mutually exclusive manner.*

$$\forall\ a \in X,\ \forall\ b \in Y, \mathcal{C}(a, b) \Rightarrow a\ ME\ b$$

Property 2 *Two steps are executed in a mutually exclusive manner if the finish of a step precedes the start of another.*

$$\forall\ a \in X,\ \forall\ b \in Y,\ a\ ME\ b\ \equiv\ (f(a)\ \rightarrow\ s(b))\ \vee\ (f(b)\ \rightarrow\ s(a))$$

This is true even if a step is rolledback and re-executed. Hence, the following sequence of events will not *occur in history.*

$$f(a)\ \rightarrow\ s(a)\ \rightarrow\ s(b)$$

The following sequence can occur is history and is valid.

$$f(a)\ \rightarrow\ s(b)\ \rightarrow\ f(b)\ \rightarrow\ s(a)$$

4.3.4.2 Relative Ordering

Requirement 15*Steps a and c of workflow X (a happens before c) are to be relatively ordered with respect to steps b and d of workflow Y (b happens before d) only if both the conflict conditions $\mathcal{C}(a,b)$ and $\mathcal{C}(c,d)$ evaluate to true.*

$$\forall\ a,c \in X\ \ni\ a\ \prec\ c,\ \forall\ b,d \in Y\ \ni\ b\ \prec\ d, \mathcal{C}(a,b)\ \wedge\ \mathcal{C}(c,d)\ \Rightarrow$$
$$(a\ RO\ b)\ \wedge\ \ (c\ RO\ d)$$

Property 3*If steps a and c of workflow X (a happens before c) are to be relatively ordered with respect to steps b and d of workflow Y (b happens before d), then the precedence order of finish and start events between steps c and d is the same as that between steps a and b.*

$$\forall\ a,c \in X\ \ni\ a\ \prec\ c,\ \forall\ b,d \in Y\ \ni\ b\ \prec\ d,\ (a\ RO\ b)\ \wedge\ \ (c\ RO\ d)\ \equiv$$
$$((f(a)\ \rightarrow\ s(b))\ \wedge\ (f(c)\ \rightarrow\ s(d)))\ \vee\ ((f(b)\ \rightarrow\ s(a))\ \wedge\ (f(d)\ \rightarrow\ s(c)))$$

4.3.4.3 Rollback Dependency

Requirement 16*If a conflict condition $\mathcal{C}$ defined between step a of workflow X and step b of workflow Y is true and a executes after b then there is a rollback dependency between the two steps*

$$\forall\ a \in X,\ \forall\ b \in Y, \mathcal{C}(a,b)\ \wedge\ (e(b)\ \rightarrow\ e(a))\ \Rightarrow\ a\ RD\ b$$

Property 4*If two steps have a rollback dependency and if the step that executed first is compensated then the other should be compensated as well.*

$$\forall\ a \in X,\ \forall\ b \in Y,\ (a\ RD\ b)\ \Rightarrow\ c(b)\ \rightarrow\ c(a)$$

4.4 Example Specification and Discussion

The schema of a sample workflow is graphically shown in *Figure 4.1.* Current WFMSs allow the specification of such a schema using language primitives similar to the ones described in components 1 through 4 of LAWS. In addition, LAWS permits the specification of failure handling and coordinated execution requirements for a workflow as shown in *Figures 4.2,4.3, and 4.4.*

As discussed in section 3 the goal of this 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

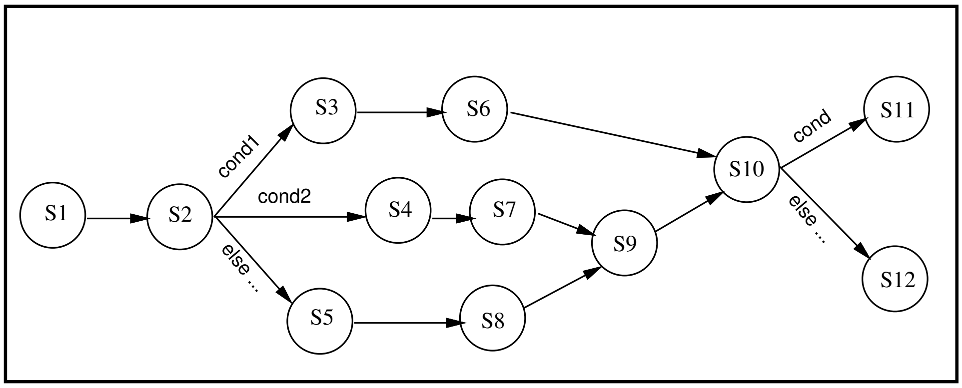

Figure 4.1 Structure of a sample Workflow

```
WF-SCHEMA-NAME ;
WF1 : I1 I2 I3 I4 I5 : O1 O2 ;

STEP-INFO ;
S1 : eternity : sum comp1 : I1 I2 I3 I4 : O1 ;
S2 : era : prod comp2 : I1 I2 19 : O1 ;
S3 : eternity : prod comp3 : I1 I2 : O1 ;
S4 : era : prod comp4 : I1 23 I3 : O1 ;
S5 : eternity : sum comp5 : I1 I2 I3 I4 : O1 ;
S6 : eternity : prod comp6 : I1 I2 : O1 ;
S7 : era : sum comp7 : I1 22 I3 : O1 ;
S8 : eternity : prod comp8 : I1 I2 I3 : O1 ;
S9 : era : sum comp9 : I1 I2 : O1 ;
S10 : eternity : prod comp10 : I1 I2 18 : O1 ;
S11 : eternity : sum comp11 : I1 I2 I3 I4 : O1 ;
S12 : era : prod comp12 : I1 I2 I3 : O1 ;

DATA-FLOW ;
S1.I1 = WF.I1 ;
S1.I2 = WF.I2 ;
S1.I3 = WF.I3 ;
S1.I4 = WF.I4 ;
S2.I1 = S1.O1 ;
S2.I2 = WF.I4 ;
........ /* specification skipped */
S12.I1 = S10.O1 ;
S12.I2 = S7.O1 ;
S12.I3 = S9.O1 ;
WF.O1 = S11.O1 ;
WF.O2 = S12.O1 ;

CONTROL-FLOW ;
S1: S2 ;
S2: IF S2.O1 ¿ 100 THEN S3 IF S2.O1 > 50 OR S1.O1 > 50 THEN S4 S5
                          IF S2.O1 < 25 THEN S6 ;
S3: S7 ;
S4: S8 ;
S5: S9 ;
S6: S10 ;
S7: S11 ;
S8: S11 ;
S9: S11 ;
S10: S11 ;
S11: S12 ;
```

Figure 4.2 Sample Workflow schema specified by workflow designer

the designer to take advantage of the modeling capabilities to specify step resource requirements, failure handling and coordinated execution requirements in a correct manner.

Thus, a practicable approach is provided for enforcing additional correctness properties at the workflow level. The workflow schema 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 correctness, performance and failure handling capabilities of the system.

```
STEP-FAILURE-SPEC ;
S1: RE-START S1 ;
S2: COMPENSATE S1 RE-START S1 ;
S3: COMPENSATE S1 RE-START S1 ;
S4: COMPENSATE S1 ABORT ;
S5: COMPENSATE S2 RE-START S2 ;
S6: COMPENSATE S2 ABORT ;
S7: COMPENSATE S2 RE-START S2 ;
S8: COMPENSATE S2 RE-START S2 ;
S9: COMPENSATE S2 ABORT ;
S10: COMPENSATE S2 RE-START S2 ;
S11: COMPENSATE S10 ABORT ;
S12: COMPENSATE S10 RE-START S10 ;

STEP-RE-EXECUTION-SPEC ;
S1: IF INPUTS-CHANGED COMPENSATE RE_EXECUTE ;
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
S3: IF I2.NEW - I2.OLD > 100 COMPENSATE RE_EXECUTE ;
S4: ALWAYS ;
S5: IF I3.NEW > I3.OLD OR I4.NEW < I4.OLD COMPENSATE RE_EXECUTE ;
S6: IF O1 < S3.O1 COMPENSATE RE_EXECUTE ;
S8: IF (I1.NEW I1.OLD) THEN MODIFY (I1.NEW = I1.NEW - I1.OLD) RE_EXECUTE
     ELSEIF (I2.NEW != I2.OLD) THEN COMPENSATE RE_EXECUTE;
S9: ALWAYS ;
S11: ALWAYS ;
S12: IF I3.NEW != I3.OLD COMPENSATE RE_EXECUTE ;
```

Figure 4.3 Optional Failure Handling Specification

```
COMPENSATION-DEPENDENT-SETS ;
CDS1: S3 S6 ;
CDS2: S4 S7 ;
CDS3: S2 S3 S4 ;

STEP-CONFLICT-SPEC;
IF (S1.I1 == WF4.S2.I2) THEN S1 CONFLICTS_WITH WF4.S2 ;
IF (S2.I1 == WF4.S3.I1) THEN S2 CONFLICTS_WITH WF4.S3 ;
IF (S8.I2 == WF4.S10.I3) THEN S8 CONFLICTS_WITH WF4.S10 ;

COORDINATED-EXECUTION-SPEC ;
IF (S1.I1 CONFLICTS_WITH WF4.S2.I2) THEN S1 MUTUALLY-EXCLUDE WF4.S2 ;
IF (S8.I2 CONFLICTS_WITH WF4.S10.I3) THEN S8 MUTUALLY-EXCLUDE WF4.S10 ;
IF ((S1 CONFLICTS_WITH WF4.S2) AND S1 RELATIVE_ORDER WF4.S2
        AND (S8 CONFLICTS_WITH WF4.S10)) THEN S8 RELATIVE_ORDER WF4.S10 ;
ON WF4.S3.COMPENSATE IF ((S2 CONFLICTS_WITH WF4.S3)
        AND (S1 HAPPENED_AFTER WF4.S3))
        THEN COMPENSATE S2 RE-START S2 ;
```

Figure 4.4 Optional Coordinated Execution Specification

4.5 Completeness and Extensibility of LAWS

Earlier in section 4.3 we discussed the semantics of LAWS specification and how the correctness requirements are satisfied. In this section we compare the additional modeling power that has been added in LAWS with respect to modeling features provided by state-of-the-art WFMSs. This is followed by a discussion on how LAWS can be extended to support new modeling features.

Traditional Features: LAWS supports almost all of the modeling features offered by state-of-the-art WFMSs. These features primarily define the behavior of normal execution of workflows. Very complex workflows can be modeled using features such as conditional branching, looping, and nesting. Given the vast number of domains to which workflows management is applicable, it would be impossible to check if these modeling features are complete enough to capture all the requirements. However, some commercial products suggest that these modeling features are sufficient enough to model workflows in different domains such as insurance companies, banks and healthcare organizations [4]. Hence, we believe that the modeling features provided by LAWS are sufficient enough to model the normal execution of workflows. However, there are other requirements that have to modeled for workflows. There can be logical failures in the execution of workflows and modeling features are needed to correctly and efficiently handle such failures. Also, concurrent access to underlying databases/resources by steps executing from different workflows can cause data inconsistency. To handle such modeling requirements extended features have been added to the language as discussed below.

Extended Features: Our failure handling modeling feature allows the specification of the step to which a workflow has to rollback in case of a logical failure. Also, using the notion of compensation dependent sets, a workflow developer can specify how the steps have to be compensated and re-executed after the rollback. These features combined with the opportunistic compensation and re-execution scheme provide considerable flexibility in the way in which failures are handled and also significantly improve the performance offered by WFMSs. The features provided for coordinated execution offer better correctness properties compared to traditional WFMSs. Better correctness can be achieved by specifying mutual exclusion, relative ordering and rollback dependency requirements. The coordinated execution specifications are handled by generic low level primitives that have been developed for use with the rule-based system.

Extension to LAWS: The use of a rule-based system and generic low level primitives for notifying events across workflows has resulted in a flexible and extensible CREW WFMS irrespective of the underlying workflow control architecture. LAWS can be easily extended if additional modeling features are needed for normal execution, failure handling or coordinated execution. Since the underlying system is based on events and conditions, it is not difficult to add appropriate system support for the additional modeling features.

4.6 Summary

This chapter presented the primitives of LAWS, the workflow specification language of the CREW approach. A workflow schema provided by a designer has a mandatory (traditional) part that describes the structure of the workflow. An optional part then specifies how failures are to be handled and the type of coordinated execution requirements needed. A third part of the schema consists of conflict and dependency specification generated by the data dependency compiler. The use of the primitives of LAWS was illustrated with examples. The semantics of the specification provided in the LAWS language was then specified using LITES, a simple formal notation. The final discussion focused on the completeness of LAWS and the design features that makes it extensible.

CHAPTER 5

Compilation Support — Translating Schema into Rules

This chapter discuss the two stage compiler for transforming workflow schemas written in LAWS into rules. It also presents the details of how the low level primitives are used to implement high level coordinated execution requirements.

5.1 Compilation of Schemas Specified using LAWS

The compiler for LAWS works in two phases. The first phase corresponds to data dependency compilation. This phase is responsible for creating an enhanced workflow schema by taking into consideration the data dependencies between steps within a workflow and across workflows. In the second phase, the compiler generates 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 synchronization. This section discusses the details of the different phases.

Note that the compiler also performs syntax and semantic checking prior to the two phases during which the compiler does the following:

1. for each step specification check if all the required inputs (like nodename, program name, etc.) are specified,

2. for data flow specification, check if the inputs sources to each step are valid

3. for control flow specification, check if the condition and step(s), to which control flows, are valid and ensure that there are no cycles.

4.for failure handling specification check that the steps to be compensated do exist

5.for re-execution specification check if the input variables referred to in the condition are valid and

6.for conflict and synchronization 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.

5.1.1 Data Dependency Compiler

The first phase of the compilation is responsible for creating an enhanced workflow schema by taking into consideration the data dependencies between steps within a workflow and across workflows. It is in this phase that the compiler deduces failure handling and coordinated execution requirements. Using the deduced requirements the user provided workflow schema is transformed into an enhanced workflow schema.

• *Compensation Dependent Sets:*

Compensation dependent sets are deduced based on the resource access dependencies between the steps. Since the objective is to compensate the members of the set in the reverse execution order, only write-write dependencies are important. Consider, step S6 of a workflow that executes after step S2. If both steps access common resource(s) and step S6 accessed the conflicting resource(s) in a write mode, then S2 and S6 form a compensation dependent set. If step S2 is already part of a compensation dependent set, then S6 is added to that set.

• *Compensation and Re-execution Decision:*

Table 5.1 History of object updates by steps

Resource ID	Step Name	Access Mode
X/I1	WF1.S1	R
Z	WF1.S1	R
Y	WF1.S2	W
X/I2	WF2.S1	R
Y/I1-I2	WF2.S2	W

Compensation and re-execution conditions are explicitly specified by the workflow designers based on the inputs to the step. However, workflow designers have the option to specify *automatic* compensation and re-execution based on the updates to the resources accessed by a step. The step has to be re-executed if a resource that it accessed has been updated by any other step since the previous execution. This is achieved at run-time using the resource metadatabase and the step level resource specification. The system maintains a history of the resources accessed by the steps as shown in *Table 5.1*. In this history if the resource(s) accessed by a step have been updated by steps from other workflow instances, then the step is re-executed. Since partial compensation and incremental re-execution decisions cannot be made automatically, only complete compensation and re-execution is allowed.

• *Mutual Exclusion:*

The data dependency compiler deduces mutual exclusion requirements between individual steps and groups of steps. This deduction is based on the resource conflicts that can occur between steps. For every step, the compiler uses the step level resource specification to check if there are steps in other workflows that access common resource(s) in a conflicting mode. The compiler also checks if the resources have a mutual exclusion requirement specified in the global resource database. If there are such conflicting resource(s), then a mutual exclusion requirement is added to the workflow schema. Since the resource specification may be based on step inputs which are known only at run-time, it may not be possible to statically determine all resource conflicts. Thus, a mutual exclusion requirement is added for such steps based on the potential for a conflict even though the existence or non-existence of such a conflict can be determined only at run-time. All other relevant workflow schemas are also updated to reflect conflict from the new workflow schema being compiled.

• *Relative Ordering:*

The data dependency compiler deduces relative ordering requirement if the workflow designer explicitly specifies the need for it in a workflow schema by just including the phrase RELATIVE_ORDER in the COORDINATED-EXECUTION-SPEC without any conditions and statements. For every step, based on the step level resource specification, the compiler checks if there are steps in other workflows that access common resource(s) in a conflicting mode. Also, if there is more than one conflicting pair then a relative ordering requirement is added to the enhanced workflow schema for the steps involved. Also, the workflow designer can add additional relative ordering requirements between steps of workflow even when no conflicts are possible. To simplify the discussion in the rest of the thesis, such steps are still termed as conflicting.

• *Rollback Dependency:*

The data dependency compiler deduces rollback dependency requirements when the workflow designer explicitly specifies the need for it in a workflow schema. Rollback dependency is specified at the step level. For steps that have a rollback dependency requirement specification, the compiler checks if there are steps in other workflows that access common resource(s) in a conflicting mode. If there are such conflicting steps, then a rollback dependency requirement is added to the enhanced workflow schema.

5.1.2 Schema Compiler

In the second phase of the compilation, the enhanced workflow schema is translated into rules that are interpreted by the run-time system.

5.1.2.1 Generating Rules from Mandatory Specification

The next phase of the compilation is more involved. The rules generated in this phase are the conventional Event-Condition-Action rules [32] used in active database systems. All rules generated during this phase 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 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 is generated for step S11 in the sample schema shown in figure 4.1. This rule states that S11 is to be started only after the start of the workflow and the completion of steps S1, S2 and S9. This requirement could have arise from the data flow specification (indicating that S11 receives data from the workflow input and steps S1, S2 and S9).

Next, the existing rules are modified based on the control flow dependencies; More events are added to the rules that start the steps. Conditions specified in the control flow specification (for if-then-else branching) are also added to the rules. Let us consider the case where after step S10 is executed a if-then-else branching is to be performed. Then, the rule above with the control flow specification will essentially be transformed into two new rules as follows

Rule A:

E: S1.D AND WF.S AND S2.D AND S9.D AND S10.D

C: (S10.O1 > 50)

A: S11.S

and

Rule B:

E: S1.D AND WF.S AND S2.D AND S9.D AND S10.D

C: (S10.O1 <= 50)

A: S12.S

After the execution of step S10 both rules will be considered for execution. After the evaluation of the condition, only on of the rules will be triggered and thus either step S11 or S12 will execute.

Rules for loops and nested workflows are generated similarly. In the case of loops, after the execution of the last step in the loop, two rules similar to the ones above (Rules A and B) are generated. The action on one of the rules will start the execution of the next step outside the loop while the action for the other rule will executed the first step of the loop. For nested workflows, in the rule that will trigger the execution of the step the action will indicate the execution of a workflow (for example WF7.S instead of S4.S).

5.1.2.2 Generating Rules from 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. By taking a transitive closure of the compensation dependent sets specified in *Figure 4.4* with respect to step S5 we get the following rule.

E: S5.F

A: S7.C S4.C S6.C S3.C S2.C S2.S

This indicates that on the failure of step S5, steps S7, S4, S6, S3, S2 are to be compensated (only those that executed) in that order and the workflow is to be re-executed from step S2.

5.1.2.3 Generating Rules from Coordinated Execution Specifications

The conflict specification and the coordinated execution specification are processed in an integrated manner. From these specifications 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, rules are added to the run-time RuleSet *dynamically* based on coordinated execution requirements. For example a mutual exclusion statement gets compiled into a rule that looks like

E: S4.D S5.D S7.D

A: S8.S ME WF4.S10

From the event part of the rules it can be seen that when the rule is fired the mutual exclusion action (S8 ME WF4.S10) is triggered. At this point the low level primitives discussed in the next section for realizing the mutual exclusion requirement are enacted, *i.e.*, step S8 is executed while satisfying the mutual exclusion requirement with respect to step S10 of WF4.

The low level primitives used for implementing coordinated execution requirements are discussed in the next section 5.2 and the correctness of these primitives is discussed in section 5.3.

5.1.2.4 Discussion

Most of the rules generated contain composite events, *i.e*, more than one event has to be satisfied for the rule to fire. The events to be satisfied are deduced based on data flow and control flow requirements. In addition to handle coordinated execution requirements the low level primitives are enacted which may require additional events to occur before a step can be executed. Although evaluation of composite events requires a considerable amount of work, they are very useful in capturing complex requirements that have to be satisfied when a step is to be executed or a step fails. Additional description of rule evaluation in the CREW environment is discussed in the next chapter.

5.2 Low Level Primitives for Coordinated Execution

This section begins with a description of the small set of implementation-level primitives that have been developed to realize different coordinated execution requirements expressed in LAWS. Then it is shown that they are sufficient enough to capture the different requirements.

We use two workflows, WF1 and WF2 to illustrate how coordinated execution requirements are satisfied. In what follows,

•*CoordEvent* 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.

•*CoordAction* can be one of the following:

▷*FinishOrBlock* Step — complete the step if it has already started otherwise block it

▷*Block* Step — block the step

▷*Finish* Step — complete the step

Essentially these are special types of actions within a workflow.

The lower level primitives are as follows:

•*AddRule(WorkflowName.InstanceID, CoordEvent, Condition, CoordAction)*
which indicates that a rule consisting of the tuple *(CoordEvent, Condition, CoordAction)* 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 describe how the semantics of higher level building blocks (mutual exclusion, relative ordering and rollback dependency) are realized using lower level primitives. In the discussion that follows, WF1: < *primitive* > indicates that WF1 invokes the enclosed primitive. Also, since only one instance of each of WF1 and WF2 in considered in this discussion, the instanceID has not been specified in the parameter parts of the lower level primitives. In practice, when there are several instances, the instanceID has to be specified to properly manage and deliver messages across concurrent instances. Enactment of the primitives alone is shown to complement the various figures.

5.2.1 Mutual Exclusion

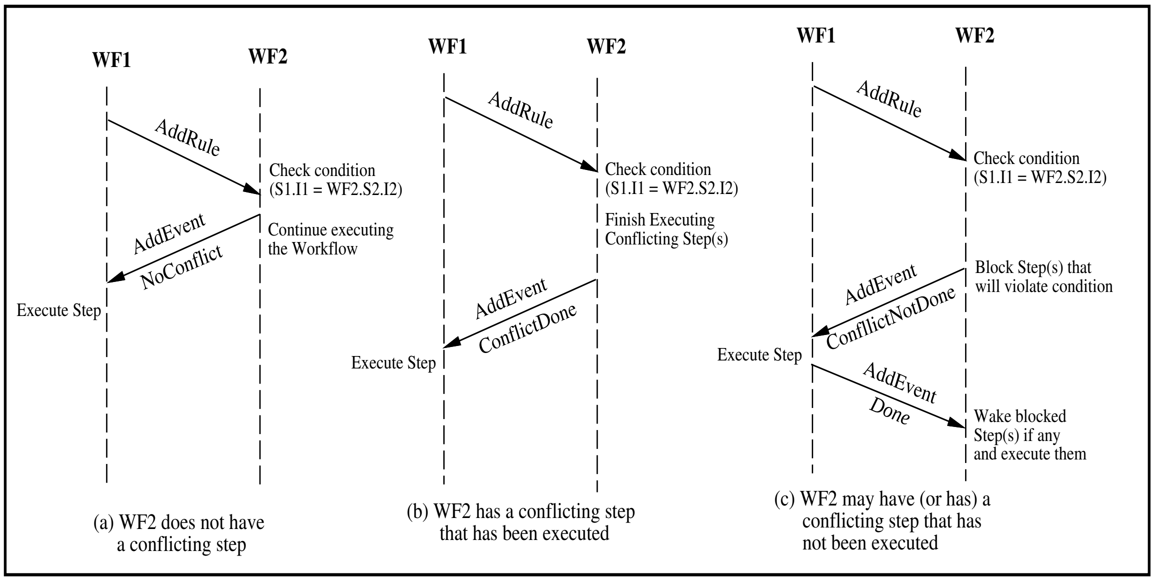

Figure 5.1 Enforcing Mutual Exclusion

An example specification for mutual exclusion in LAWS is

IF S1.I1 = WF2.S2.I2 THEN S1 CONFLICTS_WITH WF2.S2 ;

IF S1.I1 CONFLICTS_WITH WF2.S2.I2 THEN S1 MUTUALLY-EXCLUDE WF2.S2 ;

Using the low level primitives, this statement is implemented as shown in *Figure 5.1.* Three different cases are possible.

Case a:

If the condition (S1.I1 > WF2.S2.I2) evaluates to FALSE the following happens:

WF1: *AddRule(WF2, IMMEDIATE, S1.I1 = WF2.S2.I2, FinishOrBlock S2)*

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

Case b:

If the condition (S1.I1 = WF2.S2.I2) evaluates to TRUE and message was received after S2 has started the following happens:

WF1: *AddRule(WF2, IMMEDIATE, S1.I1 = WF2.S2.I2, FinishOrBlock S2)*

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

Case c:

If the condition (S1.I1 > WF2.S2.I2) evaluates to TRUE and message was received before S2 has started (this happens even if the condition could not be evaluated due to unavailability of data) the following happens:

WF1: *AddRule(WF2, IMMEDIATE, S1.I1 = WF2.S2.I2, FinishOrBlock S2)*

WF2: *AddPrecondition (Self, WF1.S1.Done, S2)*

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

WF1: *AddEvent (WF2, S1.Done)*

5.2.2 Relative Ordering

An example specification for relative ordering in LAWS 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 ;

Using the low level primitives, this statement is implemented as shown in *Figure 5.2*. Two different cases are possible.

We continue from the discussion on mutual exclusion. While handling the action *(FinishOrBlock S2)*, WF2 could have either finished step S2 or blocked it from happening. The event that WF2 sends to WF1 using *AddEvent()* indicates which one of the two happened and this will determine further action in relative ordering.

Case a:

If conflicting step was executed before message is sent from WF2 to WF1, the relative order is *WF2* $\rightarrow$ *WF1* and the following happens:

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

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

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

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

Case b:

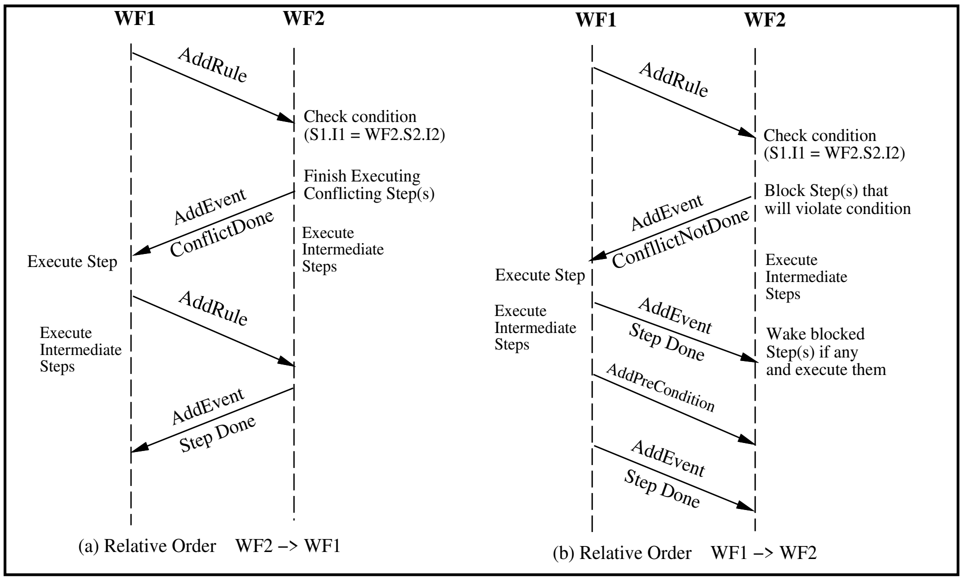

Figure 5.2 Enforcing Relative Ordering

If conflicting step was not executed before message is sent from WF2 to WF1, the relative order is *WF1* $\rightarrow$ *WF2* and the following happens:

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

WF1: *AddEvent (WF2, S1.Done)*

WF1: *AddRule (Self, S8.Done, NULL, Notify WF2 S8.Done)*

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

WF1: *AddEvent (WF2, S8.Done)*

In the above discussion, $Notify$ is a special command that enacts *AddEvent*() to notify the specified event to a given workflow.

5.2.3 Rollback Dependency

An example specification for rollback dependency in LAWS is

IF S2.I1 − WF2.S3.I1 > 100 THEN S2 CONFLICTS_WITH WF2.S3 ;

ON WF2.S3.COMPENSATE

IF S2 CONFLICTS_WITH WF2.S3 AND

S1.START HAPPENED_AFTER WF2.S3.START

THEN COMPENSATE S2 RE-START S2 ;

Using the low level primitives, this statement is implemented as shown in *Figure 5.2*. Three different cases are possible.

Case a:

If the condition evaluates to FALSE the following happens:

WF1: *AddRule (WF2, IMMEDIATE, (S2.I1 − WF2.S3.I1)¿100,*

Notify WF1 S3.Compensate)

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

Case b:

If the condition evaluates to TRUE and S2.compensate does not take place in WF2 the following happens:

WF1: *AddRule (WF2, IMMEDIATE, (S2.I1 − WF2.S3.I1)¿100,* *Notify WF1 S3.Compensate)*

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

WF1: *AddRule (Self, S3.Compensate, NULL, Compensate S2 Re-StartS2)*

Case c:

If the condition evaluates to TRUE and S2.compensate happens in WF2 this will be the sequence:

WF1: *AddRule (WF2, IMMEDIATE, (S2.I1 – WF2.S3.I1)¿100,*

Notify WF1 S3.Compensate)

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

WF1: *AddRule (Self, S3.Compensate, NULL, Compensate S2 Re-Start S2)*

WF2: *AddEvent (WF1, S3.Compensate)*

5.2.4 Summary

All the workflow coordinated execution requirements can be realized using the building blocks that have been developed. To illustrate the use of these building blocks only two workflow instances were considered. In a real workflow environment

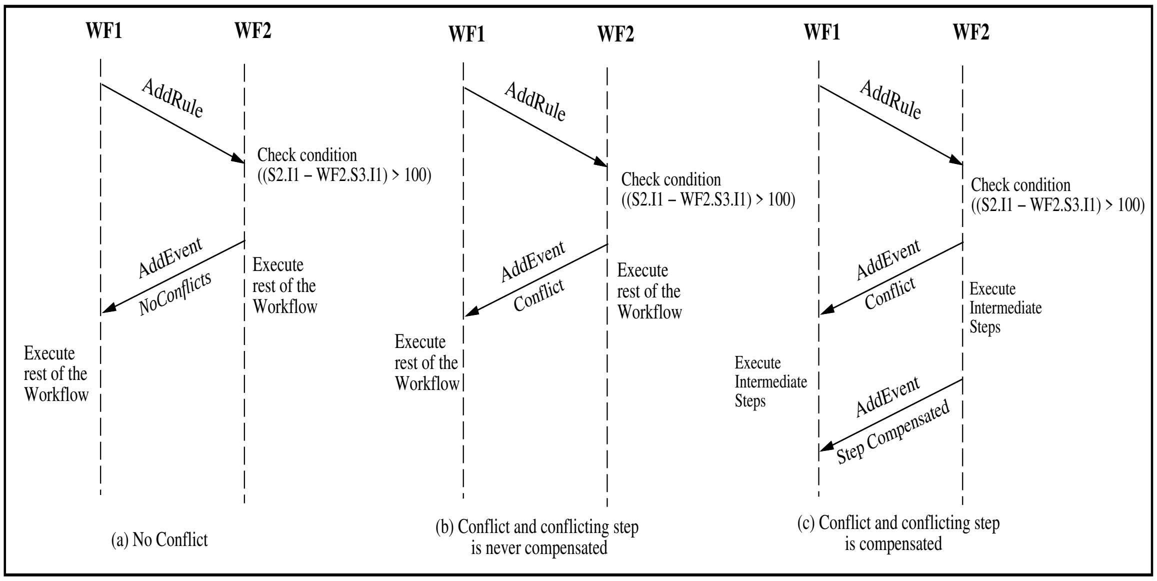

Figure 5.3 Enforcing Rollback Dependency

however, several instances will be running concurrently, with their state information generated/stored at different locations depending on the workflow control environment. Details of the run-time support to realize coordinated execution for different workflow control environments will be discussed in the next chapter.

5.3 Correctness of Mechanisms for Coordinated Execution

This section provides formal proofs to establish the correctness of the mechanisms and protocols used in the CREW low level implementation techniques for coordinated execution.

Table 5.2 lists enhancements to the LITES language introduced in section 4.3 for the various low level primitives. For messages that are sent, arguments X, Y denote the originating and destination workflows respectively. For messages that are received, the arguments are *vice-versa*. Also, $\mathcal{M}$ denotes the contents of a post/receive message events. If the contents are not important for a discussion, $\mathcal{M}$ is dropped, else if the contents are important then the value is represented. Workflows *enact* events like AddRule(), AddEvent() and so forth depending on the coordinated execution specification being handled. In the rest of the section based on the enactment of such events and using rules that govern the low level primitives we show that the properties discussed earlier in section 4.3.1 on LITES for the high level requirements are satisfied.

Table 5.2 Enhancements to LITES

Event	**Notation**
AddRule	$ar(X, Y, \mathcal{M})$
AddEvent	$ae(X, Y, \mathcal{M})$
AddPrecondition	$ap(X, Y, \mathcal{M})$
ReceiveRule	$rr(X, Y, \mathcal{M})$
ReceiveEvent	$re(X, Y, \mathcal{M})$
ReceivePrecondition	$rp(X, Y, \mathcal{M})$

5.3.1 Mutual Exclusion

We now state rules that abstract the semantics of the low level mechanisms that enforce mutual exclusion as shown in figure 5.1. Note that the message that is sent between workflows is represented by symbol $\mathcal{M}$ and it is further qualified only when needed. For example, in the case of AddEvent(), $\mathcal{M}$ can be one of NC or CND or CD to respectively denote *no conflict, conflict not done* and *conflict done.*

Rule 1 *If step a of workflow X has a conflict condition defined on it with respect to step b of workflow Y, step a is executed only after workflow X enacts AddRule() followed by ReceiveEvent().*

$$\forall\ a \in X,\ \forall\ b \in Y,\ a\,\mathcal{C}\,b\ \Rightarrow\ ar(X,Y,\mathcal{M})\ \rightarrow\ re(X,Y,\mathcal{M})\ \rightarrow\ s(a)$$

Rule 2 *After a message is sent by workflow X to workflow Y, it is ultimately received by workflow Y (i.e., the message is delivered reliably and workflow Y enacts the appropriate receive call once the message is received by the node controlling workflow Y).*

$$\forall\ X,Y,\ ar(X,Y,\mathcal{M})\ \rightarrow\ rr(Y,X,\mathcal{M})$$
$$\forall\ X,Y,\ ae(X,Y,\mathcal{M})\ \rightarrow\ re(Y,X,\mathcal{M})$$
$$\forall\ X,Y,\ ap(X,Y,\mathcal{M})\ \rightarrow\ rp(Y,X,\mathcal{M})$$

Rule 3 *If workflow Y enacts a ReceiveRule() on behalf of step b and conflict condition C evaluates to true, workflow Y enacts a AddEvent() only after the finish of step b.*

$$\forall\ a \in X,\ \forall\ b \in Y,\ (s(b)\ \rightarrow\ rr(Y,X,\mathcal{M}))\ \wedge\ \mathcal{C}(a,b)\ \Rightarrow$$
$$rr(Y,X,\mathcal{M})\ \rightarrow\ f(b)\ \rightarrow\ ae(Y,X,CD)$$

Rule 4 *If workflow Y enacts a ReceiveRule() on behalf of step b and if there is insufficient data for conflict condition C to be evaluated (step b is not yet ready for* *execution), workflow Y enacts an AddEvent() immediately. Also, step b is started only after workflow Y enacts a ReceiveEvent() to correspond to the AddEvent() enacted by workflow X on the completion of the conflicting step. This will ensure that if $\mathcal{C}(a,b)$ evaluates to true, the steps will be executed in a mutually exclusive manner.*

$$\forall\ X,\ \forall\ b \in Y, rr(Y,X,\mathcal{M})\ \rightarrow\ s(b)\ \Rightarrow$$
$$rr(Y,X,\mathcal{M})\ \rightarrow\ ae(Y,X,CND)\ \rightarrow\ re(Y,X,\mathcal{M})\ \rightarrow\ s(b)$$

Rule 5 *If conflict condition $\mathcal{C}$ evaluates to true, after completion of step a of workflow X, an AddEvent() is enacted by workflow X.*

$$\forall\ a \in X,\ \forall\ Y,\ \mathcal{C}(a,b)\ \Rightarrow\ f(a)\ \rightarrow\ ae(X,Y,\mathcal{M})$$

Now it will be shows that the abstraction of the implementation correctly handles the requirement of the cases shown in *Figure 5.1*. Only two cases are considered (b) and (c) shown in the figure since in case(a) the conflict condition evaluates to false and mutual exclusion is not required. Let workflow X correspond to WF1 and workflow Y correspond to WF2 and let steps a and b be the respective steps to be executed mutually exclusively.

Example 7 *Case b: Step b from Y has started and Y enacts a ReceiveRule() after that*

$(s(b)\ \rightarrow\ rr(Y,X,\mathcal{M}))\ \wedge\ \mathcal{C}(a,b).$

$\Rightarrow\ (rr(Y,X,\mathcal{M})\ \rightarrow\ f(b)\ \rightarrow\ ae(Y,X,CD))\ \wedge\ \mathcal{C}(a,b)$ *... by rule 3*

$\Rightarrow\ (ar(X,Y,\mathcal{M})\ \rightarrow\ rr(Y,X,\mathcal{M})\ \rightarrow\ f(b)\ \rightarrow\ ae(Y,X,\mathcal{M})\ \rightarrow\ re(X,Y,\mathcal{M}))$
$\wedge\ \mathcal{C}(a,b)$ *... by rule 2*

$\Rightarrow\ ar(X,Y,\mathcal{M})\ \rightarrow\ rr(Y,X,\mathcal{M})\ \rightarrow\ f(b)\ \rightarrow\ ae(Y,X,\mathcal{M})\ \rightarrow$
$re(X,Y,\mathcal{M})\ \rightarrow\ s(a)$*... by rule 1*

$\Rightarrow\ f(b)\ \rightarrow\ s(a)$ *... by simplification*

$\Rightarrow\ a\ ME\ b$ *... by property 2 in section 4.3.1*

We also observe that when workflow Y enacts $re(Y,X,CD)$ then the precedence order $f(b) \rightarrow s(a)$ holds.

Example 8 *Case c: Y enacts a ReceiveRule() before Step b from Y has started*

$(rr(Y,X,\mathcal{M}) \rightarrow s(b)) \wedge \mathcal{C}(a,b)$

$\Rightarrow (rr(Y,X,\mathcal{M}) \rightarrow ae(Y,X,CND) \rightarrow re(Y,X,\mathcal{M}) \rightarrow s(b))$

$\wedge\, \mathcal{C}(a,b)$ ... *by rule 4*

$\Rightarrow (ar(X,Y,\mathcal{M}) \rightarrow rr(Y,X,\mathcal{M}) \rightarrow re(Y,X,CND) \rightarrow re(X,Y,CND)$

$\rightarrow ae(Y,X,\mathcal{M}) \rightarrow re(Y,X,\mathcal{M}) \rightarrow s(b)) \wedge \mathcal{C}(a,b)$ ... *by rule 2*

$\Rightarrow ar(X,Y,\mathcal{M}) \rightarrow rr(Y,X,\mathcal{M}) \rightarrow re(Y,X,CND) \rightarrow re(X,Y,CND) \rightarrow f(a)$

$\rightarrow ae(Y,X,\mathcal{M}) \rightarrow re(Y,X,\mathcal{M}) \rightarrow s(b)$ ... *by rule 5*

$\Rightarrow f(a) \rightarrow s(b)$ ... *by simplification*

$\Rightarrow a\ ME\ b$ ... *by property 2 in section 4.3.1*

We also observe that when workflow Y enacts $re(Y,X,CND)$ then the precedence order $f(a) \rightarrow s(b)$ holds.

Generalization: It was formally shown that a step is executed in a mutually exclusive manner by considering two cases — one in which the conflicting step has executed (or is executing) and the other in which a conflicting step is prevented from starting until the step under consideration is executed. Irrespective of the number of workflows executing concurrently, conflicting steps from such workflows will fall in one of the two categories. Thus the low level primitives will always ensure that a step which conflicts with another will always be executed in a mutually exclusive manner.

5.3.2 Relative Ordering

We now state rules that abstract the semantics of the low level mechanisms that achieve relative ordering as shown in figure 5.2. Note that relative ordering is a specialization of mutual exclusion since multiple pairs of steps are to be executed in a mutually exclusive manner along with an additional requirement that they also have the same execution sequence. Hence, these rules work in conjunction with rules we described in the previous section on mutual exclusion.

Rule 6 *When a workflow X enacts ReceiveEvent() before the execution of a step belonging to the first pair of steps to be relatively ordered, and the message indicates that the conflicting step has been done ($\mathcal{M} = CD$) then workflow X executes the step and then enacts AddRule() followed by ReceiveEvent() before the execution of the step belonging to the next pair of steps to be relatively ordered.*

$$\forall\ a, c \in X\ \ni\ a\ \prec\ c,\ \forall\ Y,\ re(X, Y, CD)\ \Rightarrow$$
$$re(X, Y, CD)\ \rightarrow\ s(a)\ \rightarrow\ ar(X, Y, \mathcal{M})\ \rightarrow\ re(X, Y, \mathcal{M})\ \rightarrow\ s(c)$$

Rule 7 *When a workflow X enacts ReceiveEvent() before the execution of a step belonging to the first pair of steps to be relatively ordered, and the message indicates that the conflicting step has not been done ($\mathcal{M} = CND$) then workflow X executes the step and then enacts AddPrecondition() followed by the execution of the step belonging to the next pair of steps to be relatively ordered followed by enactment of AddEvent().*

$$\forall\ a, c \in X\ \ni\ a\ \prec\ c,\ \forall\ Y,\ re(X, Y, CND)\ \Rightarrow$$
$$re(X, Y, CND)\ \rightarrow\ s(a)\ \rightarrow\ ap(X, Y, \mathcal{M})\ \rightarrow\ f(c)\ \rightarrow\ ae(X, Y, \mathcal{M})$$

Rule 8 *If workflow Y had enacted AddEvent() with the message that conflicting step b has been done ($\mathcal{M} = CD$), and conflict condition $\mathcal{C}(c, d)$ evaluates to true and if workflow Y enacts a ReceiveRule() on behalf of step d, workflow Y enacts an AddEvent() only after the finish of step d.*

$$\forall\ c \in X,\ \forall\ d \in Y,\ ae(Y, X, CD)\ \wedge\ \mathcal{C}(c, d)\ \Rightarrow$$
$$ae(Y, X, CD)\ \rightarrow\ rr(Y, X, \mathcal{M})\ \rightarrow\ f(d)\ \rightarrow\ ae(Y, X, \mathcal{M})$$

Rule 9*If workflow Y had enacted AddEvent() with the message that conflicting step b has not been done ($\mathcal{M} = CND$), and conflict condition $\mathcal{C}(c, d)$ evaluates to true and if workflow Y enacts a ReceivePrecondition() on behalf of step d, workflow Y executes step d only after enacting ReceiveEvent().*

$$\forall\ c \in X,\ \forall\ d \in Y,\ ae(Y, X, CND)\ \wedge\ \mathcal{C}(c, d)\ \Rightarrow$$
$$ae(Y, X, CND)\ \rightarrow\ rp(Y, X, \mathcal{M})\ \rightarrow\ re(Y, X, \mathcal{M})\ \rightarrow\ s(d)$$

Now it will be shown that the abstraction of the implementation correctly handles the requirement of the cases shown in *Figure 5.2.* Let workflow X correspond to WF1 and workflow Y correspond to WF2 and let steps a, b and c, d be the pairs of steps from the respective workflows to be relatively ordered. Note that we consider only condition $\mathcal{C}(c, d)$ since the set of events discussed here happen only if $\mathcal{C}(a, b)$ has been evaluated to true. Note that by proofs of cases b & c in the previous section, when $\mathcal{C}(a, b)$ evaluates to true, $ae(Y, X, CD)$ and $ae(Y, X, CND)$ correspond to precedence orders $f(b)\ \rightarrow\ s(a)$ and $f(a)\ \rightarrow\ s(b)$ respectively. Only cases where $\mathcal{C}(c, d)$ evaluates to true, *i.e.*, cases (b) and (c) from *Figure 5.1* are considered.

Example 9*Case a: subsumes case b from* Figure 5.1 *and Example 7*

$(f(b)\ \rightarrow\ s(a))\ \wedge\ ae(Y, X, CD)\ \wedge\ \mathcal{C}(c, d)$

$\Rightarrow\ (f(b)\ \rightarrow\ s(a))\ \wedge\ (ae(Y, X, CD)\ \rightarrow\ rr(Y, X, \mathcal{M})\ \rightarrow\ f(d)\ \rightarrow\ ae(Y, X, \mathcal{M})))$

... *by rule 8*

$\Rightarrow\ (f(b)\ \rightarrow\ s(a))\ \wedge\ (ae(Y, X, CD)\ \rightarrow\ re(X, Y, CD)\ \rightarrow\ ar(X, Y, \mathcal{M})$

$\rightarrow\ rr(Y, X, \mathcal{M})\ \rightarrow\ f(d)\ \rightarrow\ ae(Y, X, \mathcal{M}))\ \rightarrow\ re(X, Y, \mathcal{M}))$ ... *by rule 2*

$\Rightarrow\ (f(b)\ \rightarrow\ s(a))\ \wedge\ (ae(Y, X, CD)\ \rightarrow\ re(X, Y, CD)\ \rightarrow\ ar(X, Y, \mathcal{M})$

$\rightarrow\ rr(Y, X, \mathcal{M})\ \rightarrow\ f(d)\ \rightarrow\ ae(Y, X, \mathcal{M}))\ \rightarrow\ re(X, Y, \mathcal{M})\ \rightarrow\ s(c))$

... *by rule 6*

$\Rightarrow\ ((f(b)\ \rightarrow\ s(a))\ \wedge\ (f(d)\ \rightarrow\ s(c)))$ ... *by simplification*

$\Rightarrow\ (a\ RO\ b)\ \wedge\ \ (c\ RO\ d)$ ... *by property 3 in section 4.3.1*

Example 10 *Case b: subsumes case c from* Figure 5.1 *and Example 8*

$(f(a) \rightarrow s(b)) \wedge ae(Y, X, CND) \wedge \mathcal{C}(c, d)$

$\Rightarrow (f(a) \rightarrow s(b)) \wedge (ae(Y, X, CND) \rightarrow rp(Y, X, \mathcal{M}) \rightarrow re(Y, X, \mathcal{M}) \rightarrow s(d)))$

... by rule 9

$\Rightarrow (f(a) \rightarrow s(b)) \wedge (ae(Y, X, CND) \rightarrow re(X, Y, CND) \rightarrow ap(X, Y, \mathcal{M})$

$\rightarrow rp(Y, X, \mathcal{M}) \rightarrow ae(Y, X, \mathcal{M})) \rightarrow re(X, Y, \mathcal{M}) \rightarrow s(d))$ *... by rule 2*

$\Rightarrow (f(a) \rightarrow s(b)) \wedge (ae(Y, X, CND) \rightarrow re(X, Y, CND) \rightarrow ap(X, Y, \mathcal{M})$

$\rightarrow rp(Y, X, \mathcal{M}) \rightarrow f(c) \rightarrow ae(Y, X, \mathcal{M})) \rightarrow re(X, Y, \mathcal{M}) \rightarrow s(d))$

... by rule 7

$\Rightarrow ((f(a) \rightarrow s(b)) \wedge (f(c) \rightarrow s(d)))$ *... by simplification*

$\Rightarrow (a\ RO\ b) \wedge (c\ RO\ d)$ *... by property 3 in section 4.3.1*

Generalization: It has been formally shown that steps from a workflow X can be relatively ordered with steps from another workflow Y by considering two different orders — one in which X is ordered before Y and another in which Y is ordered before X. Irrespective of the number of workflows executing concurrently, any two workflows that have a relative ordering requirement will fall in one of the two categories. Thus the low level primitives will always ensure that relative ordering requirements between steps of concurrent workflows will be handled correctly.

5.3.3 Rollback Dependency

We now state rules that abstract the semantics of the low level mechanisms that enforce rollback dependency as shown in figure 5.3. Note that rollback dependency is a specialization of mutual exclusion since steps have to be first executed in a mutually exclusive manner to determine rollback dependency. Hence, these rules work in conjunction with rules described in the earlier section on mutual exclusion. In the rules, the lower case alphabets are used to denote steps and upper case alphabets are used to represent workflows.

Rule 10*If step a of workflow X has a rollback dependency with respect to step b of workflow Y, workflow X enacts AddRule() on self after the completion of step a.*

$$\forall\ a \in X,\ \forall\ b \in Y, \mathcal{C}(a,b)\ \Rightarrow\ e(a)\ \rightarrow\ ar(X,X,\mathcal{M})$$

Rule 11*If step b in workflow Y induced a rollback dependency on step a of workflow X and step b is compensated then workflow Y enacts an AddEvent().*

$$\forall\ a \in X,\ \forall\ b \in Y,\ \mathcal{C}(a,b)\ \wedge\ c(b)\ \Rightarrow\ c(b)\ \rightarrow\ ae(Y,X,\mathcal{M})$$

Rule 12*If step a in workflow X has a rollback dependency with respect to step b in workflow Y and it enacted an AddRule() on self, if it subsequently enacts ReceiveEvent() to get notification of the compensation of step b then step a is compensated.*

$$\forall\ a \in X,\ \forall\ b \in Y,\ \mathcal{C}(a,b)\ \wedge\ (ar(X,X,\mathcal{M})\ \rightarrow\ re(X,Y,\mathcal{M}))\ \Rightarrow$$
$$ar(X,X,\mathcal{M})\ \rightarrow\ re(Y,X,\mathcal{M})\ \rightarrow\ c(a)$$

Now it will be shown that the abstraction of the implementation correctly handles the requirement of the case (b) shown in *Figure 5.3*. Let workflow X correspond to WF1 and workflow Y correspond to WF2 and let steps a and b be the respective steps that have a rollback dependency between them.

Example 11*Case b:*

$\mathcal{C}(a,b)\ \wedge\ c(b)$

$\Rightarrow\ (e(a)\ \rightarrow\ ar(X,X,\mathcal{M}))\ \wedge\ c(b)$ ... *by rule 10*

$\Rightarrow\ e(a)\ \rightarrow\ ar(X,X,\mathcal{M})\ \rightarrow\ c(b)\ \rightarrow\ ae(Y,X,\mathcal{M})$ ... *by rule 11*

$\Rightarrow\ e(a)\ \rightarrow\ ar(X,X,\mathcal{M})\ \rightarrow\ c(b)\ \rightarrow\ ae(Y,X,\mathcal{M})\ \rightarrow\ re(X,Y,\mathcal{M})$ ... *by rule 2*

$\Rightarrow\ e(a)\ \rightarrow\ ar(X,X,\mathcal{M})\ \rightarrow\ c(b)\ \rightarrow\ ae(Y,X,\mathcal{M})\ \rightarrow\ re(X,Y,\mathcal{M})\ \rightarrow\ c(a)$
... *by rule 12*

$\Rightarrow\ c(b)\ \rightarrow\ c(a)$ ... *by simplification*

$\Rightarrow\ a\ RD\ b$ ... *by property 4 in section 4.3.1*

Generalization: It has been formally shown that if a step a from workflow X has a rollback dependency with steps b from another workflow Y and if b is compensated, a will be subsequently compensated as well. Irrespective of the number of workflow instances executing concurrently, once a rollback dependency is established the dependent step is correctly compensated.

5.3.4 Completeness of Low Level Primitives

From the various proofs in this section, it can be observed that the low level primitives are well suited for notifying events in the rule-based framework. The two primitives *AddEvent()* and *AddPrecondition()* are very useful in *dynamically modifying existing rules* in a rule-base associated with an engine/agent. They provide the ability to add new constraints in the form of additional events and conditions in a rule. The only other primitive that has been defined, *AddRule()* provides the ability to *dynamically add new rules* to a rule-base associated with an engine/agent. This is very useful in triggering additional actions based on the occurrence of local or remote events (at a different engine/agent). Based on the occurrence of events specified in a rule and the associated condition evaluating to TRUE, there may be a need to trigger additional actions thus justifying the need for an *AddAction()* primitive. Our analysis has not shown the need for such a primitive. Even if such a primitive was needed, it has not been defined since the *AddRule()* primitive is capable of handling the functionality needed from *AddAction()*. Thus, the rule-based framework along with the low level primitives is capable of handling any high level requirement specified at the workflow schema level (using LAWS).

5.4 Summary

This chapter discussed the functioning of the two stage compiler that ultimately translates a given workflow specification into a set of rules. These are consolidated rules generated by taking into considerations the three different parts of the schema.

A rule may be fired based on the occurrence of a single event or multiple events provided the required condition (if any) is satisfied. The rules are then interpreted by the run-time system described in the next chapter to execute the workflows according to the specification in different workflow control architectures. The low level mechanisms used for realizing the coordinated execution requirements were also discussed, including proofs for their correctness based on the LITES formal language. Finally, the completeness of the low level primitives was discussed for coordinating execution of steps across workflows.

CHAPTER 6

System Support for Various Workflow Architectures

This chapter describes the mechanisms that handle the execution of the workflow instances, including normal navigation, failure-handling and coordinated execution. The mechanisms are discussed for the different workflow control architectures. Finally we discuss performance and implementation issues.

6.1 Execution Support in Centralized Workflow Control

This section discusses normal execution support for workflows as well as support for handling user requests, handling failures and enforcing coordinated execution in a centralized workflow control architecture. The relevant APIs are discussed and the run-time mechanisms that support their functionality. Wherever possible optimizations are suggested to reduce resource consumptions or wastage of work.

6.1.1 Normal Execution

The API for starting a workflow is as follows:

WorkflowStart(WorkflowName, InstanceNum, Arguments)

Workflows are usually instantiated through an application interface. All customer interactions are handled by the front end database that is external to a WFMS. The front end database forms an interface between a customer and the back end which interacts with an organizations information system and is usually managed by the WFMS administrator. It stores all the mapping information between customer interactions and workflows executions. For example, when an order is placed for a product, the front end database accepts the order and issues a customer order number to the customer. Then the appropriate workflow is instantiated at the back end using the central engine's *WorkflowStart()* interface to process the customer order. The customer is unaware of the workflow name, schema and even the instance number. The mapping between the order number and the workflow instance number is stored in the front end database. When a customer requests some action (change or cancelation) on an existing order, using the mapping information the requests are performed on the appropriate workflow instance. Issues specific to user initiated workflow actions are discussed later.

When a workflow is instantiated a data structure called the workflow instance table is created in the central engine to store the workflow state and context information. If the workflow instance is the first instance of a particular workflow class then another data structure called workflow class table is created in the central engine. These data structures are shown in figure 6.1.

A *workflow class table* is created only once for each workflow class (workflow schema definition). It contains a number of sub-structures. A *step table* stores step related information. While some steps can be executed at only one agent others can be executed at one of many eligible agents. It stores the name(s) of agent(s) at which a step can be executed and the name of the program needed for executing the step and compensating the step. The table also has information about the type of the step, *e.g.*, whether the step performs updates on data resources or just queries. A *data mapping table* stores information about how the data items are mapped to one another. This table provides information about where a step gets its input from. The final sub-structure is an instance summary table that contains essential information about all the instances of that workflow class. It indicates the status of the instance, *i.e.*, whether the instance is executing or committed and maintains a

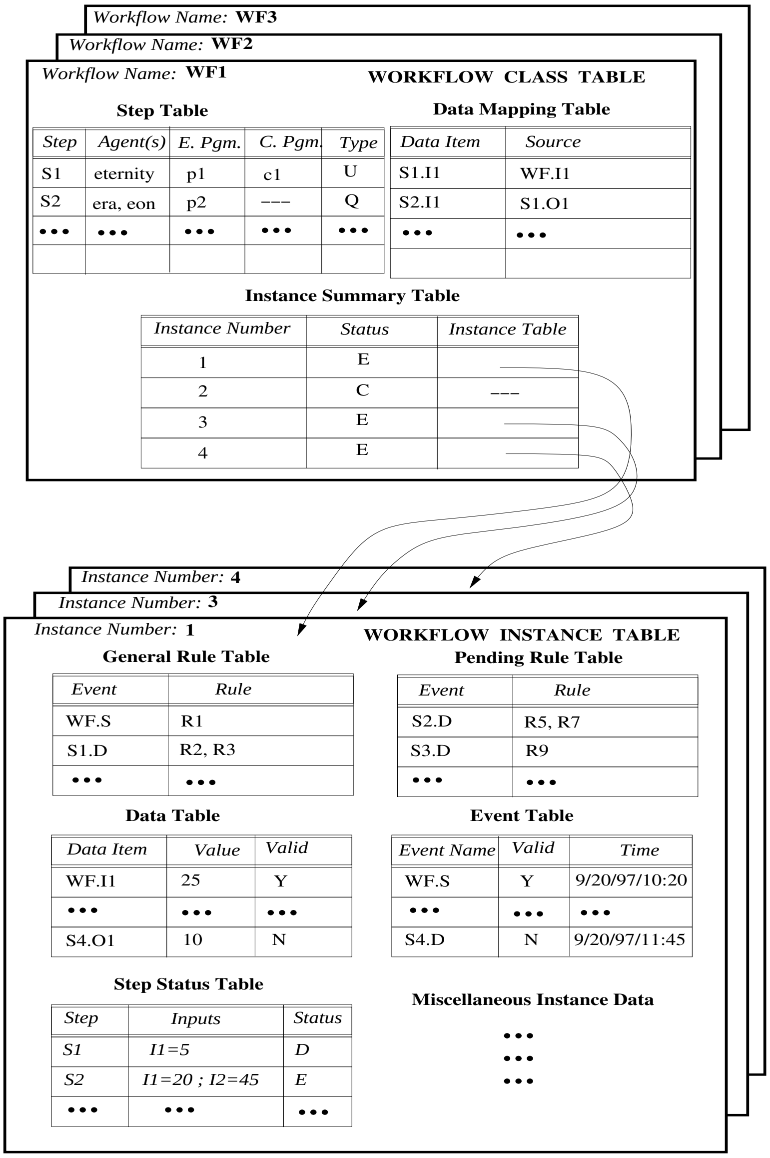

Figure 6.1 Data Structures for Managing Workflow State Information

pointer to the actual workflow instance table as long as the instance is in execution. Thus the workflow class table stores information that is common to all the instances of that class and pointers to information about the individual instances. Thus the total size of the workflow class table is directly dependent on the size of the workflow (number of steps) and the number of instances currently in execution.

A *workflow instance table* is created for each instance of a workflow. It contains many sub-structures. A *general-rule table* holds rules produced by the compilation of the workflow schema. This table is stored on a per instance basis since the rules are modified at run-time for performing coordinated execution. A *data table* is used for storing information about the values of the data items produced (step outputs) as a result of the execution of the steps. Note that if a step has been executed more than once (as part of workflow rollback and re-execution), previously generated step outputs are also stored in this table. The data table is accessed to perform the actual data mapping between outputs and inputs of steps. The mapping itself is done in conjunction with the data mapping table. The size of the general rule table is also dependent on the number of steps in the workflow. An *event table* stores all the events, their version numbers (explained later in the section) and the times at which they were generated. The event table is accessed to inquire about the occurrence of specific workflow events (to assist in rule firing). The size of the event table depends on the progress that has been made in the workflow, *i.e.*, on the number of steps completed. A *pending-rule table* is used to hold the rules that are waiting for some additional events to occur before they can be fired. Rules are first selected for evaluation from the general rule table when the first of many events required to fire the rule occurs. A copy of the rule is placed in the pending-rule table after modifying the event part to exclude the event that triggered the evaluation of the rule. As an event occurs, rules from the pending table are checked and the event is deleted from the rules that require them. After the deletion, if the list of events needed to fire a rule is empty then the rule is fired, else the rule continues to remain in the pending rule table. As rules are fired, steps are executed, and events are generated the above tables are accessed and updated. A *step status table* stores the status information of the steps and its inputs. This table is used primarily while enforcing coordinated execution. Also, this table stores the previous inputs to the steps if the step has been executed more than once as part of workflow rollback and re-execution. As mentioned earlier, previous inputs to the steps are needed to evaluate compensation

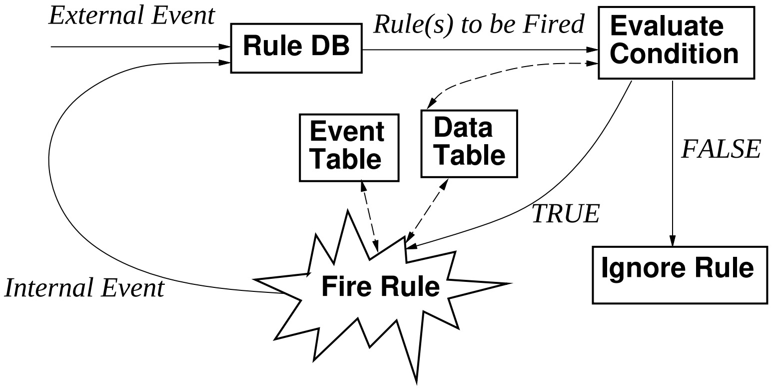

Figure 6.2 Rule firing in the Rule Based Engine

and re-execution conditions. The size of the step status table is dependent on the number of steps that have been completed in the workflow. The tables are stored in the workflow database and the changes are immediately committed in the database to support forward recovery.

The main loop in the engine waits for events in the event queue and dispatches them as soon as they occur. As an event is generated, both new and pending rules are considered for firing. New rules are those that have not been fired and are considered from the general-rule table. If a new rule requires additional events before it can be fired then it is added to the pending-rule table. Rules in the pending-rule table which have been waiting for the event that just occurred are also fired. Note that for ECA rules to be fired, in addition to the occurrence of required events their condition should also evaluate to true. This functioning of the rule evaluation and firing mechanism is shown in figure 6.2.

On the instantiation of a workflow, a $workflow.start$ event is generated which triggers 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. Rules that need other events are added to the pending-rule table. The workflow engine is responsible for coordinating and scheduling the steps. The steps themselves are executed by the agents.

The API for executing and compensating a step at an agent are as follows:

StepExecute(WorkflowName, InstanceNum, StepNum, Program, Arguments)
StepCompensate(WorkflowName, InstanceNum, StepNum, Program, Arguments)

To execute a step by invoking an agent's *StepExecute()* interface the engine sends the required information to the agent responsible for executing the step. Other than the step identifying information (workflow name, instance number, step number), the information sent includes the name of the program to be executed and the input data items to the step. Based on the information received from the engine, the agent executes the step by invoking the required program with the supplied arguments. It logs the results produced and returns the results to the engine. The engine then examines the results returned by the agent to check if the step was successfully executed. It the step was successful then the engine updates the data table to store the output produced by the step. After the successful completion of the step a *step.done* event is generated. The event table is updated and the engine triggers more rules based on the event. If the step was not successful then a *step.fail* event is generated. For compensating a step the engine invokes the *StepCompensate()* interface at an agent in a similar manner.

The central engine is responsible for handling the navigation of all workflow instances. Navigating through an instance involves handling the various control structures *i.e.*, sequential flow, parallel branching, if-then-else branching and branch-joins (where several branches meet at a confluence step) as specified in the workflow schema. These control structures are translated into the appropriate rules after schema compilation. A rule that fires a step in a sequential path will require only the occurrence of one event (corresponding to *step.done* of the previous step) to be fired. However, the rule may require other *step.done* events depending on which of the steps it gets its input data from. After all the required events have occurred the rule is fired resulting in the execution of the step. All these mechanisms satisfy *requirements 1 and 2* stated in section 4.3. In the case of an if-then-else branching, after a *step.done* event is generated for the branching step, two rules are eligible for firing. The two rules only differ in their condition parts. The conditions exactly correspond to the condition associated with the if-then-else branch specification in the workflow schema. After the evaluation of the condition only one of the two rules is fired. The rules that trigger the rest of the steps in a branch operate in the same way as in a sequential path. None of the rules that trigger the execution of steps in the other (non-selected) branch are fired. All these mechanisms satisfy *requirements 3, 4, 5, and 6* stated in section 4.3. In the case of parallel branching, after the execution of the branching step multiple rules are fired. These rules do not have conditions associated with them. Each rule will trigger the first step of each of the parallel branches. Finally, consider branch-joins that occur at confluence steps. The execution of a step is triggered when control flow along all the incoming branches reaches the step. The rule that triggers the execution of a confluence step will be fired on the occurrence of *step.done* events for the last steps of all incoming branches. All these mechanisms satisfy *requirements 7, 8, 9, and 10* stated in section 4.3. Since the entire state information (data and event table) is available at a single node it is easy to manage rule execution and handle the various control structures.

Another important aspect of scheduling has to do with multiple agents being eligible to execute a step as specified in the workflow schema (step table). In such a situation, the central engine has to decide which agent should execute the step. The engine uses a two phase process to make this decision. The engine first polls the eligible agents about their states. The agents return this information and the engine picks the agent with the minimum load to execute a step. Alternatively, the agent where a step is executed could be decided based on which staff member among a group decides to execute the step. The agent that is running on the same node on which the staff member is working may be selected as the agent for executing the step. The procedure for selecting an agent for executing a step in a centralized architecture remains the same irrespective of whether the step is a confluence step or a step in a sequential path of a workflow.

A workflow is committed when the last step along each active execution path have executed successfully. This implies that the last step along a sequential thread, along an if-then-else branch and along all parallel branches must be committed before a workflow is considered to have committed. A $workflow.done$ event is generated when all the steps complete. The central engine is responsible for committing the workflow. Committing a workflow corresponds to making its changes permanent. This implies that the *workflow will not be rolled back* in the future, *i.e.*, the effect of the individually committed steps of the workflow will not be compensated. This is true for nested workflows as well. After the execution of the nested workflow, the effects of the workflow can be compensated only by executing another compensating workflow.

While a committed workflow is never rolled back a committed step can be rolled back depending on other events in the workflow. After a workflow commits, any future request for aborting the workflow or changing the workflow inputs will be rejected. After a workflow is committed, the instance table information is archived and the status of the instance is marked as committed in the instance summary table. Since all the state information of a workflow is maintained at the engine, the engine does not require additional communication for committing a workflow.

6.1.2 Enforcing Coordinated Execution

In a centralized environment, the global state information, *i.e.*, state information of all the workflow instances is available on a single node. Thus enforcing coordinated execution requires message exchanges between instances all controlled by a single node. Strictly speaking, an instance is not active entity but state information. However, the notion of message passing between instances is used for convenience.

Note that the messages exchanged between instances are logical and not physical messages on the network.

When coordinated execution is required between workflows, messages are to be exchanged between the workflow instances to handle the low level primitives. 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. To facilitate communication between workflow instances in the central engine a *local communication layer* has been added as shown in *Figure 6.3*. This layer implements the engine APIs for *AddRule(), AddEvent(), and AddPrecondition()* primitives. An instance can make calls to this API by supplying the required arguments. The AddRule() call adds the rule to the specified workflows and requests the threads handling the workflow instances to fire the rule immediately if necessary. The AddEvent() call adds the event in the event queues of the specified workflows. Similarly, the AddPrecondition() call adds the precondition to the specified rules in the specified workflows. The engine deals with the individual instances of the specified workflows to process the call. Thus a workflow instance making the call will initially have no knowledge about the instances of other workflows that are in execution. Later when the call returns, the instances will learn about conflicting steps from other instances of the same and different workflow classes. If a *AddRule (WF2, ...)* API is invoked by an instance IN#3 of WF1, the rule is added to all instances of WF2. A message is sent to the local communication layer which routes the message to *all active* instances of WF2. To properly keep track of the *sending* workflow instance and the *receiving* workflow instance(s), the workflow schema name and instance ID is prepended to the instance data by the local communication layer. Hence for the above message the *sender* will be referred to as *WF#1.IN#3* and the *recipients* will be referred to as *WF#2.IN#1,*

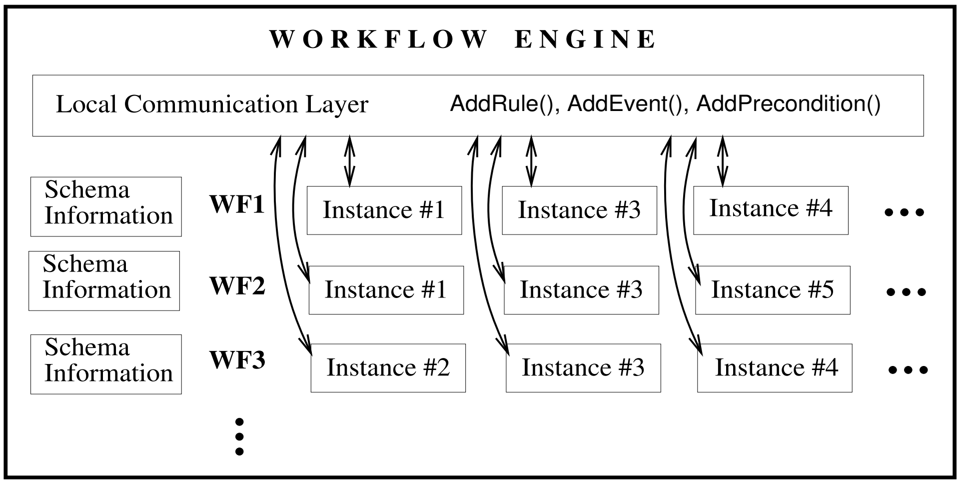

Figure 6.3 Communication in Central Workflow Control

WF#2.IN#3, etc. This naming scheme is important since messages passed between specific instances of the workflow have to be correctly delivered.

To enforce mutually exclusive execution of a step or a group of steps, it is necessary to ensure that no other step from another workflow that requires common resources in a conflicting mode is executing concurrently. Using the low level implementation mechanisms discussed in section 5.2, the engine ensures that no conflicting step is already in execution and also prevents other conflicting steps from starting until the step that is being executed completes. Enforcing such a requirement is easy since the engine needs to refer to the data structures (data table, event table, step status table, etc. all located on a single node) to process the API calls for the low level primitives. Since a workflow is navigated from a single node, the mechanism for enforcing mutual exclusion for a group of steps is the same. Instead of notifying completion of a step, a notification using the *AddEvent()* call is made after the completion of the group of steps.

The number of messages exchanged between the instances for mutually exclusive execution of a step can be reduced by performing certain optimizations. For example, if steps from a workflow at workflow engine A conflict with steps from multiple instances of another workflow executing at workflow engine B, then the workflow instance at engine A can be notified (*AddEvent()* response) using a single message upon the completion of all the conflicting steps at engine B rather than multiple messages upon the completion of each of the steps.

Another enhancement to the mutual exclusion mechanisms is needed to avoid deadlock (mutual wait) scenarios. Upon compilation of the workflow schemas, *static priorities* are assigned by the workflow compiler to steps that are defined to have conflicts. Later, in the event of a conflict the step with a higher priority executes first. Now consider two steps that can potentially conflict and are unaware of the existence of the other. Suppose messages are exchanged between the respective workflow instances almost concurrently to check if the conflicting steps are in execution. The static priority information is used to unilaterally decide which step to execute first, thus avoiding a deadlock situation.

To enforce relative ordering of steps between a pair of workflows, conflicting steps from the workflows must be scheduled to maintain the same precedence order. To fix the precedence order the first pair of conflicting steps must be executed in a mutually exclusive manner as discussed earlier results in one workflow becoming the leading workflow and the other becoming the lagging workflow. Any subsequent

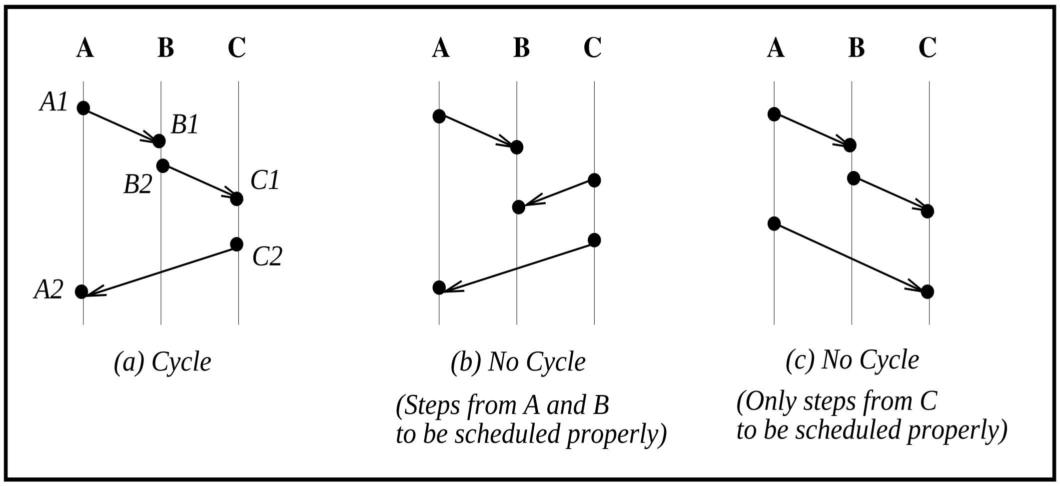

Figure 6.4 Cycles in Relative Ordering of Workflows

conflicting steps between the workflows must be scheduled in the same order. Every conflicting step from a lagging workflow will have to wait for the completion of the corresponding conflicting step from the leading workflow. Hence, a lagging workflow must be notified about the completion of steps from the leading workflow, which is again achieved by exchanging data and event information between the workflow instances within a single node. Note that a given workflow may have one or more leading workflows. Hence, a step from a lagging workflow may have to wait for the completion of events from more than one instance before execution. As discussed earlier in *chapter 3*, it is essential to prevent stalling (or starvation) of the lagging workflow. So, it must be appropriately notified if a different course of action is taken in a leading workflow such that the events in the leading workflow that the lagging workflow is waiting for will never happen. An example of such a situation is an if-then-else branching in a leading workflow which may take a branch such that the steps which will conflict with those from the lagging workflows will never be executed. This will be true only if the failure of any of the steps along the branch taken will not require the leading workflow to rollback to the branching step and take a different course of action that might cause the execution of the conflicting steps. This information is extracted statically at schema compilation time by verifying that any of the steps along the branch do not have a rollback requirement beyond the branching point. In such a scenario, the leading workflow will just signal the completion of the events the lagging workflow is waiting for using the *AddEvent()* call. This will ensure that the lagging workflow will never starve.

Deadlock avoidance techniques have to be incorporated into the mechanisms needed for relative ordering. *Figure 6.4* shows three workflows A, B and C that have a relatively ordering requirement between each. From *figure 6.4 (a)* it can be seen that cycles (ordering sequence $A \rightarrow B \rightarrow C \rightarrow A$) can occur when enforcing relative ordering of workflows. Assume that the relative order $A \rightarrow B$ has already been established between the two workflows. Now workflow C is scheduled. If proper care is not taken while scheduling steps from workflow C, a relative order of $B \rightarrow C$ and $C \rightarrow A$ is produced. Such an ordering can result in deadlocks if three subsequent conflicting steps (one from each workflow) have relative ordering requirement with respect to each of the workflows. None of the steps can now execute since the execution of one will violate one of the relative ordering requirements. In transaction processing systems a deadlock situation is handled by aborting one of the transactions. In contrast workflow aborts are more expensive (amount of lost work is large) and should be avoided. Hence, it is crucial to schedule the first pair of conflicting steps between the workflows such that deadlocks do not occur in the future. For the same set of workflows, the relative ordering schedules shown either in *figure 6.4 (b) or (c)* are permissible, *i.e.*, schedules do not have cycles. Of the two schedules, the one shown in *figure 6.4 (c)* is preferable for the following reason. Since the relative order between workflows A and B has already been fixed, steps of workflow C have to be scheduled such that there are no cycles and it is easier to achieve this requirement since scheduling of steps from only one workflow is performed properly. More specifically, if it is ensured that the conflict steps of a workflow are scheduled such that the workflow will lag with respect to all other workflows with which it has a relative ordering requirement, then no cycles will be formed. To achieve this, the call made to *AddRule()* for ensuring mutual exclusion of the first pair of steps between two workflows should have its last argument as $Finish$ instead of $FinishOrBlock$ which will ensure that the calling workflow will always become a lagging workflow. This technique is automatically enforced by the mechanisms that implement the low level mechanisms.

The rollback dependency between a pair of workflows is established if a step from two different workflows conflict and establish an ordering such that the compensation of the leading step requires a rollback in the workflow containing the lagging step.

First, the low level primitives are enacted in the same manner as they are used for mutual exclusion (Figure 5.3) to establish a rollback dependency between the pair of workflows that have the conflicting steps. Once a dependency is established the lagging step (workflow) is notified only if the leading step is compensated. Note that a lagging step may be rollback dependent on more than one leading step and the compensation of a leading step may require the rollback of more than one lagging step. The dependency is established as follows. Similar to the halt event associated with every branch for handling race conditions, an instance level halt event is established that corresponds to the *step.compensated* event of the leading step. On the compensation of the leading step, a *step.compensated* event is generated and delivered to all the lagging workflows. The receipt of this event causes all lagging workflows to halt their execution. Then appropriate action is taken in the individual workflows as specified in the workflow schema, *i.e.*, rolling back the workflow to some previous point and re-executing from there or aborting the workflow. If a rollback has happened in the lagging workflow, then the halt event received will be discarded. Although communication is required between workflows to establish the dependency and for subsequent notification of a step compensation, it is easy to achieve in a central engine since all the workflow state information is available at a single node. Note that failure or compensation of a step in a leading workflow can also cause rollback in the lagging workflow if the lagging step also has a rollback specification. No special mechanisms are needed to handle this situation since the engine will rollback the lagging workflow in the same manner it would handle an actual failure of the lagging step.

6.1.3 Handling Workflow Changes and Aborts

This section discusses how the central engine handles user initiated workflow abort and workflow input data changes.

The API for changing workflow inputs is as follows:

WorkflowChangeInputs(WorkflowName, InstanceNum, NewArguments)

Customers who have existing orders can make changes to those orders. For example, a customer could increase the number of items ordered after an initial order has been placed. Since the number of items is an input to the workflow that is instantiated to process the order, changing the details of the order translates to changing the workflow input information when the instance is in execution. The customer changes are first propagated to the workflow engine through the *WorkflowChangeInputs()* interface. Note that the customer order number is translated into the appropriate instance number using the mapping information in the front end database. Until when the *WorkflowChangeInputs()* interface can be invoked is dependent on the specific workflow schema. This can be customized by specifying the requirement in the workflow schema as follows:

WORKFLOW_CHANGE_INPUTS S8;

. This specification states that the workflow inputs can be changed as long as the flow of control in the workflow has not reached step S8.

Based on changes to the workflow inputs, the engine determines the step to which the workflow should be rolled back. Essentially, the workflow is rolled back to the earliest affected step (closest to the start step) affected by the change. For a sequential workflow, the earliest affected step can be determined statically and is the first step of the workflow that requires the input that has been changed. For more complex workflows that contain control structures such as if-then-else branching and parallel branching, the affected step(s) to which the workflow should be rolled back to has to be determined dynamically at run time. For example, in a workflow with an if-then-else branch, two steps that require the changed input may be located on different branches. The earliest affected step can only be determined dynamically at run time depending on which branch is taken at the branching point. In the case of parallel branching, the rollback requirements is more complicated because several branches are in execution in parallel and steps along one or more of the branches may require the changed input. Hence, in workflows with parallel branches each branch may have to be individually rolled back to the earliest affected step on that branch. Also, race conditions can result in parallel branches during rollback. They require the use of the solution described in the next subsections.

To perform the rollback itself, the engine waits for the completion of steps that have already been scheduled with the agents. After such steps complete the engine does not schedule any new steps. Instead a rollback is performed to the earliest affected step and the opportunistic compensation and re-execution strategy is used to re-execute the workflow.

The API for a user initiated workflow abort is as follows:

WorkflowAbort(WorkflowName, InstanceNum)

Consider a user initiated workflow abort using the *WorkflowAbort()* interface. As mentioned earlier, a user's (customer's) cancelation order is translated in a workflow abort using the mapping information stored in the front end database. The customer does not know the instance number of the workflow processing the user's order. The abort request can be processed as long as the workflow has not been committed. Since the entire state information of an instance is available at the engine, an abort request can be processed quickly. Once it is determined that the abort request can be processed, steps which are to be compensated are compensated (as specified in the workflow schema) and the workflow is aborted. The instance table information for that workflow is archived and the status of the instance is marked as aborted in the instance summary table.

6.1.4 Handling Failures

On the failure of a step, there are two possibilities. As specified in the workflow schema, either the workflow is aborted (action taken is similar to a user initiated abort) or the workflow is rolled back to some earlier step and the step is compensated and re-executed. Determining the steps that are affected by a step failure is very much dependent on the control flow specified in the workflow schema. Steps that fail on a sequential path or along a branch taken in an if-then-else branch require simple handling. The workflow is rolledback to some previous step based on the rollback specification of the failed step and the workflow continues its forward execution from there.

Failure handling requirements are complex if the failed step exists on a parallel branch (thread). If steps along multiple parallel threads fail, a rollback is performed along each of the individual threads. Also, if rollback is required beyond the branching point (to a step earlier than the branching point), the workflow is rolled back to the earliest (closest to the start step) of all the steps as determined from the rollback specification for each of the failed steps. Whenever a rollback is needed to or beyond a parallel branching point, other concurrent threads that emanated from the branching point are halted since the inputs to the steps along these branches might be affected by the output produced from the re-execution of the branching step. Halting a thread corresponds to quiescing control flow along that thread. No new steps (that may be eligible for execution) are scheduled for execution along that thread. The mechanism for halting a thread is as follows: If the branching point or a step that executed before the branching point is compensated then a halt event is generated. The halt event is uniquely named with the name of the branching step appended to it, *e.g.* if the branching step is *S8*, then the event name is *H8*. Each branch has a halt event associated with it. This event name can be determined statically by analyzing the workflow structure and determining all the branching steps on a path from the start step. Since multiple branching steps can be encountered on such a path, a thread can have multiple halt events associated with it. Any one of the events will cause the thread to halt. Thus all parallel branches emanating from a step have the same halt event. When a halt event appropriate for a thread occurs, the engine halts the execution of that thread. Since the state information of the entire instance is available at a single node (the halt event is generated on the same node), halting the other threads is easy.

Once the affected threads are halted the engine then uses the opportunistic compensation and re-execution strategy to re-execute the rest of the workflow. The mechanisms are as follows. Just as in the case of normal execution, a rule that triggers the execution of a step is fired when all the specified events events occur. However, instead of immediately requesting the agent to execute the step, the compensation and re-execution condition is checked first to determine the exact course of action, *i.e.*, whether the step is partially compensated and incrementally re-executed or whether a complete compensation and re-execution is needed. The condition is evaluated by referring to the values of the different data items in the data table and step status table, including inputs to the previous execution of the step. If a re-execution is not necessary then a *step.done* event is generated, else a message is sent to the appropriate agent to perform the compensation and the re-execution. If a step is compensated and it is part of a compensation dependent set then the other members of the set that executed after the step are also compensated in the reverse execution order before the step is compensated and re-executed. To compensate the members of the set, the engine communicates with the agents responsible for executing the steps. Since the entire workflow state information is available at a single node, compensating the set in the reverse execution order is easy. Control can flow out of a loop when a step in a loop fails. Control cannot enter a loop at some intermediate step if the loop has been executed more than once. If a loop is executed more than once then there is ambiguity in the semantics of a compensation dependent set. Consider for example steps S5 and S8 that are part of a loop that executed 10 times. If these two steps are part of a compensation dependent set, it is not clear as to which of the executions of steps S5 and S8 have to be compensated and in what order. Due to this ambiguity steps of a loop are not allowed to be part of a compensation dependent set. A simple alternative however is to compensate the steps in the loop in the reverse execution order of the loop. The net affect is the same as unwinding the loop executions into a single sequential path and compensating the steps that are part of a compensation dependent set in the reverse execution order, *i.e.*, S8 and S5 from 10th execution, S8 and S5 from the 9th execution and so forth.

If multiple agents are eligible to execute a step, then the compensation and re-execution of a step can be performed at an agent different from the one that previously executed the step. Also, if a step need not be re-executed no message is sent to the agent responsible for executing that step. Also, if a step has already been compensated since it was part of a compensation dependent set, then the step is re-executed without checking the compensation and re-execution condition. Another important requirement during workflow re-execution has to do with if-then-else branching. If a branch different from the previous execution is taken, steps on the previously executed branch have to be compensated. Note that some of the steps on such a branch may already have been compensated since they were part of a compensation dependent set. Hence, a *compensation thread* is initiated at the branching point along the old branch. This thread compensates all steps along the branch until it meets a confluence point. The above mentioned procedure continues until the rest of the workflow is committed. If there is another step failure, the same procedure is repeated. All these mechanisms satisfy the *failure handling requirements 11, 12, 14, 16, 17, and 18* stated in section 4.3.

Handling rule execution in the context of rollbacks is tricky. In particular the *step.done* event generated for several completed steps is invalid if the workflow is

Pending Rule Table

Event	*Rule*
S2.D	R5, ~~R7~~
S3.D	~~R9~~
•••	•••

Event Table

Event Name	*Valid*	*Time*
WF.S	Y	9/20/97/10:20
S1.D	Y	9/20/97/10:32
S2.D	~~Y~~ N	9/20/97/10:56
S3.D	~~Y~~ N	9/20/97/11:12
S4.D	~~Y~~ N	9/20/97/11:45
•••	•••	•••

Figure 6.5 Table Changes on Workflow Rollback

rolled back to some previously executed step. As part of the compensation and re-execution phase, several of the steps are to be re-executed and hence the appropriate rules have to be fired again. Thus, it is necessary to carefully determine which of the rules have to be fired again and which ones need not be. Also, if rules are fired inadvertently based on the invalidated events, race conditions can arise. For example, re-execution of a step can result in the consideration of a new rule whose only other event is the completion of another step which has been previously executed but is now invalid. Thus the rule can be inadvertently fired based on the new execution of a step and old execution of the another step resulting in a race condition and causing inconsistency. Care should be taken that the rule is fired only based on the new execution of a step if the previous execution has been invalidated due to the rollback. This race condition resembles the race condition studied in figure 2.3. However, it is different in the sense that it can occur even in a sequential workflow where a step requires data input from two or more steps.

The next focus is on run time mechanisms for handling rule execution in the context of rollbacks such that race conditions are prevented. Once a rollback point (step to which the workflow is rolled back) has been determined, the *step.done* events of all steps that succeed the rollback point and have completed have to be invalidated. This invalidation is performed with the help of a matrix that provides successor relationship between steps which is determined statically by analyzing the workflow schema and is then stored in a matrix. By applying transitive closure, the successor relationship matrix contains entries corresponding to any pair of steps in a workflow. The successor relationship is based on both control flow and data flow specification. The relationship is false if the steps do not have any data or control flow between them. Given an ordered pair of steps, the matrix can be used to determine whether the successor relationship is true for that ordered pair. As shown in figure 6.5, if a step is a successor of the rollback point (S2 in the figure) then its *step.done* event is invalidated. Also rules in the pending rule table from which the invalidated *step.done* events have been deleted are discarded to ensure that incorrect rules will not be fired. From the perspective of rule execution, the invalidation of events and discarding of pending rules mimics the state of the workflow in which none of the succeeding steps of the rollback point have executed. From this state onwards, event occurrence and rule firing happens in a normal manner as the opportunistic compensation and re-execution strategy is applied. The invalidation strategy is very simple and general enough for all types of control structure in a workflow. The invalidation strategy avoids both types of race conditions that have been discussed thus far. A rule that triggers the execution of a step along a sequential path is not fired based on the previous execution of a step since the effects of all steps that succeed a rollback point are invalidated. Also, in the case of parallel branching if control flow along some of the branches have already reached the confluence point, the steps along such branches are invalidated if they are successors to the rollback point. Thus, the rule that triggers the execution of the confluence point will be triggered only by new executions of the last steps along all incoming branches.

To handle agent failures the logging based approach outlined earlier in chapter 3 is used. Hence, if a request was sent to an agent that has failed, the engine will wait for the agent to be up again to determine the status of the request. This is required only for steps that perform updates on resources. If the step simply performs a query then the step is attempted at a different agent (if such an agent exists) without waiting for the failed agent to come up. Information about the type of step, *i.e.*, update or query is obtained from the step table in the workflow class table. Note that if several agents are eligible to execute a step and if one or more of the eligible agents are down then the engine chooses an agent from the remaining eligible agents. If such an agent does not exist then the engine does not schedule the step for execution until a suitable agent is available.

6.1.5 Interactions Between Coordinated Execution and Workflow Rollback

We now briefly discuss how rollbacks in workflows affect coordinated execution requirements of the same and other concurrent workflows.

In the context of mutual exclusion, rollbacks have little effect except in the case of mutual exclusion of a group of steps. For an individual step, mutual exclusion may be needed for step compensation as well. By adopting the same mechanism used for mutual exclusion of a step execution, mutual exclusion can be ensured for step compensations as well. To specify mutual exclusion for step compensation, minimal enhancement to LAWS in the form of a special clause indicating mutual exclusion for both forward execution and compensation is sufficient. Now consider rollback in a group of steps being executed in a mutually exclusive manner. If the rollback goes beyond the start of the group, the workflow calls the *AddEvent()* API to indicate the completion of the group execution (as if the group never executed).

If the rollback happens within the group then execution continues normally and when all the steps of the group have completed, then the *AddEvent()* API is called to indicate the completion of the group execution.

Rollbacks in workflows also have an impact on relative ordering of steps across workflows. For example, some of the steps of the leading workflow may be compensated and re-executed again. Since the initial execution of those steps happened ahead of the steps of the lagging workflow, the status of the leading workflow does not change. The workflow's status is still maintained as leading if it was already leading. However, the rollback and re-execution might cause delays in the lagging workflow. Note however that if the leading workflow takes a course of action different from the original one, then the lagging workflow must be suitably notified to prevent starvation. Specifically, step completion events corresponding to those steps in the leading workflow that will never be executed must be delivered to the lagging workflow.

6.1.6 Summary

This section discussed detailed mechanisms for handling the various functionality required at the central workflow engine. We discussed the required APIs, data structures, rule based execution support and other mechanisms needed to handle various control flow requirements (such as branching, loops and nested workflows) and failure handling/coordinated execution requirements. This discussion will form the basis for developing mechanisms for the required functionality in parallel and distributed workflow control architectures in the subsequent sections.

The state transformation of a step in a workflow is shown in *Figure 6.6*. A step is initially in an *inactive* state. When a workflow event occurs that is partly responsible for triggering the execution of a step, the step enters the *waiting* state where it waits for other necessary workflow events to occur. After all the required workflow events complete (related to data and control flow dependencies), if any

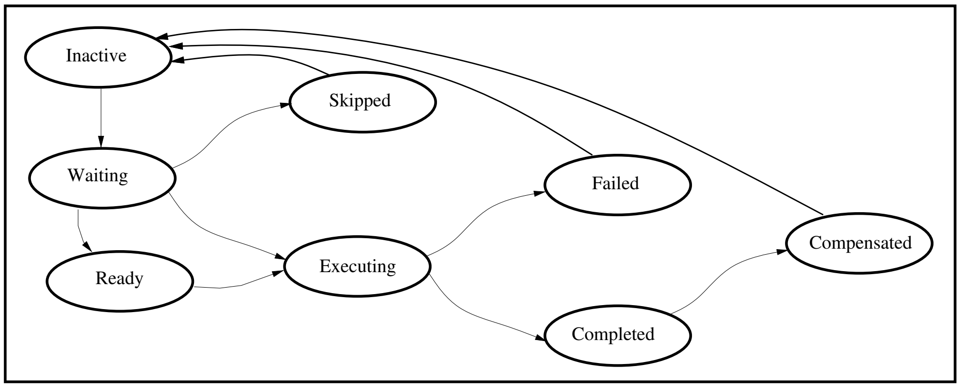

Figure 6.6 State Transition Diagram for a Step

associated condition (only in the case of if-then-else branch) evaluates to TRUE, the step enters the *executing* state else it enters the *skipped* state indicating that it need not be executed. If there are no conditions, the step directly enters the *executing* state. Also if the step is to be performed by a human, the step enters the *ready* state and when a staff member selects it for execution, the step enters the executing state. If the step executes successfully, it transitions from the executing to the *completed* state else if it fails it transitions to the *failed* state. A skipped or a failed step could enter the waiting state during the re-execution phase. Also, a completed step can enter a *compensated* state if during the re-execution phase its effect is not necessary.

6.2 Enhancing Mechanisms for Parallel and Distributed Control

This section briefly highlights the enhancements needed to the mechanisms developed for central workflow control for use in parallel and distributed control. The enhancements depend on the way in which the state information of a single workflow and all the workflows is maintained. *Table 6.1* lists the mechanisms for centralized control and indicates which of them need enhancements for use in parallel and distributed control.

Table 6.1 Enhancements for Parallel and Distributed Control

Central	*Parallel*	*Distributed*
Normal Execution:		
Data Structures		$\surd$
Sequential Navigation		$\surd$
Control Structures		$\surd$
Multiple Eligible Agents		$\surd$
Workflow Completion		$\surd$
Coordinated Execution:		
Message Exchange	$\surd$	$\surd$
Mutual Exclusion	$\surd$	$\surd$
Relative Ordering	$\surd$	$\surd$
Rollback Dependency	$\surd$	$\surd$
Workflow Changes & Aborts:		
Workflow Changes		$\surd$
Workflow Aborts		$\surd$
Failure Handling:		
Halting Thread		$\surd$
Opportunistic Compensation & Re-execution		$\surd$
Rule Execution & Rollback		$\surd$
Race Conditions		$\surd$
Agent Failures		$\surd$

Entries marked with a $\surd$ require enhancements

In parallel workflow control, each workflow is controlled by a single workflow engine and the state information of an individual workflow is still maintained at a single node just as in the case of central workflow control. Hence, mechanisms for normal execution, handling workflow changes and aborts, and failure handling do not need any changes. However, the mechanisms need to be enhanced in order to enforce coordinated execution since each engine maintains the state information of only the workflows it controls. Hence, communication is needed between the workflow engines to exchange state (data and event) information to perform coordinated execution. For example, to execute a step S in a mutually exclusive manner at a workflow engine, it is necessary to ensure that no other step controlled by a different workflow engine is executing a step in a conflicting mode with respect to step S. Since the additional messages exchanged between the engines are physical messages on the network, they have an impact on scalability and hence it is necessary to improve the protocols to reduce the number of additional messages exchanged.

In distributed workflow control, workflow navigation happens in a distributed fashion and, hence, the state information of an individual workflow may not be available at a single node. Agents exchange packets of workflow state information to perform navigation. Hence, enhancements are needed for the various mechanisms required to handle the normal execution of workflow. Specifically, handling the various control structures, choosing an agent for executing a step amongst multiple eligible agents and handling workflow completion require communication between the agents (to help in scheduling decisions). Similarly, handling workflow changes/aborts, halting a thread, handling opportunistic compensation and re-execution all require communication between agents to correctly execute a workflow. To enforce coordinated execution requirements such as relative ordering, needs to be exchanged (on top of those exchanged to handle normal execution and failures) between agents to transfer state and event information. In summary, mechanisms for distributed workflow control are quite different from those of centralized control and require carefully designed protocols to ensure that the requirements are correctly met. To maintain scalability, it is also necessary to reduce the number of physical messages exchanged between agents for handling normal execution, failures and enforcing coordinated execution. How this is accomplished is discussed in sections 6.3 and 6.4 respectively.

6.3 Execution Support for Parallel Workflow Control

This section discusses extensions to the mechanisms developed for centralized workflow control for use in parallel workflow control. Recall that in a parallel workflow environment, a number of workflow engines interact with the same set of agents that a single engine would interact with in a central workflow control environment. The increased number of engines provide scalability since the navigation load is shared by the engines. A workflow instance is controlled by only one engine, chosen based on the load at the different workflow engines. Workflow instances do not migrate from one engine to another. Hence, no change is required in the mechanisms for handling normal execution of workflows, user initiated workflow changes and aborts, and handling failures within a workflow. However, extensions are needed for handling coordinated execution since the global state information is distributed across workflow engines. Specifically, communication is needed between engines since data and event information is exchanged across workflow instances controlled by different engines. The rest of the section provides the details of the extensions needed to support coordinated execution.

6.3.1 Enforcing Coordinated Execution

Support for the APIs that handle the low level primitives *AddRule(), AddEvent() and AddPrecondition()* for centralized control ensure that the coordinated requirements are satisfied. In parallel workflow control, an additional layer on top of the local communication layer is required as shown in *figure 6.7* to ensure that remote data and events (generated at other engines) are available to an engine in the same way as locally generated data and events. The remote communication layer implements the API such that the contents of the outgoing messages are delivered to the remote communication layers of the other engines and the incoming messages are delivered to the local communication layer. Thus an *AddRule()* call made by a

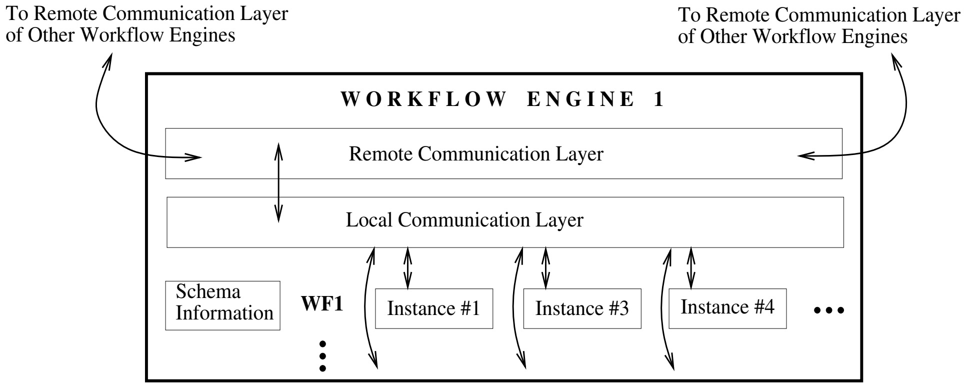

Figure 6.7 Communication in Parallel Workflow Control

workflow instance will return only after the local layer communicates with the local workflow instances and the remote layer communicates with the remote workflow instances, thus ensuring that the coordinated execution requirements are satisfied will respect to all the workflow instances.

For example, if a *AddRule (WF2, ...)* API is invoked by instance #3 of WF1, a message is sent to the local communication layer which routes the message to *all local active* instances of WF2. In addition, it passes a copy of the message to the remote communication layer which routes the message to all the other workflow engines. When the remote communication layers in the other workflow engines receive this message, they pass it on to the local communication layers which route the information to *all instances* of WF2 *local* to those engines (*remote instances* to the sending workflow). To properly keep track of the *sending* workflow instance and the *receiving* workflow instance, apart from the workflow schema name and instance ID that is prepended to the instance data by the local communication layer, the workflow engine ID is also prepended by the remote communication layer. Hence in the above message, the *sender* will be referred to as *EN#1.WF#1.IN#3*. This will ensure that the messages are delivered correctly to the appropriate workflow instance being executed anywhere in the parallel environment. If the engine numbers are not included then the front end database has to be contacted to determine the engine on which a workflow instance is running.

In this architecture steps are executed in a mutually exclusive manner as follows: Before the execution of a step the *AddRule ()* API is called and a message is sent to the local communication layer to check if any of the conflicting steps are in execution elsewhere *in the system*. The local layer in turn passes the message to the remote layer which broadcasts it to the other workflow engines. At the other workflow engines the remote layer passes the message to the local communication layer which checks if any of the conflicting steps are in execution within that engine. If there are such steps in execution then they are allowed to complete. After the completion of the steps, response messages are transmitted back via the remote layer and then delivered to the workflow instance via the local layer. Then the step is executed. The local and remote communication layers similarly handle the other APIs for enforcing all the coordinated execution requirements.

6.3.2 Techniques for Reducing Message Overheads

It is possible that potentially conflicting steps from different workflows are executing at the same agent. In such cases conflicts between steps can be detected

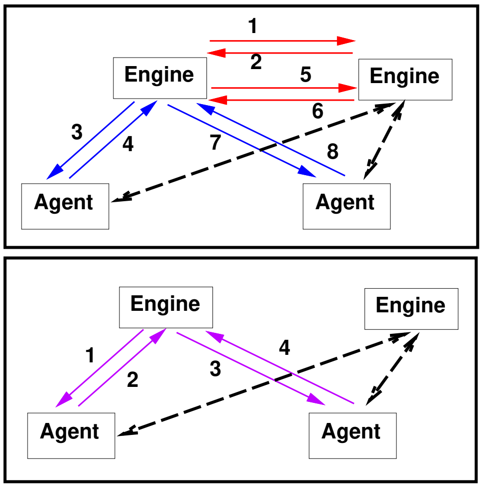

Figure 6.8 Reducing Messages by Piggybacking Requirements

at the agent rather than at the engine. Note that the procedure for determining conflicts is still based on the conflict conditions. In a central workflow control conflicts are always detected at the engine and hence at a single node. When conflicts are detected at a single node, additional messages need not be exchanged. Thus detecting conflicts at the agents has a significant impact on the number of messages exchanged between engines. The conflict definitions themselves can be preloaded at the agents when the compiler compiles the workflow schema or sent to the agents dynamically by piggybacking the conflict condition along with the step execution request. For example, if step A is to be executed in a mutually exclusive manner, A and other steps that have been defined to potentially conflict with A have to be scheduled at the same agent. The agent will ensure that if there are other steps that conflict with A they will not be executed concurrently. If two workflows have a relative ordering requirement then the first pair of steps to be ordered can be scheduled at an agent. The agent executes the step in some order and piggybacks the ordering information along with the results it sends to the engine. Once the engine has knowledge of a conflict or the ordering information, it can enforce relative ordering requirements for the rest of the steps by piggybacking the ordering requirement along with the step execution request to the agent. The agent in turn will enforce the ordering requirement by scheduling the steps in the requested manner. This is shown in *figure 6.8.* Essentially, the conflict and ordering information is known to the engine via the agents instead of exchanging messages with other engines. Thus the additional messages required between the engines using the original schema (messages marked 1, 2, 5, and 6 in the top figure) can be eliminated using this strategy. Since the agents detect conflicts, they need to persistently log input parameters of the steps as well as the outputs produced. Handling conflicting steps at the same agent and piggybacking conflict/ordering information onto the step execution request reduces the number of messages exchanged between engines for enforcing coordinated execution requirements.

Workflow engines can become unavailable due to system failures. Such failures have an impact on coordinated execution of workflows. If an engine fails then several workflow instances at other engines may wait indefinitely for a response. The failure can be ignored by other engines by timing out the connection to the failed agent and ignoring the lack of response. However, this will affect correctness since the engine could already have scheduled some conflicting steps (with steps from workflows managed by other engines) with the agents. One correct solution to handle this situation is to consider one of the available engines as a replacement (determined using one of the standard election algorithms [34]). All the engines then would resend their requests to the replaced engine to perform coordinated execution. The replaced engine then polls all the agents and gets information about the steps that have been scheduled by the failed engine. The agents respond to the replaced engine after the completion of the steps scheduled by the failed engine. If the failed engine comes up then it has to contact the replaced engine to determine the status of the steps it had scheduled when it failed such that there are duplicate executions.

6.4 Execution Support for Distributed Workflow Control

This section extends the mechanisms developed for centralized workflow control for use in distributed workflow control. Before discussing the various mechanisms in detail, we provide a quick overview of the concepts involved.

6.4.1 Overview

Recall that in a distributed workflow control, a number of workflow agents interact with each other to perform distributed navigation of workflow instances. The absence of central control using a workflow engine provides scalability. The state information of a workflow however is not available at a single node but flows from node to node and, hence, enhancements are needed for most of the mechanisms. Other than the ability to handle workflow packets, agents must also be able to communicate with each other to handle failures and coordinated execution requirements. Hence the agents support a number of Application Programming Interfaces (APIs). These APIs are enumerated in *Table 6.2*.

Some of the APIs are invoked from the front end database and others from agents. All the mechanisms that will be discussed in distributed workflow control are based on these APIs — an agents exchanges messages with other agents to invoke the appropriate APIs. A summary of which of the APIs are used to handle normal execution, failure handling and coordinated execution requirements is shown in *Table 6.3*. The individual sections details how the APIs are used. Since the number of messages must be minimized in order to preserve scalability, it is necessary to explore all opportunities for piggybacking useful information onto workflow packets.

Table 6.2 Types of Messages in Distributed Workflow Control

Supported By	*API*	*Invoked By*
Coordinator Agent	WorkflowStart	Front End Database
Coordinator Agent	WorkflowChangeInputs	Front End Database
Coordinator Agent	WorkflowAbort	Front End Database
Coordinator Agent	WorkflowStatus	Front End Database
Agent	InputsChanged	Coordinator Agent
Agent	StepExecute	Agent
Agent	StepCompensate	Agent
Agent	StepCompleted	Agent
Agent	StepStatus	Agent
Agent	WorkflowRollback	Agent
Agent	HaltThread	Agent
Agent	CompensateSet	Agent
Agent	StateInformation	Agent
Agent	AddRule	Agent
Agent	AddEvent	Agent
Agent	AddPrecondition	Agent

Table 6.3 Use of APIs in Distributed Workflow Control

API	*Used For*
WorkflowStart	Normal Execution
WorkflowChangeInputs	Failure Handling
WorkflowAbort	Failure Handling
WorkflowStatus	Normal Execution
InputsChanged	Failure Handling
StepExecute	Normal Execution
StepCompensate	Failure Handling
StepCompleted	Normal Execution
StepStatus	Failure Handling
WorkflowRollback	Failure Handling
HaltThread	Failure Handling
CompensateSet	Failure Handling
StateInformation	Normal Execution
AddRule	Coordinated Execution
AddEvent	Coordinated Execution
AddPrecondition	Coordinated Execution

These issues are discussed in the context of normal execution, failure handling and coordinated execution of workflows.

6.4.2 Normal Execution

Since navigation is performed in a distributed fashion each node has to maintain workflow schema information and also the state information of the workflow instances whose steps it has executed. Hence, the distributed agents use the same data structures used by a central engine — workflow class tables with links to the corresponding workflow instance tables. The instance summary table in a workflow class table does not contain a status field since not all agents maintain the status information of the workflow instances, *i.e.*, whether instances are executing, completed or have been aborted.

Table 6.4 Data Structures for Managing Workflow Status Information

Coordinator Instance Summary Table

Workflow Name	*Instance Number*	*Status*
1	1	Executing
3	2	Committed
4	1	Aborted
1	2	Executing

An agent that is responsible for executing the first step of a workflow is referred to as the *workflow coordinator*. The workflow coordinator has the additional responsibility of maintaining the status of the workflows (whether the workflow has completed or is in execution) it started and handling user requests on those workflows. Unlike a central engine the coordinator is not responsible for maintaining the entire state of the workflow and for performing the actual navigation of the workflow. The coordinator agents maintain a separate *coordinator instance summary table* as shown in *Table 6.4*. The coordinator instance summary table is used for handling all front end database requests including workflow status information. The front end database interacts only with coordinator agents.

The main loop at an agent waits for messages (API calls) from other agents or the front end database. Based on the type of message received a different course of action is taken. To instantiate a workflow the front end database invokes the *WorkflowStart()* API at the coordinator agent of the workflow. When a workflow is instantiated at a coordinator agent a workflow instance table is created to store the workflow state and context information. If the workflow instance is the first instance of a particular workflow class then the workflow class table is created. The workflow instance table is then linked to the workflow class table. A $workflow.start$ event is then generated which triggers 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. After this, the execution of the start step and other related events are similar to those of a general step as discussed below.

The API for executing a step at an agent is as follows:

StepExecute(WFPacket)

Whenever a *WorkflowStart()* or *StepExecute()* API is invoked, the data and event table of that instance is updated based on the call information. The call information actually consists of a workflow packet similar to the one shown in figure 6.9. Based on the updated information in the event table, both new and pending rules are considered for firing. New rules are those that have not been fired and are considered from the general-rule table. If a new rule requires additional events before it can be fired, then it is added to the pending-rule table. Rules in the pending-rule table which have been waiting for the events present in the packet that just arrived are also fired if they do not need any other events. Rules are fired only after examining that their conditions evaluate to true. When a rule is fired it triggers the execution of a step. The information needed to execute the step such as the workflow name, instance number, step number, name of the program to be executed, and the value of input data items to the step are available locally in the workflow class and instance tables. After the step is executed, the agent then examines the results produced to check if the step was successfully executed. If the step was successful then the agent updates the data table to store the output produced by the step and generates a *step.done* event. If the step fails, a $step.fail$ event is generated. The event table is also updated. After a step execution is complete, the state information is assembled into a packet and sent to all the agents responsible for handling the succeeding step(s) using the *StepExecute()* API. As shown in *figure 6.9*, the packet contains the data and event information present in the workflow instance table corresponding to the workflow instance the step belongs to. This contains information received by the agent from one or more agents and the information

Workflow Name: WF2
Instance Number: 4
Action: Execute S3
Data Items:
WF.I1 = 90
WF.I2 = Blower
S1.O1 = 20
S1.O2 = Gasket
S2.O1 = 45
S2.O2 = 400
Events:
WF1.S
S1.D
S2.D

Figure 6.9 Workflow Packet Used in Distributed Workflow Control

produced by the execution of the step at the agent itself. The information shown in the packet is slightly different from the one shown in *figure 2.6* as we also pass event information required for the rule based navigation. We refer to the agent sending the packet as the *predecessor agent* and the agent receiving the packet as the *successor agent*. The flow of workflow packet actually corresponds to both data flow (an agent may need to send data to another agent executing a step from a different parallel branch of the same workflow) and control flow.

Since navigation is performed by the agents themselves in a distributed manner, the various control structures *i.e.*, sequential flow, parallel branching, if-then-else branching and branch-joins as specified in the workflow schema have to be handled correctly. A rule that fires a step in a sequential path will require only the occurrence of one event (corresponding to *step.done* of the previous step) to be fired. However, the rule may require other *step.done* events depending on which of the steps it gets its input data from. If all the required events have not yet arrived, the rule may have to wait in the pending-rule table for future packets to arrive that carry the required events. After all the required events have occurred the rule is fired resulting in the execution of the step. All these mechanisms satisfy *requirements 1 and 2* stated in section 4.3. In the case of an if-then-else branching, after a *step.done* event is generated on the completion of the branching step, the workflow packet is sent to the two agents that are responsible for executing the succeeding steps along the branches. After both the agents receive the packet, only one of the rules will fire based on which branching condition evaluates to true. The packets that trigger the rest of the steps in a branch operate in the same way as in a sequential path. All these mechanisms satisfy *requirements 3, 4, 5, and 6* stated in section 4.3. In the case of parallel branch, the workflow packet information is sent to all the agents that are responsible for executing the succeeding steps along the various branches after a *step.done* event is generated for the branching step. The rest of the processing at the agent is the same as that for a step on a sequential path.

Finally, consider branch-joins that occur at confluence steps. The execution of a step is triggered when control flow along all the incoming branches reaches the step. The rule that triggers the execution of a confluence step will be fired at the occurrence of *step.done* events for the last steps of all incoming branches. Hence, the rule may have to wait in the pending rule-table for packets to arrive from all the agents that are responsible for handling the final steps of all incoming branches. As each packet arrives, the workflow instance table is updated with the data and event information contained in the message. When all the required events are available, the confluence step is executed. All these mechanisms satisfy *requirements 7, 8, 9, and 10* stated in section 4.3. Loops and nested workflows are handled similarly. In the case of loops, the agent responsible for executing the last step of the loop sends a workflow packet to the first step of the workflow if the loop exit condition evaluates to false. For executing a nested workflow, the agent executing the step preceding the nested workflow sends the workflow packet to the agent responsible for executing the first step of the nested workflow. Similarly, after the completion of the nested workflow, the coordinator agent of the nested workflow sends the workflow packet to the agent responsible for executing the step succeeding the nested workflow.

Another important aspect of scheduling has to do with multiple agents being eligible to execute a step as specified in the workflow schema (step table). In such a situation, a suitable successor agent has to be selected to execute the succeeding step. A simple solution would require the predecessor agent to use a two phase process to make the decision. In the first phase, it gets the state information of an agent by invoking the *StateInformation()* API of the successor agents. The successor agents then respond with this information. The predecessor agent then picks the successor agent with for example the minimum load to execute a step. When multiple successor agents are eligible to execute a confluence step, determining the successor agent has to be handled correctly since several predecessor agents may be involved in the decision making process. Each predecessor agent obtains state information from the successor agents at different times depending on when their step completes. Hence, some of the predecessor agents may have stale information about successor agent states. Then the predecessor agents will have to exchange the state information they have about the successor agents among themselves to determine which successor agent should execute the step. A uniform and better solution is to use the "leader election" algorithm that allows the successor agents themselves determine which agent will execute the step. When all the successor agents have received messages from all the predecessor agents, they can call the *StateInformation()* API of each other and choose the agent with the least load. This scheme also eliminates the need for successor agents to send state information to predecessor agents.

As discussed earlier, a workflow is committed when the final step along all active paths of execution have executed successfully. This implies that the final step along a sequential thread, along an if-then-else branch and along all parallel branches must be completed before a workflow is determined to be committed. Since the state information is distributed across agents, the agents handling the execution of such steps send a message to the coordinator agent by calling the *StepCompleted()* API to notify the completion of the step. The coordinator agent waits for the arrival of such messages from all the agents that are responsible for executing the final steps along all active paths. In the case of nested workflows, upon the commit of a child workflow the coordinator agent sends a workflow packet to the next agent responsible for executing the successor step at the parent level. Note that the step completion messages just contain steps completion information and are not the same as workflow packets. The API for notifying the coordinator agent about step completion is as follows:

StepCompleted(WorkflowName, InstanceNum, StepNum)

The coordinator agent knows from which of the agents it has to receive step completion messages before it determines that the workflow is committed. The coordinator agent gets this information from the compilation process which can statically determine the list of such agents from the workflow schema. Once a coordinator agent receives the required messages, a *workflow.done* event is generated and the coordinator agent commits the workflow. Committing a workflow corresponds to making its changes permanent. This implies that the workflow will not be rolled back in the future, *i.e.*, the effect of the individually committed steps of the workflow will not be compensated. Also, any request for aborting the workflow or changing the workflow inputs after a workflow commit will be rejected. After a workflow is committed, the instance table information is archived and the status of the instance is marked as committed in the coordinator instance summary table. Periodically the coordinator agents broadcast information to the other agents about the committed workflows so that they can purge their instance tables can be purged from the agents.

6.4.3 Enforcing Coordinated Execution

In the discussion on enhancements needed for the parallel workflow environments to enforce coordinated execution, we discussed the use of a remote communication layer to support API calls for the low level primitives across workflow engines. Additional enhancements are now presented to handle coordinated execution in distributed workflow control.

Just as in the case of a parallel workflow engine, an agent in a distributed workflow environment has a local communication layer and a remote communication layer for handling the API calls for coordinated execution. The local layer handles communication with workflows whose steps are executing locally at the same agent and the remote layer communicates with workflows whose steps are executing at other agents. To properly keep track of the *sending* workflow instance and the *receiving* workflow instance, apart from the workflow schema name and instance ID that is prepended to the instance data by the local communication layer, the workflow engine ID is also prepended by the remote communication layer. If a *AddRule (WF2, ...)* API is invoked by an instance #3 of WF1 from agent #1, the *sender* will be referred to as *AG#1.WF#1.IN#3* and the *recipients* will be referred to as *AG#2.WF#2.IN#1, AG#3.WF#2.IN#3,* etc. This will ensure that the messages are delivered correctly to the appropriate workflow instance being executed anywhere in the parallel environment. If the agent numbers are not included then the response may have to be sent to all agents eligible to execute that step. Thus when calls are made to *AddRule(), AddEvent() and AddPrecondition()* APIs, the message is routed to relevant local instances and remote instances.

Mutually exclusive execution of steps is handled as follows. The local communication layer checks whether a conflicting step is in execution locally at the agent. The remote communication layer helps perform the same check at the other agents that are eligible to execute the conflicting steps. When both the calls return there

Workflow Name: WF2
Instance Number: 4
Action: Execute S3
Data Items:
WF.I1 = 90
WF.I2 = Blower
S1.O1 = 20
S1.O2 = Gasket
S2.O1 = 45
S2.O2 = 400
Events:
WF1.S
S1.D
S2.D
R.O. Leading:
WF3.I5
WF4.I3
R.O. Lagging:
WF5.I2

Figure 6.10 Workflow Packet for Coordinated Execution

are no other conflicting steps in execution at any agent and the step is executed. Unlike parallel workflow control, the primitives and hence the checks are directly supported at the agents themselves.

Relative ordering requires significant enhancements to the mechanisms since the individual steps executed at different agents and the navigation are also performed in a distributed manner. In central and parallel workflow engines, the ordering information is available on the same node on which step scheduling is performed. However, in distributed workflow control, the first pair of conflicting steps is ordered at one agent and the rest of the steps to be ordered are performed at different agents. Thus successor steps of the lagging workflow have to wait for the corresponding steps from the leading workflow to complete. The agents responsible for steps in the lagging workflow that have to be ordered have to be notified. To save the number of messages exchanged between agents, the best way to pass ordering information to agents is to include it in the workflow packet information. Thus a workflow packet contains state information and relative ordering information as well as shown in *Figure 6.10.* The ordering information in the packets gives the instance identifiers of workflows that are leading and the workflows that are lagging. While unpacking the contents of a workflow packet, the agent checks to see if the step needs to be ordered after the execution of a conflicting step from another workflow. If so, the agent waits for a message to arrive (*AddEvent()* API call) from the agent eligible to execute the conflicting step (this information is obtained from the workflow class table available locally). Also if there are other agents to be notified (those from lagging workflows) after the completion of the step, then the agent invokes the *AddEvent()* API call of those agents to notify the completion.

Rollback dependency between steps of concurrent workflows is handled as follows. When a leading step of a rollback dependency is rolled back, an *AddEvent()* call is received by the agent of the lagging step. Upon receipt of this event, the workflow is partially rolled back and re-executed in the same manner as the failure of a step.

6.4.4 Handling Workflow Changes and Aborts

This section discusses how the agents handle user initiated workflow abort and changes to workflow instance data.

Customer changes are first propagated to the coordinator agent through the *WorkflowChangeInputs()* interface. The front end database has information about the workflow name, instance number and the coordinator agent of the specific workflow processing the request. Before accepting changes the coordinator agent checks if the workflow is still in execution. If so it accepts the change request else it rejects the request. Based on the changes to the workflow inputs, the coordinator agent determines the step to which the workflow should be rolled back. Essentially, the workflow is rolled back to the earliest step (closest to the start step) affected by the change. For a sequential workflow, the earliest affected step can be determined statically and is the first step of the workflow that requires the input that has been changed. Hence, the coordinator agent invokes the *InputsChanged()* API of the agent responsible for executing the affected step. If multiple agents are eligible to execute the step, then the coordinator agent invokes the API call with all the agents who are eligible to execute the step. For more complex workflows that contain control structures such as if-then-else branching and parallel branching, the affected step(s) to which the workflow should be rolled back has to be determined dynamically at run time. For example, in a workflow with an if-then-else branch, two steps that require the changed input may be located on different branches. The earliest affected step can be thus determined dynamically at run time depending on which branch is taken. Since the coordinator agent does not have knowledge of which branch was taken at the various if-then-else branches within a workflow, it calls the *InputsChanged()* API at the agents that could have executed the affected step. Hence, the number of agents to which the message is to be sent to is bounded by the number of agents that are eligible to execute a step. In the case of parallel branching, the rollback requirements is more complicated since several branches may be executing in parallel and steps along one or more of the branches may require the changed input. Hence, each branch may have to be individually rolled back to the earliest affected step on that branch. Again, the coordinator agent calls the *InputsChanged()* API at the agents that could have executed the affected step.

The API for notifying an agent about inputs that have changed is as follows:

InputsChanged(WorkflowName, InstanceNum, NewArguments)

When a *InputsChanged()* request is received by an agent, it checks to see if it has executed the step in question. If it has not executed the step then no further action is taken. If it has executed the step then the step may have to be compensated and re-executed. Since successor steps in the workflow will be affected by a re-execution, this also requires halting the threads of execution and using the opportunistic compensation and re-execution scheme, both of which are explained in the following sub-section.

The next discussion is on user initiated workflow aborts using the *WorkflowAbort()* interface on the coordinator agent. A customer's cancelation order is translated into a workflow abort using the mapping information stored in the front end database. The abort request can be processed as long as the workflow has not been committed. The coordinator agent verifies this fact by referring to the coordinator instance summary table. Once it is determined that the abort request can be processed, steps which are to be compensated are compensated (as specified in the workflow schema). To compensate the steps the coordinator agent invokes the *StepCompensate()* API of the agents responsible for the execution of those steps. The coordinator agent abort the workflow by halting the various threads of execution starting from the first step. Then the status of the instance is marked as aborted in the coordinator instance summary table.

6.4.5 Handling Failures

This section discusses failure handling issues which require special attention in the case of distributed workflow control. Since the steps are executed at different agents and the state information may not be available at a single node, irrespective of whether the step failed on a parallel path or a sequential path, the procedure for rolling back the workflow and re-executing the workflow is the same. This procedure automatically handles race conditions as well. Since a rollback and re-execution may affect most of the successor steps of the rollback step, all the affected threads are halted. Halting a thread corresponds to quiescing control flow along that thread. The mechanism for halting the affected threads in distributed workflow control is as follows: The agent where a step failure occurred calls the *WorkflowRollback()* API of the agent responsible for the step to which the workflow is rolled back. This information is static and is available at the agent after the workflow schema has been compiled. None of the other agents that executed steps of that workflow are notified.

As soon as an agent receives a *WorkflowRollback()* request, probe threads have to be activated to visit agents that are responsible for executing the successor steps. Hence the agent responsible for executing the rollback step calls the *HaltThread()* API of all the successor agents of the step.

The API for an agent to request halting of a thread is as follows:

HaltThread(WorkflowName, InstanceNum, OriginStep)

where the *OriginStep* is the step from which the workflow is being re-executed after a rollback. It is used in the ensuing discussion to determine which of the steps will be affected by a rollback.

When an agent receives a *HaltThread()* request it does the following. It refers to the state information of that workflow and invalidates the *step.done* events corresponding to steps that are successors of the *OriginStep*. Also rules in the pending rule table from which the invalidated *step.done* events have been deleted are discarded to ensure that incorrect rules will not be fired. If the agent is executing a step belonging to the instance then after the completion of the instance workflow packets are not sent to any of the successor agents. This results in quiescing the thread of control flow. If the agent has already sent workflow packets to the successor agents, then the agent in turn invokes the *HaltThread()* on the successor agents. Thus the *HaltThread()* call propagates from agent to agent until it reaches an agent that quiesces the thread. Note that as part of the rollback, events corresponding to the completion of step which are later rolledback have to be invalidated. This objective is also achieved as a side effect of halting the threads. From this state onwards, event occurrence and rule firing happen in a normal manner as the opportunistic compensation and re-execution strategy is applied. Our two pronged strategy of sending probes along the agents to halt the execution of the affected threads and invalidating old events is very simple and general enough for all types of control structures in a workflow. The invalidation strategy avoids race conditions.

The API for an agent to request compensation of a set of steps in the reverse execution order is as follows:

$$\mathsf{CompensateSet(OriginStep,\ StepList)}$$

where the *OriginStep* is the step from which the compensation of a dependent set is requested. It is used in the ensuing discussion.

Once the affected threads are halted each agent then uses the opportunistic compensation and re-execution strategy to re-execute the remaining steps of the workflow. Agents transfer workflow packets to successor agents using the same *StepExecute()* API. When an agent receives a *StepExecute()* call it checks to see if the step that that is referred to in the call has already been executed. If the step has been executed then it automatically applies the opportunistic compensation and re-execution strategy, else the step is executed. The opportunistic compensation and re-execution strategy works as follows. Just as in the case of normal execution, a rule that triggers the execution of a step is fired when all the specified events events occur. However, instead of immediately executing the step, the compensation and re-execution condition is checked first to determine the exact course of action, *i.e.*, whether the step is to be partially compensated and incrementally re-executed or whether a complete compensation and re-execution is needed. The condition is evaluated by referring to the values of the different data items in the data table and step status table, including inputs to the previous execution of the step. If a re-execution is not necessary then a *step.done* event is generated, else the step is compensated and then re-executed. If a step is compensated and it is part of a compensation dependent set then the other members of the set that executed after the step are also compensated in the reverse execution order before the step is compensated and re-executed.

The mechanism for opportunistic compensation and re-execution is very different from central workflow control and is as follows: The agent invokes the *CompensateSet(WFName,InstanceNum,OriginStep,StepList)* API of the agent responsible for executing the last step of the compensation dependent set. On the invocation of the *CompensateSet()* API the agent checks if the step in question has been executed. The step is also deleted from the StepList. If the step has not been executed then no action is required. If it has been executed then it is compensated using the information available locally at the agent. Then the agent invokes the *CompensateSet()* API of the agent responsible for executing the last step of the new StepList. This procedure continues until the last remaining step (originally the first step) of the StepList is compensated. Then the workflow packet is sent to the successor agent and this procedure continues until the workflow is committed.

Another important issue during workflow re-execution has to do with if-then-else branching. If a branch different from the previous execution is taken, steps of the previously executed branch have to be compensated. Note that some of the steps on such a branch may have already been compensated since they were part of a compensation dependent set. Hence, a *compensation thread* is initiated at the branching point along the old branch by invoking the *CompensateThread()* API . The agents successively calls this API until all steps along the branch are compensated prior to a confluence point. The above mentioned procedure continues until the workflow is committed. If there is another step failure the same procedure is repeated. All these mechanisms satisfy the *failure handling requirements 11, 12, 14, 16, 17, and 18* stated in section 4.3.

To handle agent failures the following approach is used. Agent failures occur in two capacities — one of the predecessor agent and one of the successor agent. Let us first consider the failure of a successor agent. An agent that is responsible for executing a step preceding a given step is referred to as a predecessor agent. A successor agent is one that is responsible for executing the succeeding step. If only one agent is eligible to execute a step and it is unavailable then the predecessor waits for that agent to come up. If several agents are eligible to execute a step and if one or more of the eligible agents are unavailable then the predecessor agent chooses a successor agent from the remaining eligible successor agents. If such an agent does not exist then the agent waits to send the workflow packet until a suitable agent becomes available. The failure of a predecessor agent requires a more complex solution. For those pending-rules that require only one event and have been in that state for a set timeout period, the successor agent polls the agents responsible for executing that step about the status of the step using the *StepStatus()* API. If the actual agent that was selected to execute the step is unavailable, the successor agent will get a response from the other eligible agents that the status as "unknown". If the step is designated as an update step then the successor agent has to wait for the failed agent to come up. Otherwise, if the step is performing a query then the successor agent requests the execution of that step by invoking the *StepExecute()* call at one of the available predecessor agents. Note that information about the type of step, *i.e.*, whether it is an update or a query is obtained from the step table in the workflow class table. A final possibility is that one of the predecessor agents respond with the status of the step as "executing" then the successor agent will wait for the packet to arrive from that predecessor agent.

6.4.6 Techniques for Reducing Message Overheads

Reducing the number of messages exchanged to improve scalability has been a primary concern throughout the design of the various mechanisms for normal execution, failure handling and coordinated execution. Here are some specific examples:

- To invalidate old and incorrect events such that race conditions are eliminated, instead of using a separate thread the same probe thread used for halting a thread of execution is used to invalidate events thereby reducing message overheads associated with extra threads

- To communicate ordering information between agents, instead of sending separate messages to specific agents the information is piggybacked on to the workflow packet exchanged between agents.

- To reduce communication between agents when workflow inputs are changed, the coordinator agent determines the earliest affected step and informs the agent(s) responsible for that step to compensate and re-execute the steps of the workflow instead of compensating and re-executing from the start step

We assume that a *reliable multicast* mode of data transmission is available in the environment. This can help reduce the number of physical copies of message since group communication is used extensively in distributed workflow environments. The following APIs supported by agents require sending the same message to a group of agents:

- *StepExecute()* — when multiple successor agents are eligible or when multiple agents may execute succeeding step as in the case of if-then-else or parallel branching

- *InputsChanged()* — when a coordinator agent has to inform a set of agents about inputs changed

- *StepStatus()* — when a successor agent enquires about the status of a preceding step from multiple agents

- *HaltThread()* — similar situation as *StepExecute()*

- *StateInformation()* — when a set of successor agents have to request state information from each other

When several instances of workflows are running concurrently in a distributed workflow environment of a large organization, multicasting can significantly reduce the number of physical copies of messages between agents.

6.5 Performance Issues

This section addresses performance related issues of the run-time mechanisms. The primary concerns include the processing load at an engine or an agent and the number of messages exchanged in the system.

We first discuss performance issues related to the opportunistic compensation and re-execution strategy. To implement this strategy, the overheads include maintaining additional data that correspond to the previous execution of the steps in the data structures 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 substantial. However, if the step involves either accessing a number of records from a database or human oriented tasks like moving inventory then the savings are considerable. Since many steps in real-world workflows correspond to the second variety, in general the benefits from the OCR scheme is considerable while paying a small overhead.

The rest of the analysis focuses on the load at a node and the number of messages exchanged based on various parameters and the values we have assumed for them. The table in *Table 6.5* shows the parameters and the value range that has been assumed. The value ranges were chosen based on intuition since performance studies related to workflow execution in the presence of failures and under different architectures are not available. *Tables 6.6, 6.7 and 6.8* summarize the approximate expressions that can be used to determine the load at a node or the number of messages exchanged for the various mechanisms under different control architectures. Using the approximate values and expressions it is possible to get a feel for how the various mechanisms compare under the different architectures. In this analysis normal execution is treated as the situation where coordinated execution is not

Table 6.5 Parameters used in Analysis

Parameter	*Symbol*	*Value Range*
Number of Steps per Workflow	s	5 - 25
Number of Workflows Schemas	c	20
Number of Concurrent Instances per Schema	i	10 - 1000
Number of Engines	e	1 - 8
Number of Agents	z	10 - 100
Number of Eligible Agents per Step	a	1 - 4
Number of Conflicting Definitions per Step	d	0 - 2
Number of Steps Rolled Back on a Failure	r	1 - 10
Number of Steps to be Invalidated on a Step Failure	v	0 - 8
Number of Final Steps in a Workflow	f	1 - 4
Number of Steps to be Compensated on a workflow abort	w	0 - 4
Number of Steps/WF needing Mutual Exclusion	me	0 - 4
Number of Steps/WF needing Relative Ordering	ro	0 - 4
Number of Steps/WF having Rollback Dependency	rd	0 - 2
Navigation and Other Load per Step (# of instructions)	l	—
Probability of Logical Step Failure	pf	0.0 - 0.2
Probability of Workflow Input Change	pi	0.0 - 0.05
Probability of Workflow Abort	pa	0.0 - 0.05
Probability of step re-execution	pr	0.0 - 0.5

required. The normalized values assigned for the expressions are based on the average values of the parameters specified in *Table 6.5.* Note that these expressions do not account for message reduction optimizations.

The expressions for load comparison are based on the estimated number of steps or other actions that would be performed at the engine in case of central and parallel control and at an agent in the case of distributed control. Similarly, the expressions that give the number of messages physically exchanged are based on the interactions between the various nodes of the system under different architecture, *i.e.*, engine with agent in a centralized architecture, engines with engines and engines with agents in a parallel architecture and agents with agents in a distributed architecture.

•*Normal Execution:* The scheduling load is based on the number of steps to be executed for all instances of all workflows schemas. The scheduling load is shared by the engines in the case of parallel control and the agents in distributed control.

Table 6.6 Load and Physical Messages in Centralized Workflow Control

Mechanism	*Expression*	*Normalized Value*
Load at Engine	**per instance**	
Normal Execution	$l \cdot s$	$15 \cdot l$
Workflow Input Change	$l \cdot r \cdot pi$	$0.125 \cdot l$
Workflow Abort	$l \cdot w \cdot pa$	$0.0500 \cdot l$
Failure Handling	$l \cdot r \cdot pf$	$0.5 \cdot l$
Coordinated Execution	$l \cdot (me + ro + rd) \cdot s$	$75 \cdot l$
Physical Messages Exchanged	**per instance**	
Normal Execution	$2 \cdot s \cdot a$	60
Workflow Input Change	$2 \cdot r \cdot pi \cdot pr \cdot a$	0.125
Workflow Abort	$2 \cdot w \cdot pa \cdot a$	0.2
Failure Handling	$2 \cdot r \cdot pf \cdot pr \cdot a$	0.5
Coordinated Execution	0	0

Since the number of agents (z) typically far outnumber the number of engines (e), the load is moderate at a parallel engine and the lowest at a distributed agent. The number of messages exchanged in the system is the same for central and parallel control but is lesser for distributed control since the number of final steps is always far less than the total number of steps in a workflow. Thus both from the perspective of engine load and number of messages, a distributed control architecture is very scalable for normal execution.

• *Workflow Input Changes:* The load factor here depends on the probability of workflow input change and the number of steps to be rolled back. Since the load is again shared by the engines in the case of parallel control and agents in the case of distributed control, the distributed agents are the most lightly loaded. The number of messages exchanged is again the same for central and parallel workflow control. However, for distributed workflow control, the number of messages is very much dependent on the number of steps to be invalidated which corresponds to the number of agents to be traversed for halting the threads affected by the change. If the number of such agents to be traversed is smaller than the number of steps crossed during

Table 6.7 Load and Physical Messages in Parallel Workflow Control

Mechanism	*Expression*	*Normalized Value*
Load at Engine	**per instance**	
Normal Execution	$l \cdot s \ / \ e$	$3.75 \cdot l$
Workflow Input Change	$(l \cdot r \cdot pi\ )$/e	$0.0313 \cdot l$
Workflow Abort	$(l \cdot w \cdot pa\ )$/e	$0.0125 \cdot l$
Failure Handling	$(l \cdot r \cdot pf\ )$/e	$0.125 \cdot l$
Coordinated Execution	$(l \cdot (me + ro + rd) \cdot s\ )$	$75 \cdot l$
Physical Messages Exchanged	**per instance**	
Normal Execution	$2 \cdot s \cdot a$	60
Workflow Input Change	$2 \cdot r \cdot pi \cdot pr \cdot a$	0.125
Workflow Abort	$2 \cdot w \cdot pa \cdot a$	0.2
Failure Handling	$2 \cdot r \cdot pf \cdot pr \cdot a$	0.5
Coordinated Execution	$(me + ro + rd) \cdot e \cdot s$	300

rollback then distributed control requires fewer messages. Thus on an average the three architectures are comparable in terms of the number of messages. Since the overall probability of workflow input changes is small, they do not affect system scalability in a big way.

• *Workflow Abort:* The mechanisms considered here are for user initiated workflow abort where some of the executed steps are compensated and the workflow is aborted. From the perspective of the load, distributed agents again have the lowest load. In terms of the number of messages exchanged, the engines know the node at which a step was executed whereas the coordinator agent may have to send messages to all eligible agents (for executing the steps) to be compensated. Hence central and parallel workflow control may have an edge over distributed control depending on the number of agents eligible for executing the steps. Since the overall probability of workflow aborts is small, they do not affect system scalability in a big way.

• *Failure Handling:* The load actually depends on the probability of step failures and the number of steps crossed during workflow rollback. Since the load is shared, distributed agents again have the lowest load. In terms of the number of messages

Table 6.8 Load and Physical Messages in Distributed Workflow Control

Mechanism	*Expression*	*Normalized Value*
Load at Engine	**per instance**	
Normal Execution	$l \cdot s \ / \ z$	$0.3 \cdot l$
Workflow Input Change	$(l \cdot r \cdot pi)/ \ z$	$0.0025 \cdot l$
Workflow Abort	$(l \cdot w \cdot pa)/ \ z$	$0.001 \cdot l$
Failure Handling	$(l \cdot r \cdot pf)/\mathrm{z}$	$0.01 \cdot l$
Coordinated Execution	$(l \cdot (me + ro + rd) \cdot a \cdot d \cdot s)/\mathrm{z}$	$1.5 \cdot l$
Physical Messages Exchanged	**per instance**	
Normal Execution	$s \cdot a + \mathrm{f}$	32
Workflow Input Change	$(r + v) \cdot pi \cdot a$	0.45
Workflow Abort	$2 \cdot w \cdot pa \cdot a$	0.2
Failure Handling	$(r + v) \cdot pf \cdot a$	1.8
Coordinated Execution	$(me + ro + rd) \cdot a \cdot d \cdot s$	150

exchanged the situation is similar to that of workflow input changes and there is no clear winner. If the number of agents to be traversed for invalidating previous step executions and halting the affected threads is smaller than the number of steps crossed during rollback, then distributed control requires less number of messages. Thus on an average the three architectures are comparable in terms of the number of messages.

• *Coordinated Execution:* The load primarily depends on the number of steps for which coordinated execution requirements of mutual exclusion, relative ordering and rollback dependency have been specified. Note that the loads are comparable at a central and a parallel engine since the number of engines, e, cancel out both at the numerator and the denominator. Also, the factor $a \cdot d/z$ is usually a small fraction and hence distributed agents are the least loaded. In terms of the number of messages, a central engine is clearly a winner since no physical messages are exchanged. Between a parallel engine and a distributed agent there is no clear winner. If the factor $a \cdot d$ is less than e, then distributed agents use fewer messages else a parallel engine uses lesser number of messages. Since the probability of the

Table 6.9 Recommended Choice of Architectures for Various Requirements

Criteria	*Normal*	*Normal + Failures*	*Normal + Coordinated*
Load at Engine	(1) Distributed	(1) Distributed	(1) Distributed
	(2) Parallel	(2) Parallel	(2) Parallel
	(3) Central	(3) Central	(3) Central
Physical Messages	(1) Distributed	(1) Distributed	(1) Central
	(2) Parallel	(2) Parallel	(2) Distributed
	(2) Central	(2) Central	(3) Parallel

existence of coordinated execution requirements for a step is the factor $(me + ro + rd)/s$, coordinated execution requirements affect the scalability of parallel and distributed control architectures.

We have performed a comparison of the various mechanisms for different architectures in terms of the scheduling load and the number of messages exchanged. From the perspective of load on a node, a distributed agent is the least loaded and hence very scalable. From the perspective of the number of messages exchanged, the choice depends largely on the extent of coordinated execution requirements. The common case is that the number of steps that require coordinated execution is small. In such a scenario, distributed workflow control is a clear choice. In the unlikely case that several steps have coordinated execution requirements then central or parallel control is preferable. Even in such cases, the performance of parallel and distributed control can be improved such that they are still scalable by enforcing certain requirements (*e.g.*, conflicting steps to be executed on the same agent). This observation is consistent with the scalability of decentralized algorithms in general [17], *i.e.*, mechanisms that require fewer message exchange between nodes scale well. A summary of the recommended choice based on various requirements is shown in *Table 6.9*.

6.6 Implementation of LAWS Compiler and CREW Run-Time Environment

The CREW environment consists of a heterogeneous collection of nodes as shown in *Figure 6.11*. Any of these nodes can act as agents or engines in central, parallel or distributed control architectures. Hence to achieve total portability across heterogeneous architectures, the LAWS compiler and the CREW run-time environment have been completely implemented using the *Java* programming language and development kit [60].

The LAWS compiler consists of several class definitions (data, step, workflow, etc.) defined for various levels of workflow information. These class definitions are reused in the CREW run-time environment. The exception handling facility of Java has been used effectively in handling syntax and semantic errors in the workflow specification so that error messages provide detailed information to the workflow designer.

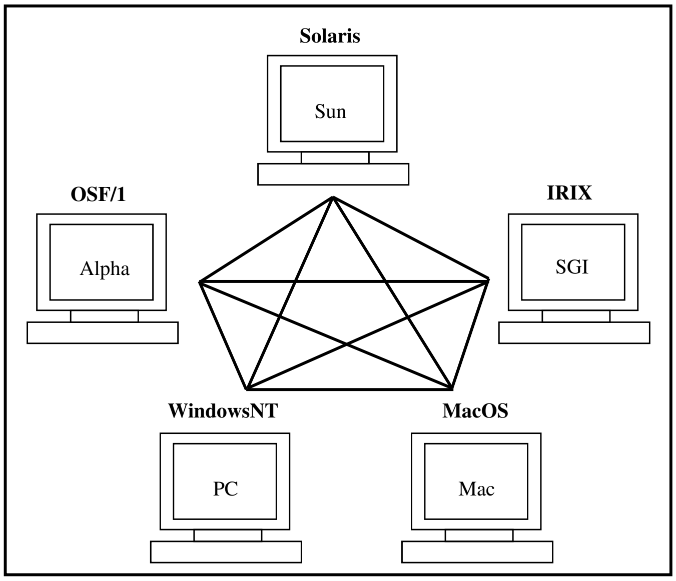

Figure 6.11 Nodes in the CREW environment

The CREW run-time environment has to manage several workflow instances concurrently. Also, workflow instances may have two or more steps from concurrent branches executing simultaneously. These requirements have been efficiently achieved in the CREW run-time environment using the *multithreading* facility available in Java. Consider for example the structure of the workflow engine (which in itself is a O.S. thread). When a workflow instantiation request is received, a separate thread called *WorkflowExecutor* is created to handle that specific workflow instance. The state data corresponding to the workflow instance, including the events that have occurred, rules that have been fired and the data that have been produced by the steps is available only to the specific WorkflowExecutor thread and the workflow engine. The execution and compensation of a step is handled by a *StepExecutor* thread. This facilitates the simultaneous execution of several steps within a workflow instance.

A *RuleHandler* thread is responsible for firing the rules for a specific workflow instance. One of the interesting functions of the RuleHandler is to evaluate composite events. Recall from section 5.1.2 that the events that comprise a composite event typically correspond to step completions. The steps complete at different points in time. Hence, a composite event is evaluated over a period of time. As and when the specified events occur, the rule state information is updated. When all the specified events have occurred the composite event evaluates to TRUE. Then if there is a condition associated with the rule it is evaluated. If the condition evaluates to TRUE or if there is no condition the rule is fired. If one or more events specified in the composite events never occur then the rule will never be fired. This typically happens in if-then-else branching cases where one of the two rules (either the if case or the else case) is never fired. Note that the rules managed by the RuleHandler are a copy of the rules generated for that workflow class. This is necessary since the rules specific to the instance are modified during the course of execution of the instance. The RuleHandler threads within the WorkflowExecutor thread and the StepExecutor threads read and write state information concurrently. The *synchronize* primitive of Java has been very useful in handling queries and updates to state information. The agents (both central and distributed) also use multithreading to efficiently supervise execution of concurrent steps. Notably, the distributed agent has a *Workflow Mapper* thread which unpacks the workflow packet and constructs the workflow state information. When a step is eligible for execution is calls the *Step Executor* thread.

In the present version of the prototype support exists for normal execution and failure handling of steps under central and distributed workflow architectures. The main objective has been to build a prototype of WorldFlow (discussed later in chapter 7 for which support is needed only for failure handling since it is difficult to achieve coordinated execution due to the autonomy of the Web servers that act as agents). The agents currently invoke simulated applications developed in Java. Each of these simulated applications are assigned a *failure probability.* By synthetically providing workflow input data and changing the failure probability of the applications, the experiments have been conducted to test the functioning of the CREW run-time environment. The experimentation was done by creating ten different workflow schemas and executing several instances of the workflows concurrently. Execution traces for the instances were taken and examined. The traces verified that the system correctly handles the requirements specified in the workflow schema definitions.

The use of simulated applications does not limit the use of the CREW environment in real world scenarios. Currently there are several initiatives and even products focusing on the interaction of Java with different types of information sources. Some of these include: Microsoft's ActiveX [2] which allows Java components to communicate with other application development tools, Sun's Java Database Connectivity (JDBC) API [7] and Java Transaction Service (JTS) API [6] that allow Java components to talk to database resource managers and transaction monitors, Sun's Java IDL [5] and Iona's OrbixWeb IDL [8] that allows Java components to manipulate CORBA objects [3] at remote servers. These interfaces allow the development of several types of applications and hence the CREW run-time environment (and the technology) should be able to control and manage several real world workflow applications.

6.7 Comparison with Other Possible Approaches

This section compares the CREW approach with other approaches from the literature when they are able to handle the problems addressed in this thesis. Both failure handling and coordinated execution requirements are considered.

6.7.1 Failure Handling Requirements

•*Rollback and Re-execution:* Use of any other possible approach for performing partial rollback and re-execution such as spheres of joint compensation [86] will result in decreased efficiency due to unnecessary compensation and re-execution of steps. The notion of lazy compensation [45] has been proposed as a techniques to handle workflow rollback and re-execution. Instead of user specified conditions, the decision to compensate and re-execute a step is based purely on data flow dependencies between the steps. There is provision for achieving the same effect in the opportunistic compensation and re-execution technique where the condition for step re-execution can be specified as 'INPUTS_CHANGED'. The changes are determined automatically based on data flow specification in the schema. Lazy compensation is just one of the many choices provided by the opportunistic compensation and re-execution technique for workflow rollback and re-execution. Thus, we believe that the opportunistic compensation and re-execution technique is a very powerful yet flexible scheme to achieve workflow rollback and re-execution.

• *Halting Thread Execution:* To reduce the amount of wasted work, the execution of affected threads is halted in a workflow. Exactly which of the threads are affected depends on the extent of rollback, *i.e.*, to which previous branching point is the workflow rolled back. An alternative to this approach it to let all the threads of execution continue. While this approach certainly minimizes overheads in terms of additional "probe threads" to be invoked to halt execution on the affected threads, the strategy may not be efficient since steps that need not be executed may be executed perhaps with inconsistent input data.

6.7.2 Coordinated Execution Requirements

• *Mutual Exclusion:* Another way of achieving mutual exclusion of workflow steps that access conflicting resources is by modifying the resource managers so that they prevent conflicting accesses. If there are a variety of resource managers supplying data in a workflow environments, then all resource managers that do not prevent concurrent accesses to objects have to be modified to support mutual exclusion whereas a general purpose mechanism at the WFMS level is provided in CREW to achieve mutual exclusion not just for a step but for a group of steps. Another possible approach is to define "virtual resources" and allow steps to execute only after acquiring locks on the virtual resources. This approach was not used since the aim was to develop a uniform approach for handling the different coordinated execution requirements. Since there exists support for exchanging event information across nodes in the rule-based framework, it is possible to incorporate other mutual exclusion algorithms.

• *Relative Ordering:* In database systems "ladder climbing" is a technique that is used to order accesses to index structures by multiple transactions. This technique ensures that the nodes along a single path down an index structure are accessed in the same relative order between concurrent transactions. By constructing an analogy between the nodes of the index structure and the steps of a workflow, and by requiring steps to acquire locks on virtual resource ladder climbing can be used as a technique for enforcing relative ordering. By acquiring locks on virtual resources at the workflow class level, ladder climbing can be used to enforce relative ordering only between instances of the same workflow. Ladder climbing cannot be used for ensuring relative ordering between steps of workflows whose class definitions are different since the structure of the workflows will be different.

• *Rollback Dependency:* In the context of database transactions the notion of spheres of control [41] has been proposed to automatically track dependencies between all transactions. Information about spheres can be used to achieve rollback dependency. However, due to the complexity of maintaining the required information to track dependent spheres it is difficult to realize this in practice. In contrast, the CREW techniques for handling rollback dependency track specific dependencies and hence only a limited amount of information is to be tracked. This helps in considerably reducing the complexity of the system while still ensuring that the specified dependencies are satisfied.

6.8 Summary

This chapter discussed detailed mechanisms for supporting normal execution, failure handling and coordinated execution of workflows. We first presented the mechanisms for a central workflow architecture where the state information of all the instances reside on a single node. The mechanisms were then extended for use in parallel and distributed workflow control. For parallel workflow control extensions are needed only for handling coordinated execution requirements since the state of a single workflow still resides on a single node. For distributed workflow control additional mechanisms and protocols were developed to handle normal execution, coordinated execution and failure handling. We have tried to minimize the number of messages exchanged in parallel and distributed workflow control architectures to preserve scalability. In particular, piggybacking is used so that as much state (data and event) information as possible is passed with the workflow packets exchanged between agents.

Based on the various factors that can affect the performance of the mechanisms, the suitability of the architectures for different requirements were also discussed. A brief description of the implementation of the Java based prototype system was also presented. Finally, the mechanisms used in CREW were compared with alternative approaches.

CHAPTER 7

EXTENDING SYSTEM SUPPORT FOR WEB WORKFLOWS

This chapter first discusses the extensions to the prototype system to realize WorldFlow, a Web based WFMS. It then provides examples of applications that can be developed using WorldFlow.

7.1 Web WFMS

In order to exploit the potential of the Web, companies offering workflow products now provide a *Web interface* to WFMSs [1, 4]. This allows both the employees and the customers of a company to interact with the WFMS through the Web. Employees can access their 'to-do' list via a Web browser instead of using a special client as in traditional WFMSs. However these WFMSs do not permit workflows whose steps directly access and update data from Web servers. In short, they support customer-to-business transactions over the Web but not business-to-business transactions. In order to conduct electronic commerce, WFMS support is needed for business-to-business transactions. Hence WFMSs should be enhanced with capabilities so that workflows can directly access information by interacting with Web servers. This allows companies to automate their business processes on the Web by allowing steps to automatically query/update information from Web servers located both within and outside the organization.

A WFMS for the Web differs from a traditional WFMS as follows. The Web is essentially a client-server based distributed system where browsers acting as clients communicate with Web servers using the HTTP protocol [9]. When a workflow is

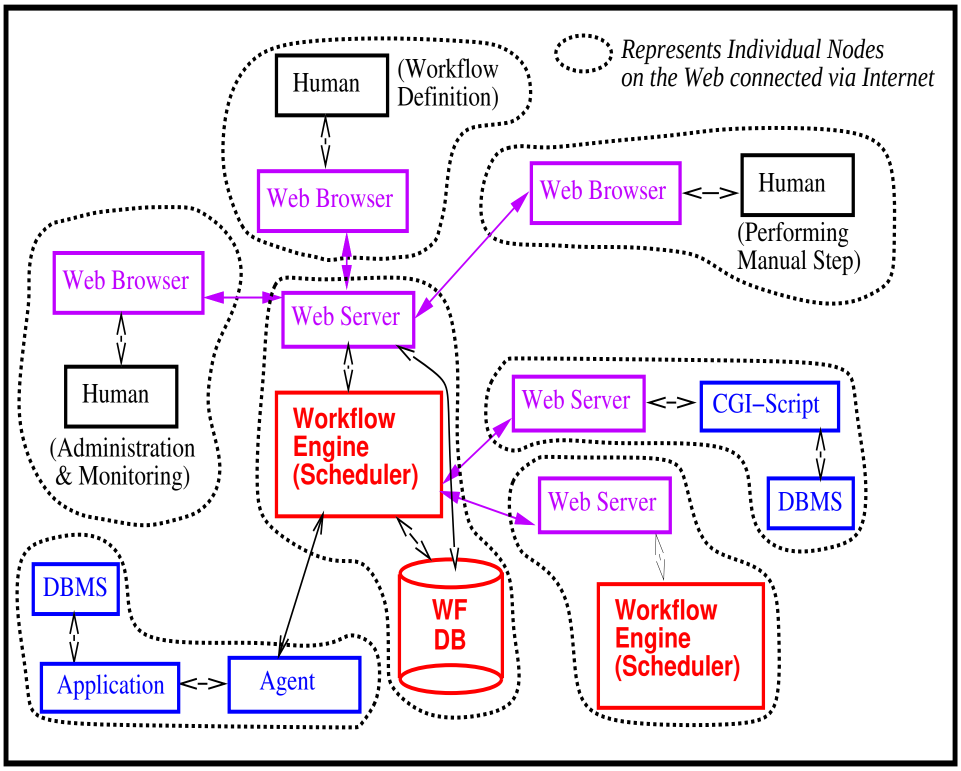

Figure 7.1 Workflow on the Web

executed on the Web, the Web servers play the role of agents. The CGI-scripts at the Web servers are the programs executed for performing an automated step. Many of these scripts execute *transactions* against a backend database. Thus, humans communicate with the workflow engine through a browser and the workflow engine communicates with the Web servers directly using the HTTP protocol. Essentially the workflow engine becomes a client of the Web servers. Figure 7.1 portrays the components of a Web WFMS. This chapter will focus mainly on the interaction between the workflow engine and the Web servers, *i.e.*, business-to-business transactions. To handle requests idempotently in Web WFMSs, the EONS (Exactly ONce Semantics) protocol described in Figure 3.8 is used.

7.2 The WorldFlow System Architecture

The architecture that has been developed for the WorldFlow system is shown in Figure 7.2. It uses a centralized architecture since distributed workflow control is difficult to achieve in a system where the agents are actually autonomous Web servers. Other than the standard WFMS components like the workflow database, the workflow engine and the modeling tool, it also contains a Web module. The figure shows the interactions between the components of the WorldFlow system. As in the case of traditional WFMSs, the workflow database (WFDB) is used to store and retrieve the workflow schemas and the state data of workflows in progress. The modeling tool facilitates the description of the steps that comprise the workflow. The workflow compiler checks the schema for semantic validity during which it verifies (in conjunction with the Web module) that the URL specified for each steps is valid. The resulting workflow schema is then stored in the workflow database.

The workflow engine is responsible for navigating the workflow instances. For example consider a workflow for handling a customer order. When a customer places an order for an item (or a set of items), based on the specifics of the order, the workflow engine refers to the workflow database and instantiates the appropriate workflow. The input information provided by the customer is passed on as input parameters to the workflow instance. Then, the workflow engine executes each step by passing the required information to the Web module. The Web module will in turn use the information provided to establish a connection to the required Web server using the HTTP protocol. A request is then sent to the Web server which executes the CGI script (this is when the step is actually performed) and returns the reply as a HTML page. The Web modules then fetches the HTML page, extracts the required fields, and passes the information to the workflow engine. Thus, the Web module is responsible for performing the transformation on the input arguments to the Web server, making the HTTP request to the remote Web server, receiving the

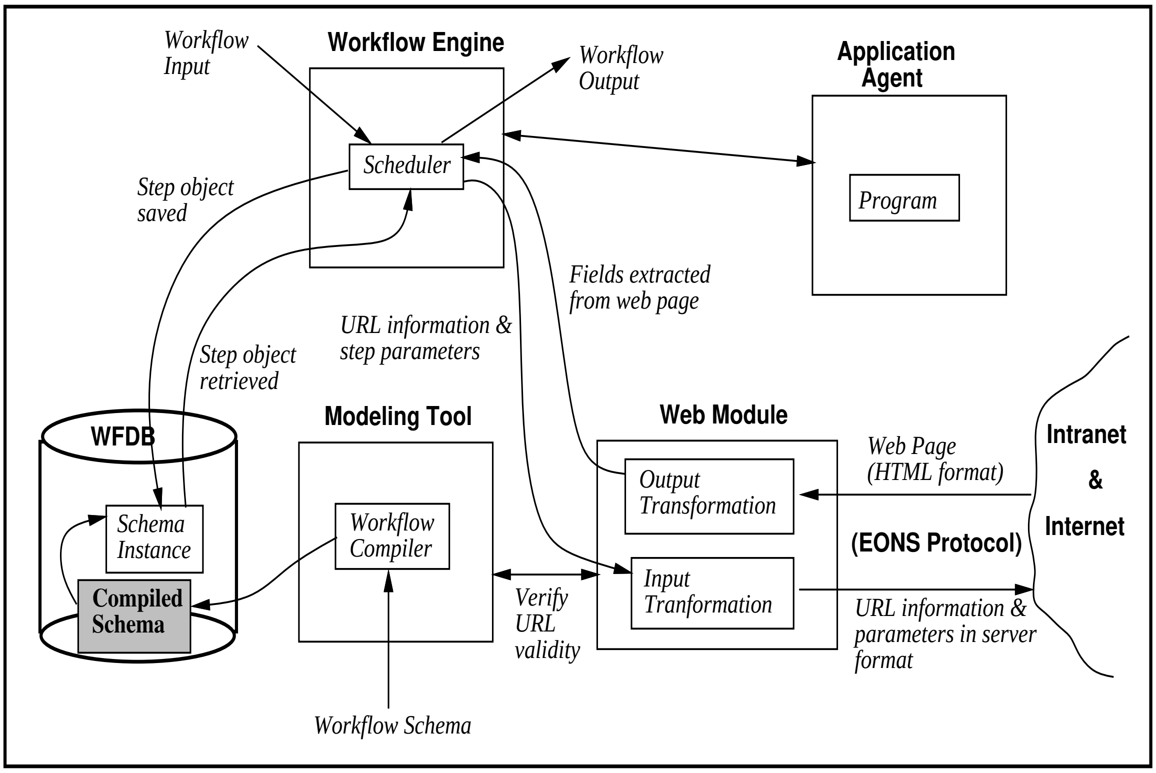

Figure 7.2 WorldFlow Components

HTML page returned by the Web server, and processing it to extract the required fields. The inputs and the outputs are processed by input and output transformation functions. The workflow engine commits the results of the step in the workflow database and refers to the workflow schema to perform the next step using the same Web module. Thus, with the help of the Web module, the workflow engine supervises the execution of the individual steps till the workflow is complete. The Web module and the Web server use the EONS protocol to ensure that steps are executed idempotently. The Web servers themselves are not modified to maintain persistent information. The unique ID for every request is stored persistently in the front end database connected to the Web server.

To perform the individual steps, the WorldFlow system uses Java abstract interfaces. Thus, all programs needed to execute a step are classes that implement the abstract interfaces. The interfaces are as follows:

```
public interface AppFunction{
  public String Execute(String input);
}
e.g., public class Competitors implements AppFunction{...}

public interface WebFunction{
  public String MapInput(String input);
  public String MapOutput(DataInputStream din, String input)
    throws IOException;
}
e.g., public class Quote implements WebFunction{...}
```

The `AppFunction` abstract interface is defined with an `Execute` method for invoking application programs. All application programs are classes that implement this interface with a `Execute` method defined for each. The `WebFunction` abstract interface is defined with two methods `MapInput` and `MapOutput` for invoking Web functions that require input and output transformations respectively. All Web functions are classes that implement this interface with `MapInput` and `MapOutput` methods defined for each. In the next ensuing sections two applications that can be developed with the WorldFlow system are presented.

Since the HTTP protocol is a synchronous protocol it times out when it does not get a response from a remote Web server it times out. In certain cases of business to business transactions, step execution requests can in turn invoke a workflow. This can cause problems because the step duration itself can be longer than the timeout duration specified in the HTTP protocol. This results in the step itself being of long duration. In such a scenario the synchronous HTTP protocol can be problematic. Hence, a WFMS designed for the Web should also have provision for asynchronous step invocation and completion as shown in *figure 7.3.* It works

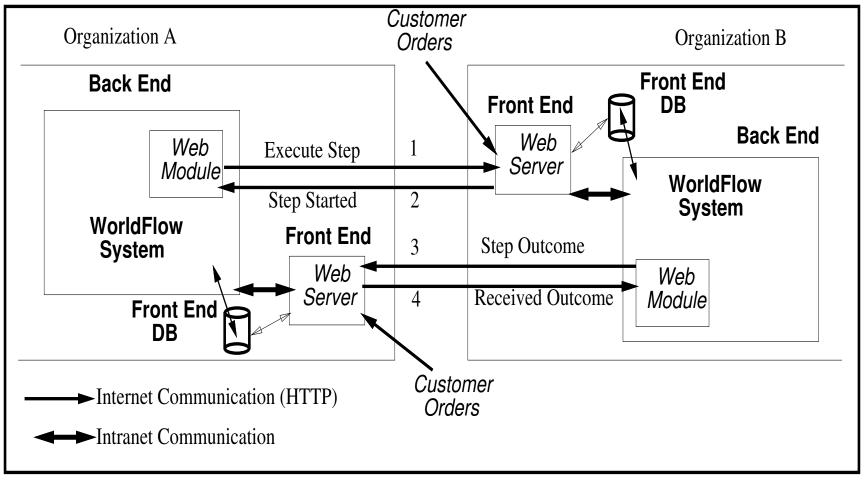

Figure 7.3 Asynchronous Step Execution in Web WFMSs

as follows: (we append alphabets A and B to the respective components from the two organizations). The Web module of the WorldFlow system A places a step execution request with the Web server B (front end) by invoking the "step execute" interface. If there is a failure while making the step execution request, the EONS protocol is used. The front end B in turn invokes a workflow with the WorldFlow system B (back end) and sends an acknowledgment to the Web module A. After the workflow completes, the WorldFlow system B uses the Web module B to invoke the "step outcome" interface of the Web server A to communicate the outcome of the step. Web server A then responds with an acknowledgment thus completing an asynchronous step execution cycle. Note that communication between the Web modules and the Web servers take place using the synchronous HTTP protocol.

7.3 WorldFlow Application — Virtual Enterprises

Consider a network of companies that have agreed to supply goods and services to each other. Using the WorldFlow system they can form a virtual enterprise by enabling steps from order processing workflows (example shown in figure 7.4) to update data from each other through their Web interfaces (for placing and cancelling

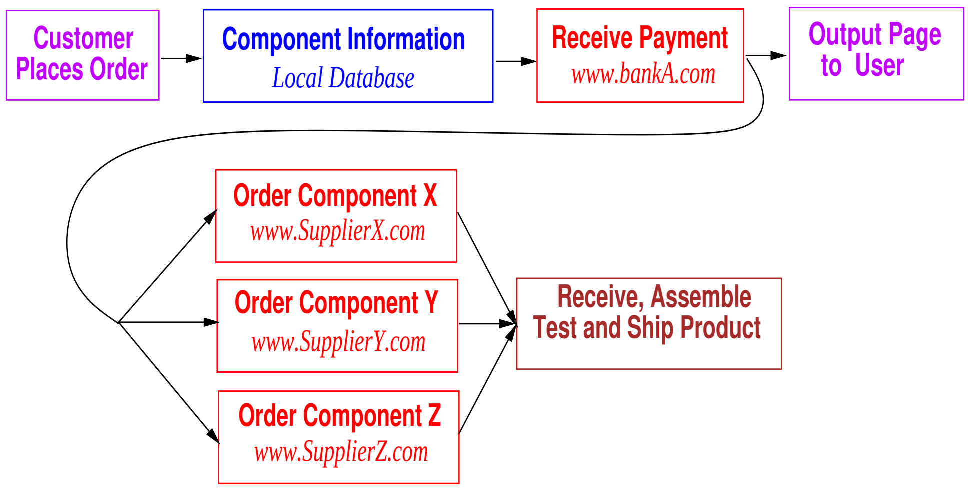

Figure 7.4 Schema for Virtual Enterprise Order Processing

orders). Failure handling techniques and the EONS protocol we discussed in this thesis will be useful in reliably executing such workflows. Such an infrastructure can be much cheaper than using EDI over proprietary networks.

In the following example it is assumed that the following interfaces are provided by the organizations (Web servers) for others to access their services.

- Place-Order (Part No., Quantity, P.O. No.)
- Enquire-Order Status (P.O. No.)
- Enquire-And-Place-Order (Part No., Quantity, P.O. No.)
- Cancel-Order (P.O. No.)
- Enquire-Price (Part No.)
- Get-Payment (Amount, Bank)
- Order-Complete (P.O. No.)

`Place-Order()` and **`Enquire-And-Place-Order ()`** are of interest in the context of the EONS protocol. The semantics of the latter is that an order with a certain P.O. No. is placed only if has not been placed earlier. The use of the EONS protocol is illustrated in the example shown in figure 7.5.

Create order information and make it persistent
Place-Order (Part No., Quantity, P.O. No.)
No response received within timeout period
Workflow Engine failure
...
Workflow Engine recovers
Retrieve order information from persistent store
Enquire-And-Place-Order (Part No., Quantity, P.O. No.)
Confirmation received
Change step state to "completed" in persistent store

Figure 7.5 Failure scenario in a workflow handled by WorldFlow

For asynchronous execution of steps, the `Place-Order()` interface invoked by the Web module at a remote Web server responds with an acknowledgment indicating the request will be processed. In that case, the actual outcome of the step will not be received by the Web module but by the Web server in the form of a call to its `Order-Complete ()` interface by the remote Web module.

The infrastructure that has been developed can be used as middleware for building global data warehouses. Loading the data, indexing the data and summarizing the data can be considered as steps of a workflow. Using the failure handling support provide in the CREW approach, the workflow schema can be suitably designed such that if any of the activities fail then a suitable rollback and re-execution can be performed. Additional details can be found in [79].

7.4 WorldFlow Application — StockAdvisor

Information available on the Internet and the World Wide Web can be viewed as global virtual databases. When one is interested in data pertaining to a specific domain, e.g., stock related information, these form virtual data warehouses. Given the autonomy of many of these data sources as well as the temporal nature of some of the data, it may not be possible to materialize these warehouses. Hence depending on the specific nature of a query, the system must be able to develop a plan by which

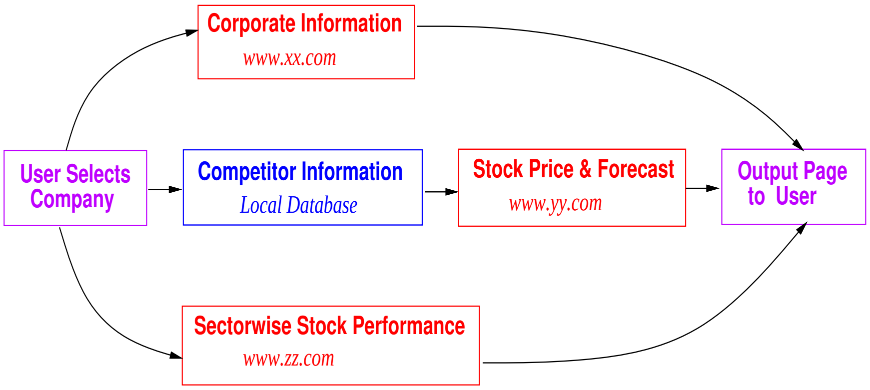

Figure 7.6 Workflow Schema for StockAdvisor Application

to access the different components of the virtual warehouse and provide a timely response. Specifically, the process of gathering data from distributed sources has to be done efficiently and correctly. The WorldFlow system can be used to achieve this objective.

We have built an application called 'StockAdvisor' using the WorldFlow system to demonstrate some of its capabilities. StockAdvisor is a Web based investment decision tool. Once a user selects a company of interest he/she is provided with (i) corporate information about the company (ii) stock performance data of the company and its major competitors and (iii) stock performance over the past year of all the different sectors of the economy. To obtain the same information, an investor has to visit several different sites assuming that the investor has the knowledge about which sites to visit. StockAdvisor uses data stored in local databases as well as data that is fetched on the fly from various Web based information sources. When a user chooses a company name, its ticker symbol and the ticker symbols of its competitors is fetched from the local database. These ticker symbols are then used to dynamically fetch the latest stock price information from Web based stock quote services. The WorldFlow systems helps automate the process of bringing the necessary data to provide the specific information needed by a user. The schema for StockAdvisor is shown in *figure 7.6*. Since the workflows themselves are executed in a reliable manner in the WorldFlow system, if a Web server is not reachable, data can be fetched from an alternative source or by repeatedly retrying with the original server.

Workflows can be used in data warehousing environments not only to fetch data from different information sources, but also to integrate applications that filter, build indexes and collate data loaded into a data warehouse. Thus, the WorldFlow system can be used as a middleware tool for reliably loading information into a data warehouse. Additional details can be found in [107].

7.5 Summary

In this chapter extensions were discussed to the CREW run time system to implement the WorldFlow system, a Web based WFMS. WorldFlow can be used to build workflows whose steps need to directly update the online information available on the Web. The general requirements of a Web WFMS were first studied and then the architecture of WorldFlow was described. Two illustrative applications that can be developed using the WorldFlow system were discussed including how the applications use the various features such as the EONS protocol for achieving idempotency of steps. The need for reliable asynchronous communication between WFMS clients and servers in the context of the Web has been motivated in [59]. The WorldFlow system provides support to precisely handle these requirements.

CHAPTER 8

Related Work

The workflow management research community has been focusing on improving the state-of-the-art by addressing problems in failure handling and decentralized scheduling of workflows. The database community has been developing relaxed concurrency control techniques and extended transaction processing models to handle long duration applications such as workflows, focusing on modeling and correctness. Researchers in software engineering have also been addressing similar issues in the context of process modeling languages. In this chapter we will compare our work to the efforts in these areas related to the thesis. Since CREW uses a rule-based run time system, this work will also be compared with the work on research in active databases.

8.1 Coordinated Execution of Workflows

Since the traditional transaction model is too strict for advanced database applications that are of long duration, several extended transaction models [55, 102, 53, 44, 48] have been proposed. Sagas [55] was one of the first proposals to integrate several transactional steps into a large application. Later several extended transaction models were proposed [48] and formalisms/languages/environments developed to realize extended transaction models. Notably, work on long running activities [43], ConTracts [127, 110], rule driven transaction management [81], dependency management [62], inter-task (inter-step) dependencies [37, 18], customized transaction models (TSME) [57, 58], modular implementation of extended transaction models

[20], durable scripts (DSDT) [115] have all focussed primarily on coordinating sub-transactions of a transaction based on the significant events like *begin, commit, abort* of sub-transactions. All these tools allow a long transaction (application) to be split into smaller sub-transactions. Thus coordinated execution is only performed at the level of the sub-transactions within a transaction. In an abstract sense, the coordinated execution requirements involve two levels of synchronization one at the workflow level and the other at the database level (by DBMS concurrency control). However, extended transactions models like nested transactions [99] and multi-level transactions [128] cannot be used to handle the coordinated execution requirements for the reason that the synchronization is performed on different autonomous systems. Mutual exclusion need not be additionally enforced since all objects accessed by the sub-transactions are under the control of resource managers that perform concurrency control. Controlled interactions between transactions is achieved by different techniques. For example, ConTracts deals with issues related to synchronization across long computations using the notion of *invariants*. Invariants can be used to synchronize access to object based on their value but they cannot be used to perform coordinated execution requirements such as relative ordering. TSME allows the specification of breakpoints in transactions which allows for interleaving accesses to objects by other transactions. Breakpoint based techniques have been suggested even earlier in the context of concurrency control techniques in database systems [50] and are not useful for coordinated execution requirements like relative ordering. ConTracts provide a superficial discussion on how rollback dependency type of requirements can be handled if spheres of control [41] are available in a system. However, it has been pointed out that spheres of control are very difficult to implement [61]. The solution for rollback dependency is simpler since it keeps track of only specified dependencies between steps from concurrent workflows.

Most business processes integrate independently developed applications that access non-database objects as well. Several commercial and prototype WFMSs have been developed [65, 66, 56] that provide good modeling facilities for handling data and control flow requirements within a workflow. Their modeling capabilities are powerful enough even to capture semantics of some extended transaction models [11]. However, they do not offer facilities for controlling the interleaving of steps from different tasks.

Although multidatabase applications access data distributed over multiple nodes, the individual accesses by a transaction are coordinated such that atomicity is achieved across all sites. New correctness criteria [46] and schemes for concurrency control, recovery and transaction management [94, 27] have been proposed. In particular, serializability with respect to compensation is specified as a correctness criterion for multidatabase transactions [82] which prevents transactions from viewing partial changes at some of the sites in the event of a rollback. In contrast, the requirements of rollback dependency are different — rollbacks of workflow are tracked only if needed and action is required only in affected workflow instances.

The need for mutual exclusion of steps from concurrent workflow processes to ensure correct interleavings has also been discussed in [10]. In this work a protocol is proposed that uses object and process 'signatures' (access histories) to dynamically impose access restrictions thus supporting mutual exclusion of steps from concurrent workflows. In contrast, in the CREW approach, conflict conditions for mutual exclusion requirements are established at compile time and evaluated at run-time based on step and workflow parameters. Additionally, in this thesis, the focus has been on providing language and run-time support to handle mutual exclusion requirements for different workflow control architectures. To achieve more sophisticated control on interleaving of steps from concurrent workflows, the notion of relative ordering has been proposed which ensures that conflicting steps from concurrent workflows are executed in a serializable manner. History information is maintained about the ordering of first pair of conflicting steps and subsequent conflicting steps are executed in the same order.

8.2 Failure and Exception Handling of Workflows

An extended transaction based approach to failure handling in workflows is suggested using the WAMO model in [47]. They have developed a workflow specification called WADL, which allows the specification of dependencies using the ACTA formalism [37]. Tasks specified using WADL are then executed on top of an active database system. Most of their focus is on compensation of activities in the case of task (workflow) nesting. In addition to the structure of the workflow, designers also specify additional properties about the activities such as (i) the manner in which an activity can be compensated and (ii) the criticality of the activity. In the WAMO model, an activity is critical if it cannot be compensated once it has been executed and a task is considered unsafe if it requires the compensation of a critical activity. The notion of a critical activity is very similar to the notion of *pivotal* sub-transactions in the flex transaction model discussed in [132]. The only similarity between the WAMO model and the CREW model is that workflow designers have to specify additional properties about the steps and how they have to be compensated. In terms of the modeling power for failure handling, the notion of compensation dependent sets and the opportunistic compensation and re-execution strategy offer much more flexibility is specifying failure handling requirements of workflows. Another significant difference is that the WAMO model is an extended transaction based approach requiring all the tasks to be executed on a single database system, whereas the CREW approach is geared towards the integration of heterogeneous applications designed to run on autonomous systems. To realize the WAMO model for real world applications, the autonomy of the systems have to be sacrificed so that the workflow (scheduler) database and the applications databases can use a two-phase protocol to enforce dependencies such as begin-on-commit and commit-on-abort.

Failure handling of steps in workflows is discussed in [86] using the notion of spheres of joint compensation. On the failure of a step, all other steps that belong to the sphere which contains the failed step are rolled back. Although this technique provides a precise way of handling failures via partial rollback of a workflow, it does not deal with the interactions between rollback and forward execution of other steps in a workflow. In contrast, the opportunistic compensation and re-execution technique is an integrated technique that considers the interaction between forward execution and rollbacks in workflows.

Handling logical failures is analogous to exception handling. There has been a great deal of work on exception handling in the area of programming languages [88, 26] and software fault tolerance [49]. In programming languages, depending on the type of exception, control is transferred to an appropriate statement or routine. In fault tolerant systems, as soon as faults are detected, control is transferred to exception handling algorithms which correct the errors and perform backward or forward recovery as required. In contrast, the failure handling techniques consider the interaction between forward execution and rollbacks of workflows.

A three layer error model for exception handling in the METEOR system is presented in [131]. The types of errors they address include (i) errors in the infrastructure external to the WFMS, (ii) errors in the WFMS itself such as failure of the scheduler and (iii) application and user errors. Their three layer error model they use is based on the enactment components in the METEOR workflow model — task (step/application), task managers (agents) and the workflow engine. When an error occurs, the error type is labeled based on the layer at which the error occurs. The task errors are the same as the logical failures referred to in this thesis. Task manager errors and workflow engine errors correspond to the system failures. They provide an ORB (Object Request Broker) based hierarchical recovery framework consisting of a local recovery unit to handle logical errors at the task manager level and a global recovery unit to handle errors at the workflow engine level. In contrast, the focus has been on generic problems that can occur at the workflow (application) level. We discussed the effect of a step failure on other concurrent steps within the same workflow and across workflows. This work has developed language support and system support for failure handling and coordinated execution under different workflow control architectures. In the CREW system, the workflow designer does not have to consider workflow scheduler configurations into account since the LAWS specification is architecturally neutral (does not depend on the type of the architecture). Thus, the solutions developed by us in this thesis and in [131] complement each other. By packaging them together flexible and reliable solutions can be developed for failure handling.

Since support for handling system failures of workflow database [72] and workflow engines [14] has been investigated earlier, this work focused on failure of the workflow agent not just in traditional WFMS but also in the context of Web WFMSs. Thus our work complements earlier work on building robust and reliable WFMSs.

The ARIES recovery method for transaction processing systems [98] reduces the amount of wasted work by preventing repeated compensation of already compensated actions during restart recovery after a failure. Although the opportunistic compensation and re-execution scheme discussed in this thesis is quite different from that of ARIES, it uses the same philosophy of reducing the amount of wasted work.

8.3 Non-Centralized Control of Workflows

The need for non centralized control of workflows has been recognized earlier in systems like INCAS [19] and Exotica/FMQM [13]. In both these pieces of work the primary focus is on supporting distributed scheduling of workflows. In addition to scheduling support for normal execution, this thesis discusses support for handling logical failures and coordinated execution. This work has explored both parallel and distributed workflow control architectures focusing on techniques for enforcing correctness and maintaining scalability. Recall that in a parallel workflow control architecture the state of an individual workflow is on a single node while the global state is distributed across nodes. Hence, solutions for coordinated execution require exchange of state and event information across nodes. In the case of distributed workflow control since the state of an individual workflow is distributed across nodes, exchange of state and event information is needed not only for coordinated execution but also for handling failures of steps within a workflow. Techniques used in the CREW approach ensure that the data and state information exchange is done correctly and as efficiently as possible. Thus the work presented here significantly extends previous research efforts in building correct and scalable non-centralized WFMSs.

Recent work on distributed constraints and scheduling of workflow computations [122, 121] is closely related to the objective of CREW. Starting with a workflow specification as a language based on temporal logic it shows how to derive preconditions for events. This work is driven by pragmatic considerations and has taken the form of developing specification-level building block, implementation-level primitives, translation from the former to the latter as well as a realization of this necessary infrastructure.

8.4 Workflow Specification Language and Compiler

Languages for specifying extended transaction models provide primitives for performing specific actions on events like the start, commit and abort of a subtransaction [62, 57, 115, 25]. These events can be tied to the occurrence of similar events of other sub-transactions in the same transaction. Thus coordinated execution is achieved between sub-transactions of a transaction. Using primitives that allows the specification of breakpoints, controlled interleaving of conflicting actions between transactions has been supported. Both in the context of transactions and workflows, a compatibility matrix based approach has been used to support handling of conflicting actions by exploiting their semantics. Such an approach is possible in the context of this work, however it would require a workflow designer to explicitly specify a large matrix that grows polynomially based on the number of workflows and the number of steps in each workflow. In contrast, a different approach has been taken in this work where the workflow designer specifies the resources accessed by the individual steps of the workflows and a data dependency compiler automatically figures out the coordinated execution requirements with minimal efforts from the workflow designer. This allows a workflow designer to focus on the details of the individual steps rather than steps from other workflows that might potentially conflict. Workflow modeling languages typically provide primitives for defining workflow structures (steps, control and data flow) and staff organization. The focus of Workflow modeling research is on improving modeling power of to capture the requirements of complicated business processes. To complement this, the workflow modeling primitives allow a workflow designer to customize correctness/performance requirements of workflows.

8.5 Rule-Based Run Time System

Rule based approaches have been used earlier to manage workflow and transaction execution [81, 44, 29]. Rules are defined statically and fired dynamically to schedule the sub-transactions (or steps) in accordance with the structure of the transaction (or workflows). In contrast, the statically defined rules in the system just form a template. The rules are first instantiated for every workflow instance and then modified dynamically at run time (*e.g.*, changing event or condition or action part of a rule) to handle coordinated execution and failure handling requirements apart from handling step execution in accordance with the workflow schemas.

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. Hence, it is necessary to have efficient algorithms for the detection of composite events as discussed in the context of the Snoop language for event specification [33]. Although the events in the rules are not primitive, they are not very complex to cause inefficient execution. In addition indexes are also maintained 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 that has been developed for scheduling steps is very efficient.

Work on mechanisms for distributed active database systems [31] is relevant to the techniques used in the CREW run time system. There are certain similarities in issues like the existence of multisite rules. However, the type of application addressed are quite different. In [31] rules at one site trigger data operations on other sites. Ordering of locally and remotely triggered rules is optional and even if it required the priorities between the rules are specified statically. In the CREW environment, often conditions based on data values and event timestamps are used to determine the order of rule firing at the different sites.

8.6 Web/Internet Based Workflows

The METEOR project [119] has implemented workflows on the Web. In this approach, a step typically consists of a form that is filled in and processed by a CGI script which in turn generates a form with the URL of the CGI script that should perform the next step of the workflow. This process is repeated to complete the workflow. Thus, there is no central workflow engine/scheduler to control and manage the workflow. The workflow schema is essentially embedded in the different CGI scripts which are responsible for performing the control flow. Reliability issues have not been discussed extensively for this architecture. Specifically, in the event of a failure, step execution may not be idempotent. Although the task managers (agents) are designed to handle HTTP error codes, the error code will not be helpful in indicating whether a step has been executed or not. Since workflows cross borders of autonomous organizations, it is difficult to ensure progress and trackability. The WorldFlow system on the other hand is centralized and has support for executing steps in an reliable asynchronous manner. Since the process states are maintained by the WorldFlow system it is easier to ensure progress and provide workflow execution tracking. Alternatively, a distributed workflow control architecture can be used in conjunction with the EONS protocol for implementing WorldFlow to integrate non-autonomous organizations since tracking and ensuring workflow progress (as discussed in section 6.4) is feasible.

Workflow support for collaborative design on the internet is discussed in [85]. Their objective is to support workflow reconfiguration, synchronous and asynchronous peer-to-peer and peer-to-workflow communication to support collaboration. They do not discuss support for failure handling. In contrast, our work focuses on workflow support for business-to-business transactions with emphasis on proper handling of steps failures and network failures.

8.7 Software Process Modeling

Software processes are comparable to workflow processes since they have activity (step) descriptions, artifact descriptions (code that is manipulated), resource requirements (both human and computer). Software processes also have precedence relationship between the activities and artifacts which is comparable to the definition of the workflow schema. In addition they have consistency requirements to govern the interactions between activities, artifacts and resources. A number of languages and object systems such as Adele citeAdele, APPL/A [67], EPOS [39], HFSP [123], Process Weaver [51], JIL [68], LOTOS [113], Merlin [70], Marvel [71], Pleiades [124], have been developed to handle the requirements of process modeling. These differ in their abstractions, semantic richness, composability (similar to workflow nesting) and so forth. A comparison of these can be found in [69] and [68]. Process modeling languages focus to a large extent on the evolution of a software process, version/concurrency control of the artifacts, and undo/redo of activities. In contrast, (i) the structure of a workflow process does not often change over the lifetime of a *workflow instance*, (ii) steps often access data from DBMSs which have their own concurrency control and (iii) the number of instances of a workflow schema is very high. This work has focused mainly on workflows whose schemas are known in advance (orderly progression of steps) but have complex failure handling and coordinated execution requirements. The opportunistic compensation and re-execution technique developed in this work is a very powerful and flexible scheme for handling undo/redo of steps in comparison with the interrupt/exception handling support provided by some of the reactive process modeling languages. Similarly in the case of coordinated execution, the relative ordering and rollback dependency primitives of LAWS offer much more flexibility than concurrency/version control support available in process modeling languages. Process modeling languages on the other hand offer more flexibility in constructing evolving processes and this is an important direction in which LAWS (and for that matter any other workflow language) can be extended. Thus the work presented here complements research in process modeling.

8.8 Summary

This chapter compared the CREW approach to other approaches that have been suggested in the context of both extended transaction models and WFMSs (commercial and prototype). The significant difference between the CREW approach and the extended transaction model based approaches is that the CTEW approach considers a workflow to be an integration of previously developed autonomous applications instead of assuming the various applications to run on a single database system. Most real world workflows fall into former category. Hence the support that has been developed for failure handling and coordinated execution is external to the application databases. In comparison with state-of-the-art commercial and research WFMSs, the CREW systems offers significant flexibility in terms of failure handling and coordinated execution of workflows. With the additional capabilities provided by LAWS and the CREW run-time systems it is now possible to achieve better correctness on workflow execution. As pointed out in *figure 3.1* in chapter 3, the CREW approach (a form of transactional workflow approach) is a carefully developed blend of concepts from extended transaction models and state-of-the-art WFMSs that offers better flexibility and correctness in handling practical workflow applications.

Solutions for some of the workflow subproblems in coordinated execution (such as mutual exclusion) are available in the literature [17]. The low level mechanisms have been chosen based on the need for developing uniform support for the various coordinated execution requirements in the context of rule based execution. Since there is support for exchanging event information across nodes, the CREW system is extensible.

Our work complements the contributions from the process modeling research community and hence could be used for process modeling. We also see benefits in adopting some of the concepts from process modeling to provide flexibility in modeling workflow schemas. The WorldFlow system has extended the capabilities of web WFMSs by adding support for reliable web workflows that can even consist of asynchronous steps.

CHAPTER 9

Summary and Future Work

9.1 Summary

State-of-the-art WFMSs do not have adequate support for handling data inconsistencies caused by (i) dependencies between steps that access shared resources, (ii) failure of steps in a workflow, and (iii) certain types of system failures. This is a major drawback, especially since WFMSs are to be used in banks, insurance companies and healthcare organizations where data inconsistencies can lead to undesirable consequences. Also, since most WFMSs use a centralized workflow control architecture they have scalability problems especially in large enterprises where several thousand workflows can be in execution concurrently. This thesis presents solutions to handle all of the above problems.

The key conceptual contributions of this thesis are as follows:

- To handle dependencies between steps across workflows, techniques have been developed for handling coordinated execution requirements such as *mutual-exclusion*, *relative ordering* and *rollback dependency* across workflow instances. These techniques have been developed to directly address the various problems encountered during concurrent execution of workflows. Since a WFMS is not aware of the resources accessed by the applications, the dependencies themselves are identified with the aid of a global resource database that is populated with the information provided by the workflow designer.

- To handle step failures in workflows, an integrated approach has been proposed to handle partial rollback and subsequent re-execution of workflows. This integrated approach uses an *opportunistic compensation and re-execution strategy* that minimizes the amount of lost work. This strategy compensates and re-executes a step *if* and *when* needed. In addition, the use of *compensation dependent sets* is proposed to customize failure handling requirements of a workflow. These techniques can be used to achieve better correctness and performance at the workflow level. Utilizing these techniques solutions have been proposed to handle race conditions in workflows which are caused due to the presence of step failures.

- To handle failure of the workflow agent, logging based solutions and idempotency protocols have been developed. Every workflow agent logs significant events, *e.g*, when a program completes, the return status code and data produced by the program. Using such a mechanism, after an agent fails and recovers it can pass on the results to the workflow engine without causing inconsistencies in the state information about the step. To handle idempotency of steps in workflows, a simple *exactly once semantics* protocol has been designed. In this protocol, the WFMS generates a unique ID for every remote Web access by a step and stores this ID persistently. The WFMS uses this unique ID and a set of standard interfaces at the remote servers to achieve idempotency even in the presence of failures.

- To address the scalability issue *two alternative workflow control architectures has been considered — parallel and distributed.* In a parallel architecture several workflow engines perform the navigation of workflow instances in parallel (still communicating with the same set of agents as in a centralized architecture). In a distributed architecture the navigation is performed by the agents themselves without the use of a workflow engine. In both these architectures, steps from concurrent workflows each of which is managed by a different workflow engine/agent can conflict. The protocols necessary for parallel and distributed workflow control have also been developed. They use piggybacking and other techniques that exploit available information whenever possible to *reduce the number of additional messages exchanged.*

The key infrastructural contributions of this thesis are as follows:

•We have developed LAWS, a *workflow specification language* that has primitives for specifying traditional workflow requirements as well as enhanced requirements for failure handling and coordinated execution. A *two stage compiler* has been developed to process workflow schemas specified in the workflow language. The first stage extracts data dependencies using a global resource database and creates an enhanced intermediate schema. 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 second stage of the compiler processes the intermediate schema and generates appropriate rules. Another characteristics of LAWS is that the specification is architecturally neutral (does not depend on the type of the architecture). Hence, if an organization decides to switch to a different workflow control architecture under the CREW environment, the existing schema definitions can be used without any modifications.

•We have also designed a rule-based run-time system for central, parallel and distributed workflow control architectures. It interprets the rules at run-time and ensure that the workflows are executed in accordance with the specification. A small set of *implementation level primitives* (that dynamically modify the rules) have been identified, sufficient enough to realize the high level coordinated execution. Mechanisms necessary to enforce failure handling requirements have also been developed. These mechanisms have been designed for centralized, parallel and distributed workflow control architectures. Using LITES, a simple notation we have also proved the *correctness* of the low level implementation mechanisms.

- To demonstrate practicality of the CREW approach, a prototype WFMS has been implemented to realize a subset of the techniques developed. The prototype has been built using the Java programming language for achieving portability across heterogeneous architectures. This prototype includes a workflow compiler and a rule-based run-time system for central, parallel and distributed workflow control architectures. Applications have been simulated with different failure probabilities and obtained traces to verify that the executions are in accordance with the schema specification. The prototype has been extended to implement WorldFlow, a system for reliably handling Web workflows.

While current WFMSs offer 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. This work is an important step in this direction. Based on the concepts and techniques developed in this thesis, WFMSs can be enhanced for use in more critical environments where correctness and performance are major concerns. The prototype system for example can offer better correctness, performance and functionality than state-of-the-art WFMSs. The significant difference between the CREW approach and the extended transaction model based approaches is that the CREW approach considers a workflow to be an integration of previously developed autonomous applications instead of assuming the various applications to run on a single database system. Most real world workflows fall into former category. Hence the support developed for failure handling and coordinated execution is external to the application databases. In comparison with state-of-the-art commercial and research WFMSs, the CREW systems offers significant flexibility in terms of failure handling and coordinated execution of workflows. As pointed out in *figure 3.1* in chapter 3, the CREW approach (a form of transactional workflow approach) is a carefully developed blend of concepts from extended transaction models and state-of-the-art WFMSs that offers better flexibility and correctness in handling practical workflow applications.

9.2 Directions for Future Work

A number of extensions are possible to this work. We envision the following ways in which the ideas developed in this thesis can be enhanced.

- In this work a static definition for workflows is assumed. However, in several application scenarios workflow definitions might have to be changed on a per instance basis. To address correctness and performance issues of instances, the techniques presented here need some enhancements. In particular, conflicts and dependencies determined by the data dependency compiler for the base (original) schema may not be valid for the modified instance. Hence, some of the capabilities of the data dependency compiler can be incorporated into the run-time environment so that conflict, dependencies and even compensation dependent sets can be determined dynamically based on instance specific information.

- The CREW techniques and prototype system can be enhanced to support cooperative activities that requires workflow support. Techniques developed in the context of software process modeling/enactment and cooperative transactions will be useful in enhancing the coordinated execution techniques of the system. Although the failure handling techniques may require enhancements they will be very useful in handling rollback in cooperative activities, *e.g.*, if the rollback and re-design of one component does not affect the other components then the other components need not be re-designed.

- The CREW techniques for coordinated execution and failure handling will need enhancements to handle system failures of the workflow engine or the workflow database. The solution would require workflow instances to migrate to a different workflow engine in the event of a failure of the workflow engine or the workflow database. When instances migrate from one engine to another, coordinated execution becomes complicated. Messages related to the coordinated execution of the migrated instances have to be re-routed to the new engines that control the migrated instances.

- As electronic commerce gets popular, inter-organizational workflows will be more common. Managing workflows that span organizations is complicated. In particular, there may be a need for transferring control of whole or part of a workflow instance from the control of one workflow engine to another workflow engine. Handling step failures in such inter-organizational workflows is also more complicated and may require the development of additional mechanisms.

APPENDIX

GRAMMAR FOR LAWS

The grammar for LAWS consists of the following production rules.

TOP LEVEL RULE FOR WORKFLOW SCHEMA:

(WorkflowSchema) $\longrightarrow$ **(WorkflowSpec) (StepSpec) (DataFlowSpec) (ControlFlowSpec) [(FailureHandlingSpec) (WorkflowResourceSpec) (WorkflowConflictSpec)] (WorkflowCoordExecSpec)]**

RULES FOR WORKFLOW INPUT/OUPUT SPECIFICATION:

(WorkflowSpec) $\longrightarrow$ (WorkflowName) : (Workflow.Inputs) : (Workflow.Outputs)
(Workflow.Inputs) $\longrightarrow$ (InputName)$^+$
(Workflow.Outputs) $\longrightarrow$ (OutputName)$^+$
(InputName) $\longrightarrow$ I(Integer)
(OutputName) $\longrightarrow$ O(Integer)

RULES FOR STEP SPECIFICATION:

(StepSpec) $\longrightarrow$ ((SimpleStep) | (NestedStep) ;)$^+$
(SimpleStep) $\longrightarrow$ (StepName) : (StepNode) : (ProgramName) (UndoProgName) : (StepInputs) : (StepOutputs)
(NestedStep) $\longrightarrow$ (StepName) : (WorkflowName) (UndoWorkflowName) : (StepInputs) : (StepOutputs)
(StepInputs) $\longrightarrow$ (InputName)$^+$
(StepInputs) $\longrightarrow$ (OutputName)$^+$
(StepNode) $\longrightarrow$ (Identifier)
(ProgramName) $\longrightarrow$ (Identifier)
(UndoProgramName) $\longrightarrow$ (Identifier)
(WorkflowName) $\longrightarrow$ WF(Integer)
(UndoWorkflowName) $\longrightarrow$ WF(Integer)

RULES FOR DATA FLOW SPECIFICATION:

(DataFlowSpec) $\longrightarrow$ ((StepName.InputName) = (StepName.OutputName) | WF.(InputName) ;)$^+$
(StepName) $\longrightarrow$ S(Integer)

RULES FOR CONTROL FLOW SPECIFICATION:

(ControlFlowSpec) $\longrightarrow$ (ControlType ;)$^+$
(ControlType) $\longrightarrow$ (LinearType) | (BranchType) | (ParallelType) | LoopType
(LinearType) $\longrightarrow$ (StepName) : (StepName)
(BranchType) $\longrightarrow$ (StepName) : IF (Condition) THEN (StepName) (ElseIf)* ELSE (StepName)
(ElseIf) $\longrightarrow$ ELSEIF (Condition) THEN (StepName)
(Condition) $\longrightarrow$ (Terms) (ExtraTerm)*
(Terms) $\longrightarrow$ (Term) (AND (Term))*
(Term) $\longrightarrow$ (((StepName.OutputName) | (Workflow.InputName)) (CompareOp) ((StepName.OutputName) | (Workflow.InputName) | Value))
(CompareOp) $\longrightarrow$ (== | $\neq$ | > | $\geq$ | < | $\leq$)
(ExtraTerm) $\longrightarrow$ OR (Terms)

⟨ParallelType⟩	$\longrightarrow$	⟨StepName⟩ : ALL ⟨StepName⟩$^+$
⟨LoopType⟩	$\longrightarrow$	LOOP ⟨StepName⟩ ⟨StepName⟩ ⟨InitStatement⟩ ⟨EndStatement⟩ ⟨ExitStatement⟩
⟨InitStatement⟩	$\longrightarrow$	INIT ⟨Assignment⟩
⟨EndStatement⟩	$\longrightarrow$	END ⟨Assignment⟩
⟨ExitStatement⟩	$\longrightarrow$	EXIT ⟨Condition⟩
⟨Assignment⟩	$\longrightarrow$	⟨Variable⟩ = Expression ((+ | −)Expression)$^+$
⟨Expression⟩	$\longrightarrow$	⟨Literal⟩ ((x | /)Literal)$^+$
⟨Literal⟩	$\longrightarrow$	⟨Variable⟩ | ⟨StepName.OutputName⟩ | ⟨WorkflowName.InputName⟩

RULES FOR FAILURE HANDLING SPECIFICATION:

⟨FailureHandlingSpec⟩	$\longrightarrow$	(⟨RollbackSpec⟩ | ⟨ReExecutionSpec⟩ | ⟨CompensationDependentSetSpec⟩)
⟨RollbackSpec⟩	$\longrightarrow$	(⟨StepName⟩ COMPENSATE ⟨StepName⟩ (RE-START | ABORT) ;)$^+$
⟨ReExecutionSpec⟩	$\longrightarrow$	(IF ⟨InputCondition⟩ THEN ⟨ModifyInput⟩ RE_EXECUTE ELSEIF ⟨InputCondition⟩ THEN ⟨ModifyInput⟩ COMPENSATE ELSEIF ⟨InputCondition⟩ THEN ⟨ModifyInput⟩ COMPENSATE RE_EXECUTE ;)$^+$
⟨InputCondition⟩	$\longrightarrow$	ALWAYS | INPUTS_CHANGED | ⟨InputComparison⟩
⟨InputComparison⟩	$\longrightarrow$	⟨InputTerm⟩ (OR InputTerm⟩)$\star$
⟨InputTerm⟩	$\longrightarrow$	⟨InputCond⟩ (AND InputCond⟩)$\star$
⟨InputCond⟩	$\longrightarrow$	⟨InputVersion⟩ ⟨CompareOp⟩ ⟨InputVersion⟩
⟨InputVersion⟩	$\longrightarrow$	⟨InputName⟩.(NEW | OLD)
⟨ModifyInput⟩	$\longrightarrow$	⟨InputVersion⟩ = ⟨InputVersionExpr⟩ ((+ | −)InputVersionExpr⟩)$\star$
⟨InputVersionExpr⟩	$\longrightarrow$	⟨InputVersion⟩ ((x | /)InputLiteral⟩)$^+$
⟨InputLiteral⟩	$\longrightarrow$	⟨InputVersion⟩ | ⟨Variable⟩ | ⟨Integer⟩
⟨CompensationDependentSetSpec⟩		(CDS⟨Integer⟩ : ⟨StepName⟩ ;)$^+$

RULES FOR WORKFLOW RESOURCE SPECIFICATION:

⟨WorkflowResourceSpec⟩	$\longrightarrow$	⟨StepName⟩ : (⟨ResourceSpec⟩⟨AccessMode⟩ ;)$^+$
⟨ResourceSpec⟩	$\longrightarrow$	⟨ResourceName⟩ / ⟨Qualifier⟩
⟨Qualifer⟩	$\longrightarrow$	⟨InputName⟩ | ⟨Range⟩
⟨Range⟩	$\longrightarrow$	(⟨InputName⟩ - ⟨InputName⟩)
⟨AccessMode⟩	$\longrightarrow$	READ | WRITE

RULES FOR WORKFLOW CONFLICT SPECIFICATION:

⟨WorkflowConflictSpec⟩	$\longrightarrow$	(IF ⟨Condition⟩ ⟨StepName⟩ CONFLICTS_WITH ⟨AnyStep⟩ ;)*
⟨AnyStep⟩	$\longrightarrow$	⟨StepName⟩ | ⟨WorkflowName.StepName⟩

RULES FOR COORDINATED EXECUTION SPECIFICATION:

⟨WorkflowCoordExecSpec⟩	$\longrightarrow$	(⟨MutualExclSpec | ⟨RelativeOrderSpec⟩ | ⟨RollbackDepSpec⟩ ;)*
⟨MutualExclSpec⟩	$\longrightarrow$	IF ⟨StepName⟩ CONFLICTS_WITH ⟨Steps⟩ THEN ⟨StepName⟩ MUTUALLY_EXCLUDE ⟨Steps⟩
⟨Steps⟩	$\longrightarrow$	⟨AnyStep⟩ | ⟨GroupOfSteps⟩
⟨GroupOfSteps⟩	$\longrightarrow$	⟨AnyStep⟩ (AND ⟨AnyStep⟩)$^+$
⟨RelativeOrderSpec⟩	$\longrightarrow$	IF ⟨StepConflict⟩ AND ⟨StepName⟩ RELATIVE_ORDER ⟨AnyStep⟩ AND ⟨StepConflict⟩ THEN ⟨StepName⟩ RELATIVE_ORDER ⟨AnyStep⟩
⟨RollbackDepSpec⟩	$\longrightarrow$	ON ⟨AnyStep⟩.COMPENSATE IF ⟨StepConflict⟩ AND ⟨StepName⟩ HAPPENED_BEFORE ⟨AnyStep⟩ THEN ⟨RollbackAction⟩
⟨RollbackAction⟩	$\longrightarrow$	(COMPENSATE ⟨StepName⟩ RE-START ⟨StepName⟩) | ABORT

BIBLIOGRAPHY

[1] *Action Workflow Metro World-Wide-Workflow*, Action Technologies, Inc. http://www.actiontech.com/metrotour/resources/Metwp.htm.

[2] *ActiveX*, http://www.microsoft.com/activex/default.htm.

[3] *CORBA Overview*, Available at http://www.omg.org/library/corbserv.htm.

[4] *IBM FlowMark*, http://www.torolab.ibm.com/workgroup/flowmark.

[5] *Java IDL*, http://splash.javasoft.com/JavaIDL/pages/index.html.

[6] *JTS API*, http://splash.javasoft.com/jts/jts.html.

[7] *JDBC API*, http://splash.javasoft.com/jdbc/index.html.

[8] *OrbixWeb for Java*, http://www.iona.ie/Orbix/OrbixWeb/index.html.

[9] *Standard for Hypertext Transfer Protocol – HTTP/1.0 and HTTP/1.1*, http://www.ics.uci.edu/pub/ietf/http/.

[10] 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.

[11] 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.

[12] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionalities and limitations of current workflow management systems. *IEEE Expert: Special Issue on Cooperative Information Systems*, 1997.

[13] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, and M. Kamath. Exotica/FMQM:A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proc. of IFIP WG 8.1 Workgroup Conference on Information Systems Development for Decentralized Organizations (ISDO95), Trondheim, Norway*, August 1995.

[14] 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, November 1994.

[15]G. Alonso and C. Mohan. Workflow management systems: The next generation of distributed processing. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures.*

[16]M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 65, Vancouver, BC, Canada, August 1992.

[17]A.S.Tannenbaum. *Distributed Operating Systems.* Prentice-Hall, 1995.

[18]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.

[19]D. Barbara, S. Mehrota, and M. Rusinkiewicz. INCAS: Managing Dynamic Workflows in Distributed Environments. *Journal of Database Management, IDEA Group Publishing.*, 7(1):5–15, December 1996.

[20]Roger Barga and Calton Pu. A Practical and Modular Implementation of Extended Transaction Models. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, 1995.

[21]N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269, September 1991.

[22]P. Bernstein. Transaction Processing Monitors. *Comm. of the ACM*, 33(11):75, November 1990.

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

[24]D. Bertsekas and R. Gallager. *Data Networks.* Prentice Hall, Englewood Cliffs, NJ, 2 edition, 1992.

[25]A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proc. 1994 SIGMOD International Conference on Management of Data*, pages 44–54, May 1994.

[26]Alexander Borgida. Language features for flexible handling of exceptions in information systems. *ACM Transactions on Database Systems*, 10(4), December 1985.

[27]Y. Breitbart, A. Silberschatz, and G. R. Thompson. An approach to recovery management in a multidatabase system. *The VLDB Journal*, 1(1):1, July 1992.

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

[29]F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Deriving active rules for WorkFlow enactment. In *Proceedings of DEXA 96, Zurich (CH)*, 1996.

[30]S. Ceri, P. Grefen, and G. Sanchez. WIDE: A Distributed Architecture for Workflow Management. In *Proc. of 7th Intl. Workshop on Research Issues in Data Engineering*, Birmingham, U.K., April 1997.

[31]S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

[32]S. Chakravarthy et al. Hipac: A research project in active time-constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, July 1989.

[33]S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Knowledge and Data Engineering Journal*, 14:1–26, 1994.

[34]D. D. Chamberlin and F. B. Schmuck. Dynamic data distribution (D3) in a shared-nothing multiprocessor data store. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

[35]S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, 26(1), March 1997.

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

[37]P. K. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proc. ACM SIGMOD Conf.*, page 194, Atlantic City, NJ, May 1990.

[38]P. K. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 103, Barcelona, Spain, September 1991.

[39]R. Conradi et al. *Software Process Modeling and Technology, A. Finkelstein and J. Kramer and B. Nuseibeh (eds.)*, chapter EPOS: Object-Oriented Cooperative Process Modelling, pages 33–70. John Wiley & Sons, 1994.

[40]Q. Cui and J. Gannon. Data-oriented exception handling. *IEEE Trans. on Softw. Eng.*, 18(5):393, May 1992.

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

[42]J. Davis, W. Du, and M.C. Shan. OpenPM: an Enterprise Process Management System. *Bulletin of the Technical Committee on Data Engineering*, 18(1), March 1995. IEEE Computer Society.

[43]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.

[44]U. Dayal, M. Hsu, and R. Ladin. A Transaction Model for Long-running Activities. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 113–122, August 1991.

[45]W. Du, J. Davis, M.C. Shan., and U. Dayal. Flexible Compensation of Workflow Processes. Technical report, Hewlett-Packard Laboratories, 1997.

[46]W. Du and A. Elmagarmid. Quasi Serializability; A Correctness Criterion for Global Concurrency Control in Interbase. In *Proc. Intl' Conf. on Very Large Data Bases*, August 1989.

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

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

[49]E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback recovery with low overhead, limited rollback and fast output control. *IEEE Transactions on Computers: Special Issue on Fault Tolerant Computing*, May 1992.

[50]Abdel Aziz Farrag and M. Tamer Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[51]C. Fernstrom. PROCESS WEAVER: Adding Process Support to UNIX. In *Proceedings of the Second International Conference on the Software Process*, pages 12–26, December 1993.

[52]L. Fisher. *The Workflow Paradigm — The Impact of Information Technology on Business Process Reengineering.* Future-Strategies Inc, 1995.

[53]H. Garcia-Molina, G. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-Transaction Activities. Technical Report CS-TR-247-90, Princeton University, 1990.

[54]H. Garcia-Molina, G. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling Long-Running Activities as Nested Sagas. *Bulletin of the Technical Committee on Data Engineering*, 14(1), March 1991. IEEE Computer Society.

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

[56]Georgakopolous D. and Hornick M. and Sheth A. An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–152, April 1995.

[57]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.

[58]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.

[59]D. Georgakopoulos and M. Rusinkiewicz. Tutorial: Workflow Management – From Business Process Automation to Inter-Organizational Collaboration. In *Intl. Conference on Very Large Databases (VLDB)*, Athens, Greece, 1997.

[60]James Gosling and Henry McGilton, editors. *The Java Language Environment*, Available at `http://www.javasoft.com/nav/read/whitepapers.html`.

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

[62]R. Günthör. The Dependency Manager. In *RIDE-NDS, New Orleans, Louisiana*, 1996.

[63]G. Heineman and G. E. Kaiser. The CORD Appraoch to Extensible Concurrency Control. In *Proc. of 13th Intl. Conference on Data Engineering*, Birmingham, U.K., April 1997.

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

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

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

[67]S.M. Sutton Jr., D. Heimbigner, and L.J. Osterweil. APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3), 1995.

[68]S.M. Sutton Jr. and L.J. Osterweil. The Design of a Next-Generation Process Language. In *Software Engineering, ESEC/FSE 97, LNCS 1301, M. Jazayeri and H. Schaure (eds.)*, pages 142–158, 1997.

[69]S.M. Sutton Jr., P.L. Tarr, and L.J. Osterweil. An Analysis of Process Languages. Technical Report 95-78, Department of Computer Science, University of Massahusetts at Amherst, August 1995.

[70]G. Junkermann, B. Peuschel, W. Schafer, and S. Wolf. *Software Process Modeling and Technology, A. Finkelstein and J. Kramer and B. Nuseibeh (eds.)*, chapter MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment, pages 103–129. John Wiley & Sons, 1994.

[71]G.E. Kaiser, N.S. Barghouti, and M.H. Sokolsky. Experience With Process Modeling In The MARVEL Software Development Environment Kernel. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, pages 131–140, 1990.

[72]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.

[73]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., June 1995.

[74]M. Kamath and K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. In *(To appear in) Proc. of Intl. Conference on Data Engineering (ICDE), Orlando, Florida*, 1998.

[75]M. Kamath and K. Ramamrithan. Performance characteristics of epsilon serializability with hierarchical inconsistency bounds. In *Proc. IEEE International Conferece on Data Engineering (ICDE)*, page 587, Vienna, Austria, April 1993.

[76]M. Kamath and K. Ramamrithan. Bridging the gap between transaction management and workflow management. In *Proc. of NSF Workshop on Workflows and Process Automation in Information Systems*, Athens, Georgia, May 1996.

[77]M. Kamath and K. Ramamrithan. Correctness issues in workflow management. *Distributed Systems Engineering Journal — Special Issue on Workflow Systems*, 3(4):213–221, December 1996.

[78]M. Kamath and K. Ramamrithan. Efficient transaction support for dynamic information retrieval systems. In *Proc. of 19th ACM SIGIR Int'l. Conf. on on Research and Development in Information Retrieval*, pages 147–155, Zurich, Switzerland, August 1996.

[79]M. Kamath, K. Ramamrithan, N. Gehani, and D. Lieuwen. Worldflow : A system for building global transactional workflows. In *Proc. of 7th Intl. Workshop on HPTS (High Performance Transaction Systems) 97*, Asilomar, California, September 1997.

[80]G. Kappel, P. Lang, S. Rausch-Schott, and W. Retschitzegger. Workflow management based on objects and rules. *Bulletin of the Technical Committee on Data Engineering*, 18(1), March 1995. IEEE Computer Society.

[81]J. Klein. Advanced Rule Driven Transaction Management. In *Proceedings IEEE Spring Compcon*, 1991.

[82]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.

[83]T. Kreifelts and G. Woetzel. Distribution and Exception Handling in an Office Procedure System. In *Office Systems: Methods and Tools, Proc. IFIP WG 8.4 Work. Conf. on Methods and Tools for Office Systems*, pages 197–208, 1986. October, 22-24, Pisa, Italy.

[84]N. Krishnakumar and A. Sheth. Specifying Multi-system Workflow Applications in METEOR. Technical Report TM-24198, Bellcore, May 1994.

[85]H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski. Executable Workflows: A Paradigm for Collborative Design on the Internet. In *Proc. of 34th Design Automation Conference*, Anaheim, CA, June 1997.

[86]F. Leymann. Supporting business transactions via partial backward recovery in workflow management. In *Proc. of BTW'95*, Dresden, Germany, March 1995. Springer Verlag.

[87]F. Leymann and D. Roller. Business Processes Management with FlowMark. In *Proc. 39th IEEE Computer Society Int'l Conference (CompCon), Digest of Papers*, pages 230–233, San Francisco, California, February 28 – March 4 1994. IEEE.

[88]B. Liskov and R. Scheifler. Data-oriented exception handling. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[89]D. Lomet and G. Weikum. Efficient and Transparent Application Recovery in Client-Server Information Systems. In *Proc. of SIGMOD Intl. Conference on Management of Data, Seattle*, 1998.

[90]F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. Distributed Object Management. *Intl. Journal of Intelliegent and Cooperative Information Systems*, 1(1), March 1992.

[91]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.

[92]S. McCready. There is more than one kind of Work-flow Software. *Computer-World*, November 1992.

[93]R. Medina-Mora, H.K. Wong, and P. Flores. ActionWorkflow as the Enterprise Integration Technology. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[94]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.

[95]J.A. Miller, A.P. Sheth, K.J. Kochut, and X. Wang. CORBA-Based Run Time Architectures for Workflow Management Systems. *(to appear in) Journal of Database Management, Special Issue on Multidatabase Systems*, 7(z), 1996.

[96]C. Mohan. Tutorial: State of the Art in Workflow Management Research and Products. In *ACM SIGMOD*, Montreal, Canada, 1996.

[97]C. Mohan, G. Alonso, R. Gunther, and M. Kamath. Exotica: A research perspective on workflow management systems. *Bulletin of the Technical Committee on Data Engineering*, 18(1), March 1995. IEEE Computer Society.

[98]C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), March 1992.

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

[100]J. Eliot B. Moss. *Nested Transactions, an Approach to Reliable Distributed Computing.* The Machine Intelligence, eds: Meltzer, and Michie, vars. PublishersT Press, 1985.

[101]M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall publishers, 1991.

[102]C. Pu, G. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 26, Los Angeles, CA, August 1988.

[103]K. Ramamritham and P. Chrysanthis. A taxonomy of correctness criteria in database applications. *The VLDB Journal*, January 1996.

[104]K. Ramamritham and P. K. Chrysanthis. Advances in concurrency control and transaction processing, 1996.

[105]K. Ramamritham and P.K. Chrysanthis. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In *Distributed Object Management, Ozsu, Dayal, Valduriez, Ed.*, Morgan Kaufmann Publishers, 1992.

[106]K. Ramamritham and P.K. Chrysanthis. Advances in Concurrency Control and Transaction Processing, IEEE Computer Society Press (to appear)., 1996.

[107]K. Ramamrithan and M. Kamath. Building virtual warehouses using workflows and the web (invited paper). In *International Workshop on Databases*, Hong Kong, July 1997.

[108]F. Ranno, S. Shrivastava, and S. Wheater. A System for Specifying and Coordinating the Execution of Reliable Distributed Applications. Technical report, CS Dept., University of Newcastle upon Tyne, 1997.

[109]B. Reinwald. Tutorial: Workflow Management Systems: Programming, Architecture, Commercial Products. In *IEEE Intl. Conference on Data Engineering*, Birmingham, U.K., 1997.

[110]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.

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

[112]Heikki Saastamoinen. *On the Handling of Exceptions in INformation Systems.* Jyvaskyla Studies in Computer Science, 1993.

[113]M. Saeki, T. Kaneko, and M. Sakamoto. A Method For Software Process Modeling and Description using LOTOS. In *Proceedings of the First International Conference on the Software Process*, pages 90–104, October 1991.

[114]K. Salem, H. Garcia-Molina, and J. Shands. Altruistic Locking. *ACM Trans. on Database Sys.*, 19(1):117, March 1994.

[115]B. Salzberg and D. Tombroff. DSDT: Durable Scripts Containing Database Transactions. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 1996.

[116]H. Schuster, S. Jablonski, T. Kirsche, and C. Bussler. A client/server architecture for distributed workflow management systems. In *Proc. of Third Int'l. Conf. on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.

[117]F. Schwenkreis. APRICOTS - A Prototype Implementation of a ConTract System - Management of Control Flow and the Communication System. In *Proc. of the 12th Symposium on Reliable Distributed Systems, IEEE Computer Society Press*, Princeton(NJ), 1993.

[118]A. Sheth, editor. *Proceedings of NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, `http://lsdis.cs.uga.edu/activities/NSF-workflow/`, May 1996.

[119]A. Sheth et al. Supporting State-Wide immunization Tracking using Multi-Paradigm Workflow Technology. In *Proceedings of the 22nd VLDB Conference*, Bombay, India, 1996.

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

[121]M. P. Singh. Distributed Scheduling of Workflow Computations. Technical Report TR-96-18, Dept of Computer Science, North Carolina State University, 1996.

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

[123]M. Suzuki and T. Katayama. Meta-Operations In The Process Model HFSP For The Dynamics And Flexibility Of Software Processes. In *Proceedings of the First International Conference on the Software Process*, pages 202–217, October 1991.

[124]P.L. Tarr and L.A. Clarke. PLEIADES: An Object Management System for Software Engineering Environments. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 56–70, December 1993.

[125]Lee The. Workflow Tackles the Productivity Paradox. In *Datamation Magazine*, August 1995.

[126]R. Unland and G. Schlageter. A Transaction Manager Development Facility for Non Standard Database Systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 11, pages 399–466. Morgan Kaufmann Publishers, San Mateo, 1992.

[127]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.

[128]G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Trans. on Database Sys.*, 16(1):132, March 1991.

[129] G. Weikum and Hans-J. Schek. Concepts and Applications of Multilevel and Open Nested Transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13, pages 515–554. Morgan Kaufmann Publishers, San Mateo, 1992.

[130] Thomas White and Layna Fischer, editors. *New Tools for New Times: The Workflow Paradigm.* Future Strategies Inc., Book Division, Alameda, CA, 1994.

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

[132] 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.