AD-HOC RECOVERY IN WORKFLOW SYSTEMS: FORMAL MODEL AND A PROTOTYPE SYSTEM

CENTRE FOR NEWFOUNDLAND STUDIES

TOTAL OF 10 PAGES ONLY MAY BE XEROXED

(Without Author's Permission)

XUEMIN XING

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

NOTE TO USERS

This reproduction is the best copy available.

UMI®

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-62442-0

Canada

Ad-hoc Recovery in Workflow Systems: Formal Model and a Prototype System

by

Xuemin Xing

A thesis submitted to the School of Graduate Studies
in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE

Department of Computer Science
Memorial University of Newfoundland
Newfoundland, Canada

February, 1999

© Copyright 1999

by

Xuemin Xing

Dedication

To my parents

for their encouragement and support

throughout the course of my education.

Abstract

A *workflow management system (WFMS)* facilitates business processing (*workflow*) across distributed nodes. State-of-the-art WFMSs do not have adequate support for the dynamic changes during the workflow execution. This thesis focuses on one of the dynamic problems, *ad-hoc recovery*. It is a phenomenon that occurs in workflow applications when an *agent* needs to alter the *control flow* prescribed in the original definition. Specifically, we are interested in the backward ad-hoc recoveries, in which the control flow is redirected backward. When this happens, some tasks will be re-executed and consistency problems may arise. In our proposed ad-hoc recovery model, the key components of the ad-hoc recovery are defined and some constraints are given to ensure the correctness of the workflow execution. We also present a WFMS prototype, describing its design strategy and implementation method, as well as a related protocol, as one application of this model. The protocol is exemplified by a hospital workflow. Some performance issues are also discussed.

Acknowledgments

This research could never have been accomplished without the constant support, encouragement and suggestions from my professor, Dr. Jian Tang. His prudent manner and innumerable criticisms pushed the work to a better and better quality. I would also like to acknowledge the comments, advice, and assistances from Dr. Gregory Kealey in the school of graduate studies, Dr. Paul Gillard, Ms. Elaine Boone, Dr. Manrique Mata-Montero, Mr. Nolan White, and Mr. Steven Johnstone, etc. in the Department of Computer Science during my studies and research. I thank all my friends in St. John's whose presence made life as a graduate student much more palatable. Finally I offer a deep and a sincere appreciation to my parents for their support throughout all these years.

Table of Contents

Abstract **iv**

Acknowledgments **v**

1 Introduction **1**

1.1 Workflow management concepts 1

1.2 Research issues 2

1.3 Motivation 4

1.4 Thesis structure 5

2 Relevant work **6**

2.1 Advanced transaction models 7

2.1.1 Sagas 7

2.1.2 Flexible Transaction Model 8

2.1.3 ConTract Model 9

2.2 Transactional Workflows 10

2.2.1 FlowMark 10

2.2.2 Action Workflow 11

2.2.3 $METEOR_2$ workflow model 12
2.2.4 C-Units 13
2.2.5 INCAs 13

3 A generic workflow model 15

3.1 Basic components 15
3.2 Definition graph (DG) 17
3.3 Normal execution 20

4 Ad-hoc recovery model 23

4.1 Two types of ad-hoc recoveries 23
4.2 Backward ad-hoc recovery 25
4.2.1 Repetition of task instances and run 26
4.2.2 Redirected execution 28
4.2.3 Redirect-to (RDT) and redirect-from (RDF) sets 36
4.2.4 Execution graph (EG) 42
4.2.5 RDF and RDT in terms of EG in backward recovery 44

5 A WFMS prototype supporting backward ad-hoc recovery 50

5.1 Design and runtime representation 51
5.2 Outline of a client-server architecture supporting ad-hoc recovery . . 59
5.3 Components 63
5.3.1 Graphical workflow designer 63
5.3.2 Workflow server 65
5.3.3 Task manager 68

5.3.4 Database server 70

5.4 Backward ad-hoc recovery protocol 70

6 Implementation and an example 74

6.1 Java features for enterprise computing 74

6.1.1 Java DataBase Connectivity (JDBC™) 75

6.1.2 Remote Method Invocation (RMI™) 76

6.2 Some implementation details 78

6.3 Ad-hoc recovery in a hospital workflow 79

6.4 Discussion about availability and scalability 83

7 Conclusions and future work 90

Bibliography 93

List of Figures

3.1 An example DG 18

3.2 Example for SFT-dependency 21

4.1 Order processing workflow 24

4.2 Well defined sequence of backward redirected executions 30

4.3 Orders in well defined sequence of backward redirected executions . . 31

4.4 Proof of Theorem 4.2 34

4.5 Example of non-minimal $< RDT', RDF >$ 41

4.6 An example EG 43

4.7 Constraints on RDT and RDF in terms of EG 49

5.1 Structures of transactional tasks 52

5.2 Structures of non-transactional tasks 53

5.3 A client-server architecture supporting ad-hoc recovery 60

5.4 Types of client-server computing 61

5.5 The infrastructure of the prototype 62

5.6 The snapshot of the Graphical Workflow Designer 64

5.7 Transition of a work item between lists 68

5.8 Task manager GUI 69

6.1 Flow chart of doneInAH() 78

6.2 Flow chart of donePostAH() 79

6.3 The initial Nurse Task Manager window 81

6.4 Processing work item Tom in the Nurse Task Manager window 82

6.5 Processing work item Tom in the Doctor Task Manager window . . . 83

6.6 External ad-hoc request and choose restart-from tasks 84

6.7 The decision 85

6.8 Undo Doctor task 85

6.9 Undo Nurse task 86

6.10 Redo Nurse task 87

6.11 Redo Doctor task 87

6.12 Build time data 88

6.13 Run time data 89

Chapter 1

Introduction

To be competitive, organizations are increasingly relying on enterprise-wide integration of information using advanced technologies. Among this trend, *workflow management* has emerged as a practical technique to automate *business processes.* It has attracted attention from both industry and research communities in numerous application domains such as telecommunications, manufacturing, finance and banking, health care, and office automation [35, 12, 27, 6, 1, 5]. Many companies in these fields have shown interest in adopting this technology for conducting their daily business.

1.1 Workflow management concepts

The interest in workflow had originated from office automation, image processing etc. in early 1980's. However, at that time, there was only very limited resource sharing and the coordination was mainly done manually. This limited the productivity of the business processes. The solution lies on the effective and efficient coordination among various segments that constitute a business process.

According to Workflow Management Coalition [16], the *workflow* is "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." According to this definition, a workflow is an abstraction of a business process. Simply put, a workflow is a collection of tasks. A task defines some work to be done. The execution of a workflow must fulfill a well defined business goal through the cooperation of all the constituting tasks. This cooperation is achieved through the coordinated execution of the tasks. Such a coordination is provided by a set of software tools called *workflow management systems* (WFMS).

The workflow life cycle can be divided into a build-time phase and a run-time phase. Build-time phase mainly concerns the modeling and specification of the business process, while the run-time phase concerns its enactment. These functions are provided by WFMSs.

1.2 Research issues

The work carried out in the past is mainly concerned with process modeling and design, inter-task dependency and scheduling, and workflow management system design [34, 33, 8, 29, 30, 32, 3, 24]. However, while growing in popularity, many problems still remain, such as the lack of a clear theoretical basis, limited support for heterogeneous and distributed computing infrastructures, lack of inter-operability and dynamic features, and lack of reliability in the presence of failures and exceptions [38, 36, 12, 20, 31, 18, 13, 14, 19]. For example, the scalability, heterogeneous and

distributed computing infrastructure, and recovery schema were studied in [36, 32, 7]. In [19] the authors summarized some of the most glaring limitations of existing workflow systems.

1. Existing systems are almost totally incompatible. These incompatibilities are not just the syntax or the platform, but the very interpretation of workflow execution. In most cases, the system is so tied to the underlying support system that it is not feasible to extend its functionality to accommodate other workflow interpretations.

2. Systems are too dependent on the modeling paradigm (Petri-nets, state charts, transactional dependencies, to name a few) and there is no clear understanding of the execution model of workflow processes.

3. The architectural limitations(single database, poor communication support, lack of foresight in the design, the problems posed by heterogeneous designs) have prevented existing systems from being able to cope with a fraction of the expected load.

4. Lack of robustness and very limited availability. Current products have a single point of failure (the database) and no mechanism for backup or efficient recovery. Moreover, since workflow systems will operate in large distributed and heterogeneous environments, there will be a variety of components involved in the execution of a process. Any of these components can fail and, nowadays, there is not much that can be done about it. Existing systems lack the redundancy and flexibility necessary to replace failed components without having to

interrupt the functioning of the system.

1.3 Motivation

Our research is mainly concerned with the dynamic features of workflow systems. We notice that most of the workflow products are based on the pre-defined structure. Pre-defining a business process can simplify the design and the implementation. But it lacks flexibility in that the execution must follow the workflow definition strictly. It cannot go back or skip forward. However, due to the frequent occurrence of *ad-hoc events*, the execution orders do need to be changed at run time. These ad-hoc events may result from the changing environments, unexpected intermediate results, or personal favors, etc. For example, the sudden cancellation of a flight, strike in the post-office, or changing an order, are all ad-hoc events that cannot be predicted in advance. Therefore, it is impossible to use a pre-defined exception handler to deal with the ad-hoc events.

Our solution to this problem is to combine the pre-defined structure with a changeable control flow. Since there is no clear basis about the workflow execution, we formalize a workflow execution model to depict both normal executions and *ad-hoc recoveries* in a workflow. This model does not include the dynamic changes made to the workflow definition. Thus it can handle only a specific kind of dynamic changes.

Another motivation is end-user involvement. During the evolution of the information systems, more and more attention is paid to end-users since they are the essence of creativity and productivity. In our model, the end-users are given more rights to

take part in the control than in other workflow products.

1.4 Thesis structure

The rest of the thesis will be organized in the following way. Chapter 2 contains a review of the relevant work. Chapter 3 introduces a generic workflow model. Chapter 4 discusses the issues related to the changeable control flow and formalizes the backward ad-hoc recovery model. Chapter 5 discusses the design of a prototype system[1]. Chapter 6 mentions some implementation features with an example hospital workflow. Section 7 gives conclusions and some future work.

[1] A preliminary version of the proposed client-server architecture is presented at the International Database Engineering and Applications Symposium (IDEAS 99).

Chapter 2

Relevant work

The concept of ad-hoc recovery is related with dynamic workflows, exception handling and recovery in workflow management systems, and cooperative information systems. It crosses the boundaries of these fields and grows into a specific topic. The ad-hoc recovery model makes use of the pre-defined workflow structure, and allows dynamic redirections of the control flow at run time. It can be called an exception handling mechanism but the exceptions it handles are those ad-hoc events that cannot be predicted in advance. It provides the users more ways to cooperate in order to fulfill the business goal.

To build the context for discussion, a brief overview of the related area will be given in this chapter. We start from the advanced transaction models which relax the ACID properties of the traditional transaction model. Then several typical workflow prototypes/products will be introduced to compare with our approach.

2.1 Advanced transaction models

Advanced transaction models (ATMs) extend the basic transaction models to incorporate more complex transaction structures and relax the traditional ACID properties. The complex transaction structures in advanced transaction models include nested and multi-level transactions; The relaxation of ACID properties refers to a controlled relaxation of the isolation and atomicity to better match the requirements of various database applications. Correctness criteria other than global serializability have also been proposed.

However, many of these extensions have resulted in application- specific ATMs that offer adequate correctness guarantees in a particular application, but not in others. Furthermore, an ATM may impose restrictions that are unacceptable in one application, yet required by another. If no existing ATM satisfies the requirements of an application, a new ATM is defined to do so.

2.1.1 Sagas

A saga [23] is a long-lived transaction that consists of a set of relatively independent subtransactions T_1,T_2, ...,T_n each of which has a compensating subtransaction C_1, C_2,...,C_n. To execute a Saga, the system must guarantee that either the sequence T_1,T_2,...,T_n (successful) or the sequence T_1,T_2,...,T_i,C_i,...,C_2,C_1 (undo partial execution) for some $1 \leq i < n$.

Sagas preserve the atomicity and durability properties of traditional transactions but relax the isolation property by allowing a saga to reveal its partial results to other

transactions before it is complete. It is useful only in limited environment because of its consistency problem and the requirement of compensatable transactions.

The concept of compensating task is first proposed in Saga. It is used in our ad-hoc recovery model too but it is not required that every task must have a compensating task. Moreover, undoing the partial execution sequence in ad-hoc recovery does not necessarily follow the reverse order of tasks as described in compensating a saga. Instead, it is possible that a task is not compensated during the undo phase but is compensated during the redo phase.

2.1.2 Flexible Transaction Model

Flexible transaction model [2] was designed to allow more flexibility in transaction processing. A flexible transaction is a set of tasks. For each task, the user can specify a set of *functionally equivalent* subtransactions, each of which when completed will accomplish the task. The execution of a flexible transaction succeeds if all of its tasks are accomplished. A flexible transaction is resilient to failures in the sense that it may proceed and commit even if some of its subtransactions fail.

The flexible transaction model also allows the specification of dependencies on the subtransactions, including failure- dependencies, success-dependencies, and external-dependencies. Flexible transactions use compensation and relax global atomicity requirements by allowing the transaction designer to specify acceptable states for termination of the flexible transaction, in which some subtransactions may be aborted. Because flexible transactions share some of the features of a workflow model, it was perhaps the first ATM to have been tried to prototype workflow applications [28].

Through the use of compensating subtransactions, the flexible transaction model allows the user to control the isolation granularity of a transaction. However, all the failures are handled before one task is accomplished. It is not possible for the task to roll back after it is committed. That is different from our approach in which committed tasks can be rolled back too.

2.1.3 ConTract Model

A *ConTract* [4] is a consistent and fault tolerant execution of an arbitrary sequence of predefined actions (called *steps*) according to an explicitly specified control flow description (called *script*). The main emphasis of the ConTract model is that the execution of a ConTract must be *forward-recoverable*. This means that when an execution of a ConTract is interrupted by failures, it must be re-instantiated and continued from where it was interrupted after the system is recovered. To be able to do so, all state information, including database state, program variables of each step, and the global state of the ConTract, must be made recoverable. These state information form the *context* in a ConTract. ConTracts provide relaxed atomicity and relaxed access restrictions on shared objects.

Forward recovery of a ConTract is different from the forward ad-hoc recovery. The former concerns how to continue the current task execution after a failure. The latter concerns a forward jump from the current task to a follow up one without affecting the semantics of the workflow.

2.2 Transactional Workflows

The concept of transactional workflow clearly recognizes the relevance of transactions to workflows. A transactional workflow is characterized by selective use of transactional properties for individual tasks or an entire workflow.

Two major approaches have been used to study and define transactional workflows. The first one is an embedded approach that assumes that the existing entities support some active data management features. This approach is frequently used in dedicated systems developed to support a particular class of workflows and usually involves modification of the executing entities. The second one is a layered approach that implements workflow control facilities on top of uniform application-level interfaces to execution entities. The degree to which each model incorporates transactional features varies, and depends largely on the requirements (such as flexibility, atomicity and isolation of individual task executions and multiple workflow instances, etc.) of the organizational processes it tries to model.

2.2.1 FlowMark

FlowMark is a product from IBM Corporation. It has been designed to manage long duration business processes in large enterprises. FlowMark runs across different platforms and supports distribution of most of its components. It is organized into five components: *FlowMark Server, Runtime Client, Program Execution Client, Buildtime Client, and Database Server* [21].

In [21], the authors analyzed the current architecture of FlowMark and proposed

extensions for better resiliency to failures. The previous architecture provides forward recovery of business processes interrupted by *system failures.* One extension is to deal with *semantic failures.* In particular, if a certain activity is unable to perform the desired task, alternative completion paths should be provided to the business processes. This may involve undoing part of the results of the path being executed until a point of the execution is reached where an optional path can be taken. This approach is part of IBM Almaden Research Center's Exotica project, which aims at incorporating advanced transaction management capabilities in IBM's products and prototypes.

Semantic failures defined in FlowMark are those that occur when a particular state of the system does not allow a certain task to be performed. For instance, to try to deliver an item that is not in stock, or to try to withdraw money from an empty bank account. In these cases, the business process cannot proceed as planned and optional courses of action should be taken.

FlowMark provides a way to group several tasks into a *sphere of control* which can maintain the integrity of them. If a task in a sphere fails, all the others must be compensated for or rolled back. It is a kind of pre-defined exception handler.

2.2.2 Action Workflow

The Action Workflow from Action Technologies Inc. consists of products that rapidly develop and deploy Web- and client/server-based work management solutions [9]. It is a communication based model where a business processing is composed of *workflow loops.* Each workflow loop has two participants, the customer and the performer. The

workflow loop can be invoked multiple times until the customer is satisfied.

A workflow loop has four states, which represent the four phases of the interaction between the customer and the performer: preparation, negotiation, performance, and acceptance. During the performance phase, secondary workflow loops may be invoked. In this case, the performer in the main workflow loop becomes the customer in the secondary loop.

The purpose of the Action Workflow is to satisfy the customer. However, there are business processes where the customer emphasis may be superficial, e.g., if the objectives are to minimize information system cost or reduce waste of material in a process. Therefore, Action Workflow is not appropriate for modeling business processes with objectives other than customer satisfaction.

2.2.3 $METEOR_2$ workflow model

The $METEOR_2$ workflow project at the Large Scale Distributed Information Systems Lab., University of Georgia is a continuation of the *METEOR* [32] effort at Bellcore. The $METEOR_2$ WFMS is being designed and developed to support coordination of automated and user tasks in the real world heteorgeneous, autonomous, distributed (HAD) environment.

It supports modeling of a hierarchical workflow process (with compound tasks), behavioral aspects of heterogeneous human-performed and application/system tasks (what can be observed, what can be controlled for a task execution by a given processing entity), inter-task dependencies (with control and data flow), specification of interfaces (involved in supporting legacy applications, client/server processing

and distributed processing), run-time environment and task assignments to various system components, error handling and recovery requirements, etc [10].

$METEOR_2$ is based on the pre-defined workflow structure and the scheduling information is embedded in the individual tasks. A recovery model is incorporated into the $METEOR_2$ workflow structure at different levels. However, $METEOR_2$ is not a dynamic WFMS. In order to be dynamic, primary enhancement is needed in the areas of modeling collaboration and specification of dependencies or interactions among tasks that support coordination and collaboration [10].

2.2.4 C-Units

One of the projects in which transactional semantics have been applied to a group of steps defines a logical construct called a *Consistency unit* (C-unit) [25]. A C-unit is a collection of workflow steps and enforced dependencies between them. It is similar to the sphere of control in FlowMark, but a C-unit can dissolve itself if it is committed or aborted.

2.2.5 INCAs

The *INformation CArrier* (INCA) [11] workflow model was proposed as a basis for developing dynamic workflows in distributed environments where the processing entities are relatively autonomous in nature. In this model, the INCA is an object that is associated with each workflow and encapsulates workflow data, history and processing rules. The transactional semantics of INCA procedures (or steps) are limited by the transaction support guaranteed by the underlying processing entity. The INCA

itself is neither atomic nor isolated in the traditional sense of the terms. However, trnasactional and extended transactional concepts such as *redo of steps, compensating steps* and *contingency steps* have been included in the INCA rules to account for failures and forward recovery.

INCA rules give recovery strategy for each processing entity in the workflow. But because the processing entities are autonomous, one cannot invoke another for recovery. In our model, a scope of tasks can be found for recovery at run-time and they are recovered in a logical order.

Chapter 3

A generic workflow model

In this chapter, we present a generic workflow model and describe in reasonable detail the related components that are vital to the theme of this thesis. The model is based on the one given by WfMC [16] and the basic concepts have been used in most of the literatures in this area. This model describes the normal workflow execution and will be used as the basis to formalize the ad-hoc recovery model in the subsequent chapter.

3.1 Basic components

A workflow is a set of tasks. A *task* represents the smallest unit of work in a workflow. It may be manual or a computer program. The internal structure of a task is transparent to the workflow level. (This makes it possible to implement tasks flexibly.) However, we do require that some states of a task to be externally observable. These states include, for example, *not executing*, *executing*, *commit*, *abort*, *done*, *fail*, *succeed*. (The exact set of externally observable states depends on workflow

applications.) Tasks can run at different geographical locations and be executed by heterogeneous processing units. (However, the issue of how to handle the heterogeneity is not the concern of this thesis.) In general cases, at any time there exists only one execution for a specific task at its processing unit. Otherwise, if multiple copies are executed concurrently, it would be confusing which output should be adopted.

In the general case, the workflow is in a certain state when the task starts execution, and may change to a different state after the execution terminates. Thus any task execution can be viewed as a mapping from a particular workflow state to the other. Since sites may preserve local autonomy, we assume a task execution can be affected only by the states local to the site where the task is executed.[2] Thus in the following we are concerned only with *local states.*

In normal cases, tasks are executed in a well defined order and subject to the satisfaction of certain conditions. This determines the *control dependency* among tasks at run time. Besides control dependency, there are also *data dependency.* A data dependency specifies a data flow among task executions. (There may also exist other types of dependencies, such as *temporal dependency.* But we will not consider them in this thesis because they are largely orthogonal to the topics discussed in this thesis.) These dependencies are sometimes aggregately called *inter-task dependencies.* An inter-task dependency is part of the application semantics and it is the responsibility of the workflow designer to specify the inter-task dependencies properly.

Tasks are invoked by *agents*[3]. An agent can be either a human agent or a com-

[2]However, a task execution can change the local states at other sites. This may happen, for example, when the output of a task is passed to a different site.

[3]In terms of the terminology by WfMC, these correspond to client application only.

puter program. An agent may be identified within the workflow definition or (more normally) identified by reference to a role, which can then be filled by one or more real agents at the run time.

3.2 Definition graph (DG)

It is natural to depict the tasks and their dependencies using a graph. Tasks can be represented by vertices, and their orders by arcs. Each arc is associated with a condition. This kind of graph is termed a *definition graph* in our ad-hoc recovery model.

Definition 3.1 (Definition Graph) *A definition graph is a weighted directed graph DG in which each vertex T_i represents a task and each arc*

$$T_i \xrightarrow{c} T_j$$

is a constraint such that:

1. *If T_j is an and-joint task, it can be activated only after the condition c associated with each of its incoming arc is satisfied.*
2. *If T_j is an or-joint task, it can be activated when the condition c associated with one of its incoming arcs is satisfied.*

A vertex is a *minimal* vertex of the DG if there is no incoming arc to it. It is a *maximal* vertex if there is no outgoing arc from it. If two of the conditions of the or-joint vertex both are true, we assume the task will be invoked twice, according to each of the conditions, respectively. Task T_i is said to be T_j's *ancestor* (or T_j is T_i's *descendent*) if there is a path from T_i to T_j in the DG. A definition graph describes the structure of a workflow, namely, its constituent tasks, their execution orders and the conditions under which these orders actually happen.

Figure 3.1 is a typical *DG*. After T_4 finishes, T_5 is activated immediately. Whether T_7 can be activated or not depends on the truth value of c_3. After T_5 finishes, either T_6 or T_2 (but not both) will be activated depending on c_1. T_{10} will be activated when T_8 and T_9 terminate successfully and both c_2 and c_4 are satisfied.

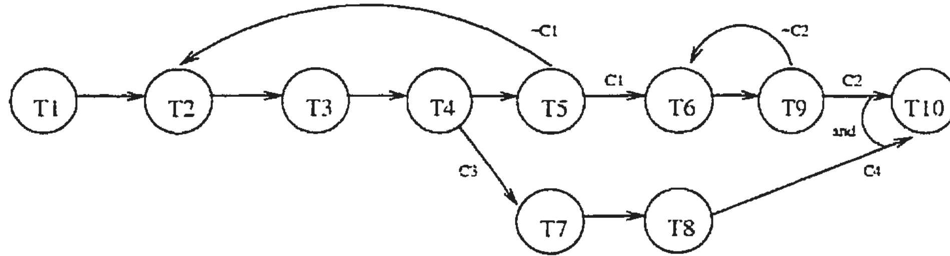


Figure 3.1: An example *DG*

Note that *DG* provides only a schema for the workflow execution, but is not the execution itself. For example, in Figure 3.1 both T_2 and T_6 depend on T_5. Suppose after the workflow is started, tasks $T_1, \cdots, T_5$ are executed successfully in that order. After T_5 is finished, condition c_1 is false. Then the next task to be invoked is T_2. (Note: this is the second invocation of T_2.) This is followed by T_3, T_4, and T_5 again. Suppose after T_5 terminates for the second time, c_1 becomes true. Thus T_6 is invoked. As can be seen from this scenario, in order to describe this execution, we must be able to model the execution of a task. To this end, we use the notation of *task instance*.

The task instances are generated according to the schema given in the workflow definition (i.e. *DG*). When the activation condition becomes true, the task will be executed, and therefore a task instance is generated. Note that, invocations of the or-joint task from different arcs are looked on as different task instances. As mentioned before, the generation of the task instance is associated with a local state.

In this thesis, we treat a local state as an entity which has two attributes, *signature* and *content*. A signature uniquely determines a local state. The content of a local state is a mapping $P \rightarrow Q$, where P is the set of all workflow relevant data items at that site, and Q is the set of all possible values for these data items. For any local state, we use function θ to map its signature to its content. For example, P={city, province}, Q={Toronto, Ottawa, Vancouver, ON, BC}. Mapping {city=Toronto, province=ON} is the content of a local state. We mark this state with signature s. Thus $\theta(s)$={city=Toronto, province=ON}

As mentioned before, at any time there exists only one execution for a specific task. Thus each task instance is unique at any time. So is its local state. In other words, the local state in which the task is invoked to generate it is determined. We observe that the signature of the local state which identifies it is unique at any time, too. However, before the task is invoked there usually exists some choices of the local states for the task invocation. This is partly because of the existence of the human factor. The user can choose from a range of input values. To characterize such a multiple choices of local states, we introduce the notion of *legal set*. The legal set for a task instance is an entity which contains all the valid local states in which the task could be invoked prior to the start of that task instance. (In some sense, this is similar to the concept of *set of consistent database states* in the traditional transaction model. A transaction can start from any state in that set.)

Any task instance must be distinguishable from the others even if multiple task instances can be instantiated from the same task. This is because from the viewpoint of the workflow schema, the presence of every task instance has a unique occasion.

Put differently, every task instance plays a unique role in the workflow execution. For example, even if a task is executed more than once due to the loop in DG, the multiple task instances invoked are logically different. Since each task instance is associated with a legal set, we use legal sets to identify task instances. As a result, the legal sets from which task instances are generated must be distinguishable. For this purpose, like local states, each legal set is also associated with two attributes, signature and content. A signature uniquely determines a legal set. The content of a legal set is the set of contents of all its member local states. For a convenient abuse of symbols, for any legal set we also use θ to map its signature to its content. Let L be a legal set, and S be its signature. Thus we have $\theta(S)=\{\ \theta(s) \mid s$ is the signature of l where $l \in L.\}$. Note that, legal sets may overlap in their contents.

3.3 Normal execution

The legal sets and local states are essential in modeling the *normal execution* of a workflow.

Definition 3.2 (Normal execution) *A normal execution of a workflow is a directed acyclic graph NE where* $V(NE) \subseteq \{<S,s,T> \mid S$ *is the signature of a legal set L associated with an instance of T, s is the signature of a local state l where* $l \in L$*, and* $T \in V(DG)\}$*, such that*

1. $\forall t_1,t_2 \in V(NE)(t_1 \neq t_2 \Longrightarrow t_1.S \neq t_2.S\ \&\ t_1.s \neq t_2.s)$,
2. $(<S,s,T> \rightarrow <S',s',T'>) \in E(NE) \Longrightarrow (T \rightarrow T') \in E(DG)$.

A triple in the above definition models a task instance in that normal execution. The edge in NE denotes the execution order of task instances. In triple $<S,s,T>$, s is the signature of the local state from which the task instance of T is generated. S

is the signature of the legal set for this task instance. The local state is a member of the legal set, modeling the fact that this local state from which the task instance gets started is only one of the possibly many choices present in the legal set. S is unique for each member in $V(NE)$. Since the normal execution is consistent with the workflow schema, $E(NE)$ must have an order consistent with $E(DG)$. In the following, we refer to the vertices without incoming arcs in NE as the *minimal* vertices of NE, and use the function $Min(NE)$ to get the set of minimal vertices of NE. Similarly, we use function $Max(NE)$ to get the set of *maximal* vertices of NE.

For easy presentation, we use the term *Start-Follow-Termination (SFT) dependency* to refer to the relation in the above definition.

Definition 3.3 (SFT-order ($\prec_{SFT}$)) *In a normal execution NE,* $<S_2,s_2,T_2>$ *is SFT-dependent on* $<S_1,s_1,T_1>$*, or denoted as* $<S_1,s_1,T_1> \prec_{SFT} <S_2,s_2,T_2>$*, if* $(<S_1,s_1,T_1> \rightarrow <S_2,s_2,T_2>) \in E(NE)$.

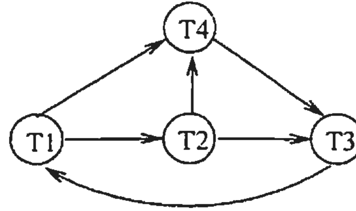


Figure 3.2: Example for SFT-dependency

SFT-dependency expresses the 'immediate follow up' relations between two task instances. For example, consider the workflow shown in Figure 3.2. In this figure, the DG contains four tasks. Suppose the execution order of the tasks is $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_3$. For simplicity, we use T_{ij} to represent the j-th instance of task T_i. (Note that, T_{ij} is associated with a unique triple in the previous definition of normal execution.) Therefore, the NE is $T_{11} \rightarrow T_{21} \rightarrow T_{41} \rightarrow T_{31}$. According to our definition, T_{31} does not SFT-depend on T_{21}.

The notion of SFT-dependency is more restrictive than that of task dependency described in DG. An arc in DG may or may not imply a SFT-dependency. In the following we use phrase *SFT*-dependency* to refer to the closure of SFT-dependency. That is, task instance A SFT*-depends on B if A=B or if A transitively SFT-depends on B. Refer to the above example, T_{31} is SFT*-dependent on T_{21}.

Chapter 4

Ad-hoc recovery model

Ad-hoc recovery is a phenomenon that occurs in workflow applications when an agent needs to alter the control flow prescribed in the original definition. When ad-hoc recovery occurs, the normal workflow execution is interrupted. After the ad-hoc recovery, it assumes the normal execution from a different state.

Ad-hoc recovery is usually caused by unpredictable reasons, such as unexpected output of some individual tasks, events or exceptions due to the changing environment, etc. Because of its irregularity in nature, ad-hoc recovery in general cannot be dealt with by using pre-defined exception handler in a traditional way.

4.1 Two types of ad-hoc recoveries

There are two kinds of ad-hoc recoveries, *backward* and *forward*. The former occurs when it is necessary to stop the execution of certain tasks and restart some previous tasks, and the latter occurs at a point when some follow up tasks should be skipped. In order to understand these, let us look at an order processing workflow depicted in

Figure 4.1.

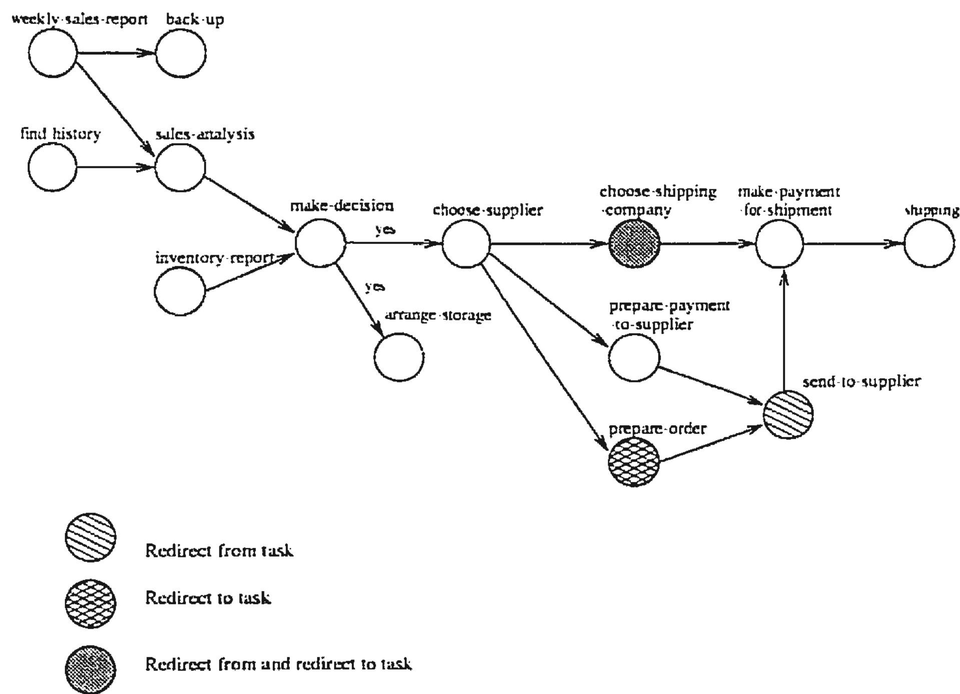


Figure 4.1: Order processing workflow

In this workflow, based on the *weekly-sales-report* and the historical data generated by *find-history*, *sales-analysis* is performed to evaluate the market perspective (i.e., how many merchandise will be sold in the next month). Task *inventory-report* determines the volumes of the merchandise in stock. Based on the outcome of *sales-analysis* and *inventory-report*, *make-decision* determines whether a new order should be placed. If the decision is 'yes', *choose-supplier* and *arrange-storage* will be performed in parallel. After the supplier has been chosen, the following three tasks, *choose-shipping-company*, *prepare-payment-to-supplier* and *prepare-order* are invoked. The successful execution of *prepare-payment-to-supplier* and *prepare-order* will lead to the invocation of *send-to-supplier* which delivers the order and the payment (or

some kind of letter of credit) to the supplier. Then the payment is made to the shipping company which will do the shipment.

Suppose during (or after) the execution of *send-to-supplier*, some events occur that renders it being necessary to increase the quantities of the merchandise on the previous order. The ordering department therefore decides to do the ordering again, or modify the previous order, whichever is more feasible. As a result, *prepare-order* is effectively re-executed. In this case, the control flow is said to be redirected back. This is a typical characteristic of the backward recovery.

The example of a forward recovery may be like this. Suppose there is a power failure in a city due to a snowstorm and candles are in urgent need. If there are not enough candles in stock, a department store may want to place an order immediately. Therefore, it would be desirable that the *sales-analysis* and its previous tasks be skipped and only the *inventory-report* be executed to make a decision. One way to realize the skip is to assemble the output of the *sales-analysis* manually by the user, say 1000 candles are demanded by the market. In this way, the semantic of the workflow is not compromised.

Our efforts are mainly on the formalization of the backward ad-hoc recovery model. The counterpart for the forward recovery remains as future work.

4.2 Backward ad-hoc recovery

The purpose of the backward ad-hoc recovery model is to solve the inconsistencies arising due to the re-execution of some tasks. Consider the order processing workflow,

if the *prepare-order* is re-executed successfully, and there is no control over the *send-to-supplier* task, two orders will be sent to the supplier. This is not the intention of the redirection. On the contrary, the users want to roll back the *send-to-supplier* task and cancel the previous order. To summarize, during the backward recovery, some active tasks cannot proceed and some finished tasks may need to be rolled back.

4.2.1 Repetition of task instances and run

As described before, backward recovery involves the repetition of task instances. However, the repetition of a task instance in the context of ad-hoc recovery has different meaning from that caused by the loop in the workflow definition. In the ad-hoc recovery the repetition of the task instance is logically the same task instance as the previous one even if they are two different executions physically. To understand this, let us look at the order processing workflow again.

Clearly, the re-execution of *prepare-order* essentially plays the same role as the previous execution intended to do. Put differently, these two executions serve the same purpose from the viewpoint of the workflow schema, but only the more recent one counts. We say one is the *peer* of the other. Formally, we have

Definition 4.1 (Peer) *Let NE be a normal execution.* $< S, s', T >$ *is a peer of* $< S, s, T >$ *if* $< S, s, T > \in V(NE)$ *and* $s' \neq s$.

Note that from the definition, peer is symmetric. The fact that peers share the same signature of a legal set indicates that they are logically the same task instance. On the other hand, since $s' \neq s$ they are different physical executions. However, we may or may not have $\theta(s') = \theta(s)$. If the equality holds, these physical executions generate the identical results, otherwise they do not.

In the definition of normal execution given in the previous chapter, a triple is referred to as a task instance, even though this triple actually identifies a physical execution. That is, T is invoked under state s, where $\theta(s) \in \theta(S)$. It is fine in that context where we did not distinguish a task instance with its physical execution, since there is only one physical execution associated with each task instance in a normal execution. With the introduction of peer, however, this is no longer the case. Now a task instance may be associated with multiple physical executions which produce different results. For example, suppose $< S, s', T >$ is a peer of $< S, s, T >$. These two (physical) executions may produce different results since we may have $\theta(s') \neq \theta(s)$. On the other hand, from the viewpoint of the workflow schema, they are associated with the same task instance since both triples share the same signature S of the legal set. Thus there is a need to distinguish a task instance with its physical execution. To this end, we refer to a triple $< S, s, T >$ as a *run* associated with task instance $< S, T >$. Furthermore, a task instance is uniquely *represented* by a run (normally the most recent run, referring to the discussion below).

Note that the concept of the run and that of the task instance described above by virtue model the physical and logical executions, respectively. (This is very similar to the concept of data item and its replicas in a replicated database.) With the introduction of run, the meaning of SFT-dependency defined in chapter 3 should be re-explained. The definition then was said to express the 'immediate follow up' relations between two task instances in a given normal execution. However, since the triples represent runs instead of task instances now, we would refer to SFT-dependency on runs. It expresses the 'immediate follow up' relations between two

runs in the workflow execution involving possibly multiple normal executions. With a given normal execution, the original expression still makes sense because every task instance has only one run associated with it. Without a specific normal execution, only the new expression is correct. This is because a run is unique in the workflow execution, and there is only one normal execution corresponding to it.

4.2.2 Redirected execution

Having defined the concept of peer, we are now at a position to formalize a key concept in backward ad-hoc recovery, *backward redirected execution.*

Definition 4.2 (Backward redirected execution) *A backward redirected execution is a pair* ($<\epsilon>$, NE_0) *or* ($<NE_0,...,NE_{n-1}>$, NE_n) *where* $n \geq 1$, $\forall e \in Min(V(NE_n))$, *there is an* $f \in V(NE_i)$ *for some* i, $0 \leq i \leq n-1$, *such that* f *is a peer of* e. *For all* $0 \leq i \leq n$, NE_i *is said to be generated by the ith redirection.*

In fact, a redirected execution is still a normal execution. But since it is generated by redirections, we give it a new name in the backward recovery model. We assume that the 0th redirection can be viewed as the start of the workflow execution. Thus NE_0 denotes the very first normal execution after the workflow starts. Any other normal executions must start from re-executions of some task executions. This simulates the fact that they are caused by exceptions that redirect the control flow back to some task instances that were already finished.

Note that this definition implies that redirected executions take place in sequence. This is often the case in reality since the exceptions that interrupt the normal control flow often come in sequence. Even thought the exceptions may occur concurrently, we can still handle them one at a time. Thus the idea of sequential executions is

justified. On the other hand, in real applications, a normal execution generated as a result of a redirection is related in some way to those before the redirection. This relation is described by the following

Definition 4.3 (Well defined sequence of redirected executions) $S_u = R_0, \ldots, R_n$ *is a well defined sequence of redirected executions, where* $R_0 = (<\epsilon>, NE_0)$, $R_i = (<NE_0, \ldots, NE_{i-1}>, NE_i)$ *for* $1 \leq i \leq n$, *if* $\forall (e_2 \rightarrow e_1) \in E(NE_i)$ *where* $0 \leq i \leq n$, *and* $e'_1 \in V(NE_j)$ *where* e'_1 *is a peer of* e_1 *and* $0 \leq j < i$, *the following conditions are met:*

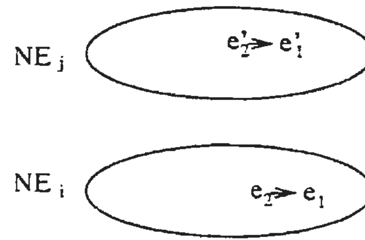
1. *if* $e'_1 \in V(NE_j)$-$Min(V(NE_j))$, *then there exists* $(e'_2 \rightarrow e'_1) \in E(NE_j)$ *where* e'_2 *is a peer of* e_2;
2. *if* $e'_1 \in Min(V(NE_j))$, *there exists a* $e'_2 \in V(NE_k)$ *where* e'_2 *is a peer of* e_2 *and* $0 \leq k < j$;

A well defined sequence of redirected executions requires that the SFT-dependencies be consistent before and after a redirection if they involve peers. The two conditions describe the two cases where the consistency must be preserved. These two cases are exemplified in Figure 4.2.

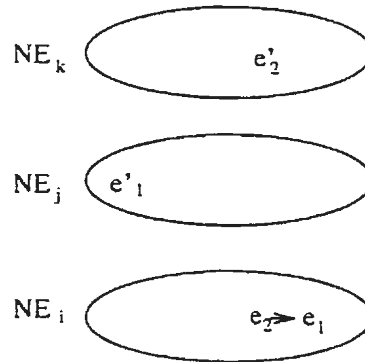
In Figure 4.2, if e_1 in NE_i has a peer e'_1 in NE_j, and e'_1 is not a minimal vertex in NE_j, its SFT-parent e_2 should also have a peer in NE_j. If e'_1 is a minimal vertex in NE_j, there exists a peer e'_2 of e_2 in NE_k.

Note that although the above definition implies a total order on redirected executions, we do not require the individual task executions in these redirected executions also to follow the same order, since this cannot always be guaranteed in real applications. Consider the scenario depicted in Figure 4.3.

In Figure 4.3, it is possible that run c is executed after run d' even though it belongs to an earlier normal execution. However, we observe that orders do exist on certain runs in a well defined sequence of redirected executions. Still consider Figure 4.3. d' is executed after d and e.



Constraint 1



0<=k<j<i

Constraint 2

Figure 4.2: Well defined sequence of backward redirected executions

To establish a similar order in general case, we first introduce the concept of *general descendent.*

Definition 4.4 (General descendent) *In a well defined sequence of redirected executions $R_0,...R_n$,*

1. *$\forall$ $e_1,e_2 \in V(NE_i)$, $0 \leq i \leq n$, e_2 is a general descendent of e_1 if e_2 SFT*-depends on e_1.*

2. *For any two elements e_1 and e_3, e_3 is a general descendent of e_1 if there exist e_2 and e'_2, such that $e_2 \in V(NE_i)$, $e'_2 \in V(NE_j)$, e'_2 is a peer of e_2, $0 \leq i < j \leq n$, e_2 is a general descendent of e_1, and e_3 is a SFT*-descendent of e'_2.*

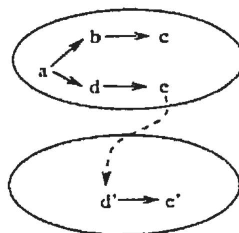


Figure 4.3: Orders in well defined sequence of backward redirected executions

If e_2 is e_1's general descendent, e_1 is e_2's *general ancestor*. In Figure 4.3, d' is the general descendent of a and d. e' is the general descendent of a, d and e.

Among all the general descendents of certain run e_1, there are some which have no general descendents and have not been affected by redirections. These runs are currently active or terminated and they are the far-most runs to which the effects of e_1 may have been propagated. They are called the *least general descendents* of e_1 in our model.

Definition 4.5 (Least general descendent) *In a well defined sequence of redirected executions S_u, e_2 is e_1's least general descendent if it is e_1's general descendent and there does not exist e_3 which is e_2's general descendent in S_u.*

When a redirection occurs that interrupts a normal execution, some current task executions will not be followed by their normal SFT-descendents, but rather by the repetitions of some previous task executions. The orders induced by the redirection are termed *RD-orders*.

Definition 4.6 (RD-order) *In a well defined sequence of redirected executions $R_0, \ldots, R_n$, $\forall e_1 \in Min(V(NE_i))$ where $0 \leq i \leq n$, if e'_1 is a peer of e_1, $e'_1 \in V(NE_j)$, e_2 is a general descendent of e'_1, $e_2 \in V(NE_k)$ where $0 \leq j \leq k < i$, then e_1 RD-follows e_2, denoted as $e_2 \prec_{RD} e_1$.*

For example, in Figure 4.3, $d' \prec_{RD} d$ and $d' \prec_{RD} e$. The definition requires that when the execution of a task is repeated as a result of redirection, all the general

descendents of the previous execution of that task must have been terminated. This makes sense since once a task is re-executed, all its follow up tasks may be re-executed too. To avoid the inconsistencies, their previous executions must be terminated and rolled back.

Now we have two kinds of orders, the SFT-order and RD-order. Both of them indicate the execution orders in reality. They together can be termed *computational order* in our model.

Definition 4.7 (Computational order ($\prec_c$)) *In a well defined sequence of redirected executions $R_0,\ldots, R_n$, $e_1 \in NE_i$ and $e_2 \in NE_j$, $0 \le i,j \le n$. $e_1 \prec_c e_2$ if either $e_1 \prec_{SFT} e_2$ or $e_1 \prec_{RD} e_2$.*

The transitive closure of the computational order produces paths that correspond to the task executions in a time path in the real world. We want to prove that this transitive closure is a partial order.

Theorem 4.1 *The transitive closure of $\prec_c$, $\prec_c^+$ is a partial order.*

Proof: In order to prove $\prec_c^+$ is a partial order, we prove it is non-reflexive, antisymmetric and transitive. Since it is the transitive closure of $\prec_c$, the transitivity is ensured.

1. Antisymmetric

Suppose $a \in V(NE_i)$, $b \in V(NE_j)$, $i \le j$, and $a \prec_c^+ b$. There exists a sequence $a \prec_c e_1 \cdots \prec_c e_n \cdots \prec_c b$, where $n \ge 0$, $e_1 \in V(NE_{k_1}),\ldots,e_n \in V(NE_{k_n})$. From the definition of $\prec_c$, we have $i \le k_1 \le \cdots \le k_n \le j$. Therefore, $i \le j$. Suppose there also exists $b \prec_c^+ a$. Then we have $j \le i$. That is to say, $i = j$, or the two sequences exist in one normal execution NE_i. That is, there existences a loop between a and

b. This contradicts the property that a normal execution is an acyclic graph. Thus $\prec_c^+$ is antisymmetric.

2. Non-reflexive

Suppose $a \prec_c^+ a$. If $a \prec_c a$, it is symmetric, contradiction. If $a \not\prec_c a$, there exists a sequence $a \prec_c \cdots b \prec_c \cdots a$, $b \neq a$. Therefore, we have two sequences $a \prec_c \cdots \prec_c b$ and $b \prec_c \cdots \prec_c a$. From the definition of $\prec_c^+$, we have both $a \prec_c^+ b$ and $b \prec_c^+ a$. This contradicts with the antisymmetric property. Therefore, $\prec_c^+$ is non-reflexive. $\prec_c^+$ is thus a partial order. □

Theorem 4.2 *For any two elements $e_1 \in V(NE_i)$ and $e_2 \in V(NE_j)$ where $0 \leq i < j$, if e_2 is a peer of e_1, $e_1 \prec_c^+ e_2$.*

Proof:

We assume one of the paths including e_2 in NE_j is denoted as $a_n \rightarrow \cdots \rightarrow a_1$ where $n \geq 1$, $a_1 = e_2$ and $a_n \in Min(V(NE_j))$, where n is a constant. We want to prove the following claim by induction on k where $k \geq 1$:

If there exists a path $b_k \rightarrow \cdots \rightarrow b_1$ where $b_k \in Min(V(NE_j))$, then for any peer b'_1 of b_1 where $b'_1 \in V(NE_u)$ and $0 \leq u < j$, there exists a peer b'_k of b_k where $b'_k \in V(NE_d)$ and $0 \leq d \leq u$ such that b'_1 is a general descendent of b'_k.

Basis: $k = 1$.

$b_1 \in Min(V(NE_j))$. For any peer b'_1 of b_1 where $b'_1 \in V(NE_u)$ and $0 \leq u < j$, there exists a peer b'_1 of b_1 where $b'_1 \in V(NE_d)$ and $d = u$ such that b'_1 is a general descendent of b'_1.

Inductive step: Suppose the claim is true for k. We want to prove it is also true for $k + 1$. Now there is a path $b_{k+1} \rightarrow b_k \rightarrow \cdots \rightarrow b_1$ in NE_j, and $b_{k+1} \in$

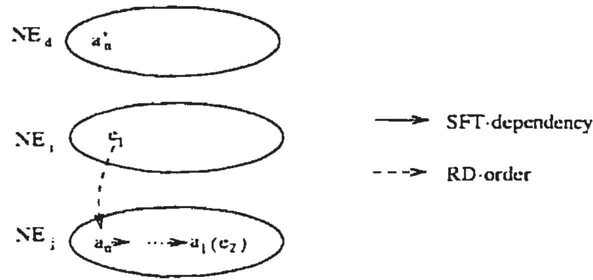


Figure 4.4: Proof of Theorem 4.2

$Min(V(NE_j))$. The following is to be proved: for any peer b'_1 of b_1 where $b'_1 \in V(NE_u)$ and $0 \leq u < j$, there is a peer b'_{k+1} of b_{k+1} and $b'_{k+1} \in V(NE_d)$ for some $0 \leq d \leq u$, such that b'_1 is a general descendent of b'_{k+1}.

We consider the following two cases.

Case 1) $b'_1 \in V(NE_u) - Min(V(NE_u))$

According to the definition of well defined sequence of redirected executions, there must exist a peer b'_2 of b_2 in NE_u, such that $(b'_2 \rightarrow b'_1) \in E(NE_u)$.

For path $b_{k+1} \rightarrow \cdots \rightarrow b_2$ in NE_j, b'_2 is a peer of b_2 where $b'_2 \in V(NE_u)$. By the inductive hypothesis, there exists a peer b'_{k+1} of b_{k+1} in some NE_d where $0 \leq d \leq u$, such that b'_2 is a general descendent of b'_{k+1}. This means there exist c and its peer c' where c is a general descendent of b'_{k+1} and b'_2 is an SFT*-descendent of c'. Since $(b'_2 \rightarrow b'_1) \in E(NE_u)$, b'_1 is also an SFT*-descendent of c'. Thus, b'_1 is a general descendent of b'_{k+1}.

Case 2) $b'_1 \in Min(V(NE_u))$

According to the definition of well defined sequence of redirected executions, there exists a peer b''_1 of b'_1 in NE_x where $0 \leq x < u$. If $b''_1 \in Min(V(NE_x))$, b''_1 has a peer b'''_1, and so on. Every time when a new minimal peer is found, the normal execution

containing it has its subscript be decreased by at least one. Eventually, we should be able to find in some NE_d where $0 \leq d < u$, the peer of b'_1, denoted as B, belongs to $V(NE_d) - Min(V(NE_d))$. Otherwise, we would have $B \in Min(V(NE_0))$, a contradiction to condition 2 of well defined sequence of redirected executions.

Using the similar argument to that in case 1), B is a general descendent of b'_{k+1}. Since B is a peer of b'_1, and b'_1 is an SFT*-descendent of itself, b'_1 is also a general descendent of b'_{k+1}.

We have proved the claim is true for any $k \geq 1$. Thus for any peer e_1 of $e_2 (e_2 = a_1)$ where $e_1 \in V(NE_i)$ and $0 \leq i < j$, there exists a peer a'_n of a_n where $a'_n \in V(NE_d)$ and $0 \leq d \leq i$, such that e_1 is a general descendent of a'_n.

From the definition of RD-order, we have $e_1 \prec_{RD} a_n$. This order together with the SFT*-order from a_n to e_2 form a sequence of computational orders from e_1 to e_2. Hence $e_1 \prec_c^+ e_2$. □

Run e_2 is said to be the *more recent run* compared with e_1.

For easy presentation, we use a triply indexed notation T_{ijk} to denote a run (triple). T_{ijk} represents the k-th run of task instance T_{ij}. The mapping from triples to this notation has the following property. Suppose <S,s',T> is the *next* peer of <S,s,T>. If <S,s,T> is denoted as T_{ijk}, <S,s',T> is $T_{ij(k+1)}$. In general, T_{ijk_1} and T_{ijk_2} $(k_1 < k_2)$ serve the same purpose from the viewpoint of the workflow schema, but only T_{ijk_2} counts. T_{ijk_2} is a more recent run compared with T_{ijk_1}. Refer to the order processing example, the two executions of *prepare-order* are different runs of the same task instance. However, as far as the workflow schema is concerned only the second order is valid.

4.2.3 Redirect-to (RDT) and redirect-from (RDF) sets

As can be seen from the previous example, when backward ad-hoc recovery occurs, the control flow is redirected back. But the backward redirected executions introduced in our model is too general to describe ad-hoc recoveries. From its definition, any normal execution beginning with a set of peers, or the first runs of the task instances, is called a backward redirected execution. There are no other conditions in this definition. We further add a constraint on the SFT-dependencies in redirected executions if peers are involved. Thus the well defined sequence of redirected executions are generated to model the preservation of the SFT-dependencies during recovery in reality. This constraint is still not strong enough to characterize the ad-hoc recovery in the aspect of maintaining the correctness of the workflow execution. This aspect cannot be reflected only by the well defined sequence of redirected executions. Some other constraints on the runs before and after the redirections are needed. These constraints will be discussed in this section.

Many runs are affected during an ad-hoc recovery. Their executions are interrupted and rolled back. The range of affected runs can be given by two sets, the set of the runs the control is redirected to and that it is redirected from. We call them RDT (ReDirect-To) set and RDF (ReDirect-From) set (hencefore called RDT and RDF for simplicity). All the minimal vertexes in the redirected execution form RDT. It is a set of peers. Formally, RDT can be defined as follows.

Definition 4.8 (RDT) *For a redirected execution* $R = (< NE_0, \cdots NE_{i-1} >, NE_i)$, $RDT = Min(V(NE_i))$.

RDF is a set containing the most recent runs before the redirection. It can be defined as:

Definition 4.9 (RDF) *For a redirected execution* $R = (< NE_0, \cdots NE_{i-1} >, NE_i)$, $RDF \subseteq \bigcup_{j=0}^{i-1} Max(V(NE_j))$

The re-execution starts from runs in RDT, and continues according to the paths consistent with the SFT*-dependencies. The redirected execution containing RDF occurs before that containing RDT in a well defined sequence of redirected executions. For example, consider the ad-hoc recovery in the order processing workflow mentioned before. The redirection is from the most recent run of *send-to-supplier* to a new run of *prepare-order*. RDT is the set containing only the peer of *prepare-order*, and RDF is the set containing only the most recent run of *send-to-supplier*. For simplicity, in the remaining part of this example we directly use task instances to describe the memberships of RDT and RDF, with the understanding that the runs are involved implicitly. We would like to point out that RDT and RDF should not overlap. The reason is that a run cannot be both redirected to and redirected from.

The concept of RDT and RDF is important since it is de facto of the soundness of the semantics of ad-hoc recovery. An ill-defined concept not only makes the semantics awkward, but also directly affects the efficiency of the implementation. Consider again the above ad-hoc recovery scenario, but this time we assume that after the shipping company has been chosen but before the payment is made, some events occur that renders it being necessary to choose a new one. The question we would like to ask is what should be the RDT and the corresponding RDF. If RDT remains the same, i.e. $\{$*prepare-order*$\}$, then we should not include also *choose-shipping-company* to the corresponding RDF, even if presumably the shipping company must be reselected. This is because the re/execution of *prepare-order* has no effect on selecting a new shipping company intended by such an inclusion. On the other hand,

if RDF remains the same, i.e. {*send-to-supplier*} then including *choose-shipping-company* to RDT would be erroneous since one of its SFT*-descendents, itself, is the current run which is affected by the ad-hoc recovery. Therefore the *choose-shipping-company* should be included in RDF.

There are six constraints on RDF and RDT in the backward recovery. They can be characterized into four categories as follows.

- **Intra-group independency**

  1. For any two elements T_{ijk} and T_{uvw} in RDT, $T_{ij(k-1)}$ is not a general descendent of $T_{uv(w-1)}$ and vice versa.

  2. For any two elements T_{ijk} and T_{uvw} in RDF, T_{ijk} is not a general descendent of T_{uvw} and vice versa.

  The rationale for condition 1 is that if for example, $T_{ij(k-1)}$ is a general descendent of $T_{uv(w-1)}$ then after the redirection, T_{ij} may have two runs. One is indicated by run T_{ijk}. The other results from $T_{uv(w-1)}$ and the execution path from T_{uv} to T_{ij}. This is not allowed since any task instance can be represented only by one run after each redirection.

  For condition 2, if T_{ijk} is a general descendent of T_{uvw} (or vice versa), the semantic is confusing because restarting the descendent task instance but not its ancestors means all its ancestor's effects are deemed acceptable. However, the effect of T_{uvw}, which is the general ancestor of T_{ijk}, is not acceptable because it appears in the RDF.

- **Inter-group dependency**

3. For each element T_{ijk} in RDF, there is an element T_{uvw} in RDT such that T_{ijk} is a least general descendent of $T_{uv(w-1)}$.

4. For each element T_{uvw} in RDT, there is an element T_{ijk} in RDF such that T_{ijk} is a least general descendent of $T_{uv(w-1)}$.

These two constraints require that any element in RDT (RDF) are related to certain element(s) in RDF (RDT). If there is a T_{ijk} in RDF none of whose general ancestor's peer is in RDT, the recovery will not create a new run of T_{ij} which updates T_{ijk}'s effect. Thus T_{ijk} should not be included in RDF. Nevertheless, such question may be asked what if task T_i is invoked by another ancestor in DG. Remember in our generic workflow model, we assume if a task is an or-joint, invocations from different ancestors are deemed as different task instances. Thus our reasoning is justified.

The same reason works for RDT. For any element T_{uvw} in RDT, if none of its previous run's general descendents is in RDF, then re-executing T_{uvw} will not affect RDF.

- **Least general descendents closure**

In case e_1 is re-executed, its least general descendents should all be stopped to maintain the consistencies. Otherwise, re-executing e_1 may result in duplicate runs of certain general descendent and it is not known which execution represents the task instance. Therefore, we have the following constraint.

5. For each element T_{uvw} in RDT, all of $T_{uv(w-1)}$'s least general descendents are in RDF.

The next constraint will use two new concepts, *family* and *closed family*. Given set R and set S of runs, let set $R_1 \subseteq$R, and set $S_1 \subseteq$S.

Definition 4.10 (Family) *$<R_1,S_1>$ is a family within domain $< R,S>$ if*

1. *$\forall s \in S_1$, $\exists r \in R_1$ (s is r's least general descendent);*
2. *$\forall r \in R_1$, $\exists s \in S_1$ (s is r's least general descendent);*

Definition 4.11 (Closed family) *$<R_1, S_1>$ is called a closed family within domain $< R,S>$ if*

1. *$<R_1,S_1>$ is a family within domain $<R,S>$;*
2. *all the least general descendents, within S, of every element in R_1 are in S_1.*
3. *all the general ancestors, within R, of every element in S_1 are in R_1.*

Theorem 4.3 *If $< R_1, S_1 >$ is a family within domain $< R_1, S_1 >$, it is also a closed family within the same domain.*

The reason of this theorem is that condition 2 and 3 are satisfied automatically because of $R = R_1$ and $S = S_1$.

- **Minimality**

  Given RDT, let $RDT' = \{T_{ij(k-1)} | \forall T_{ijk} \in RDT\}$.

  6. $<RDT',RDF>$ in backward recovery is a minimal closed family within domain $<RDT',RDF>$.

This constraint is intended to preclude the situation depicted in Figure 4.5. In this figure, a subset of RDT' and a subset of RDF form a closed family. $A \subset RDT'$, $B \subset RDF$, $< A, B >$ is a closed family within domain $< RDT', RDF >$. In this

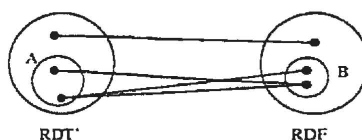


Figure 4.5: Example of non-minimal $< RDT', RDF >$

case, $< A, B >$ corresponds to an ad-hoc recovery and $< RDT' - A, RDF - B >$ corresponds to another recovery. They do not interfere with each other.

'Minimality' requires that an ad-hoc recovery involve only 'related' task executions. That is, RDF should contain only the related runs. So is RDT. More specifically, in RDF (RDT'), all the elements should be related to each other, directly or indirectly, by means of the transitive closure of computational orders via some elements in RDT' (RDF). In this sense, $<RDT',RDF>$ is minimal. To get some concrete idea about this property, let us look at the scenario described at the end of the last section again.

Suppose, when the assumed event occurs *choose-shipping-company* has been finished. But before the payment is made, a strike occurs in the selected shipping company. Thus the control flow should be redirected not only from *send-to-supplier* to *prepare-order*, but also from *choose-shipping-company* to itself. It may seem that RDF must contain both *send-to-supplier* and *choose-shipping-company*, and RDT' contains both *prepare-order* and *choose-shipping-company*. We observe, however, that *prepare-order* has no effect on selecting a new shipping company, since the latter does not SFT*-depends on the previous run of the former in the normal execution. Likewise, *choose-shipping-company* has no effect on *send-to-supplier*, due to the similar reason. We say that the two elements in RDF are not related to each other by

means of SFT*-dependency via any element in RDT'. The same can be said to the two elements in RDT'. The point here is that the RDT and RDF defined above do not satisfy the minimality of $< RDT', RDF >$ within domain $< RDT', RDF >$.

4.2.4 Execution graph (EG)

We can simplify the above concepts and constraints using a graph. It is termed the *execution graph (EG)* in our model.

Definition 4.12 (Execution graph) *An execution graph is a directed acyclic graph each of whose vertexes is a run and each arc is either a consistent arc or a backward inconsistent arc.*

Definition 4.13 (Consistent arc) *Consistent arc* $T_{ijk} \rightarrow T_{uvw}$ *indicates* $T_{ijk} \prec_{SFT} T_{uvw}$. *A path is called a consistent path if it contains only consistent arcs.*

A redirected execution is a subgraph of EG containing only consistent arcs and associated vertexes. A consistent arc represents the normal invocation of a task. Similarly we would like to define a backward inconsistent arc to represent the abnormal invocation of a task, or a redirection. We note that the redirections have already introduced the RD-orders into the model. The RD-order between a least general descendent and a new run of the ancestor can be used to represent a redirection. This order is abstracted into a new concept, the *direct RD-order* ($\prec_{DRD}$).

Definition 4.14 (DRD-order) *In a well defined sequence of redirected executions* S_u, $\forall e_1 \in Min(V(NE_i))$ *where* $0 \leq i \leq n$, *if* e'_1 *is the immediately preceeding run of* e_1, $e'_1 \in V(NE_j)$ *where* $0 \leq j < i$, e_2 *is a least general descendent of* e'_1, $e_2 \in V(NE_k)$ *where* $j \leq k < i$, *then* e_1 *DRD-follows* e_2, *denoted as* $e_2 \prec_{DRD} e_1$.

Definition 4.15 (Backward inconsistent arc) $T_{ijk} \rightarrow T_{uvw}$ *is a backward inconsistent arc if* $T_{ijk} \prec_{DRD} T_{uvw}$.

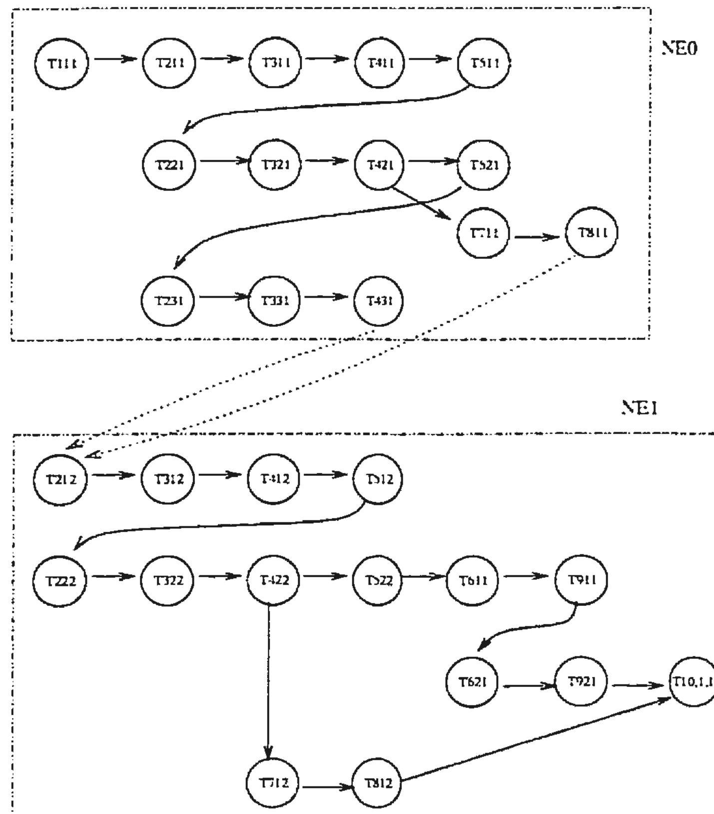


Figure 4.6: An example EG

Figure 4.6 is an example EG whose DG is Figure 3.1. The solid lines represent consistent arcs, for example, $T_{111} \rightarrow T_{211}$; The dotted lines represent backward inconsistent arcs, for example, $T_{431} \cdots > T_{212}$. There are two normal executions in Figure 4.6, NE_0 and NE_1. T_{512} is a general descendent of T_{211} but not a least general descendent of T_{211} because T_{222} is a general descendent of T_{512}. The least general descendents of T_{211} include T_{431}, T_{811}, and $T_{10,1,1}$. T_{212}, which is the peer of T_{211}, directly RD-follows T_{431} and T_{811}, the two least general descendents of T_{211}.

Properties of EG

1. Each minimal vertex of EG is the first run of a starting task.

2. Each maximal vertex of EG is either a run of a terminated task or an active run of some task instance.

3. Each minimal vertex of a redirected execution is either a minimal vertex of EG or the destination of a backward inconsistent arc.

4. Each maximal vertex of a redirected execution is either a maximal vertex of EG or the source of a backward inconsistent arc.

5. The source of a backward inconsistent arc is a maximal vertex of a redirected execution. The destination of a backward inconsistent arc is a minimal vertex of a redirected execution.

6. For any inconsistent arc $T_{ijk} \rightarrow T_{uvw}$, there is a $T_{uv(w-1)}$ and T_{ijk} is $T_{uv(w-1)}$'s least general descendent.

4.2.5 *RDF* and *RDT* in terms of *EG* in backward recovery

EG introduced above does not involve the six constraints on RDT and RDF. It is the graphical representation of other concepts given in the model. EG makes most of the contents in the model clear and straightforward. In this section, we will incorporate the six constraints on RDT and RDF into EG in a simplified format. The new statements in terms of EG are easier to understand and more applicable.

First of all, let us explain what RDT and RDF are in terms of EG. As mentioned before, backward inconsistent arcs represent redirections. The source/destination of an inconsistent arc is a member of RDF/RDT. During an ad-hoc recovery, there is

a set of redirections which generates a group of inconsistent arcs. The vertexes of this group of inconsistent arcs form RDF and RDT. In order to avoid the overlap of RDT and RDF, the group of inconsistent arcs should not form any path whose length is greater than 1. The counterpart of each previous constraint in EG is as follows.

- **Intra-group independency**

  1. For any two elements T_{ijk} and T_{uvw} in RDT, $T_{ij(k-1)}$ is not a general descendent of $T_{uv(w-1)}$ and vice versa.

  2. For any two elements T_{ijk} and T_{uvw} in RDF, T_{ijk} is not a general descendent of T_{uvw} and vice versa.

  We observe that this constraint is satisfied automatically if two inconsistent arcs have different destinations. Because both source vertexes are the least general descendents of the same element, none of them is a general descendent of the other.

- **Inter-group dependency**

  3. For each element T_{ijk} in RDF, there is an element T_{uvw} in RDT such that T_{ijk} is a general descendent of $T_{uv(w-1)}$.

  4. For each element T_{uvw} in RDT, there is an element T_{ijk} in RDF such that T_{ijk} is a general descendent of $T_{uv(w-1)}$.

  These two constraints are automatically satisfied in EG due to properties of EG.

- **Least descendents closure**

  5. For each element T_{uvw} in RDT, all of $T_{uv(w-1)}$'s least general descendents are in RDF.

  This constraint should be re-iterated in terms of EG as follows. *If T_{uvw} is the destination of an inconsistent arc, all of $T_{uv(w-1)}$'s least general descendents are the sources of inconsistent arcs leading to T_{uvw}.*

- **Minimality**

  Given RDT, let $RDT'=\{T_{ij(k-1)} | \forall T_{ijk} \in RDT\}$.

  6. $<RDT',RDF>$ in backward recovery is a minimal closed family within domain $<RDT',RDF>$.

Because the inter-group dependency constraint is met, $< RDT', RDF >$ generated by inconsistent arcs is a family within $< RDT', RDF >$. According to theorem 4.3, $< RDT', RDF >$ is a closed family within $< RDT', RDF >$. To be minimal, it should form a connected [4] subgraph of EG. Formally, we have the following theorem:

Theorem 4.4 *Given a redirected execution $R = (< NE_0, \cdots NE_{i-1} >, NE_i)$ and RDT, RDF on R, assume the first five constraints are satisfied by RDT and RDF. Let BEG be a subgraph of EG. Each destination vertex of arcs in BEG is in RDT, and each source vertex of arcs in BEG is in RDF. $< RDT', RDF >$ is a minimal closed family within $< RDT', RDF >$ if and only if BEG is connected.*

[4] A direct graph is connected if there is no isolated part in it.

Proof:

1. If:

Since RDT and RDF satisfy the inter-group dependency constraint, $< RDT', RDF >$ is a family within $< RDT', RDF >$. From theorem 4.3, it is also a closed family. Next, we prove its minimality. That is, any proper subset of RDT' and any subset of RDF cannot form a pair which is a closed family. The following three cases cover all the possibilities that the pair can be formed by means of subsets. If we can prove none of them is a closed family within $< RDT', RDF >$, the minimality of $< RDT', RDF >$ is ensured.

a) $< A, RDF >$ where $A \subset RDT'$.

b) $< RDT', B >$ where $B \subset RDF$.

c) $< A, B >$ where $A \subset RDT'$, $B \subset RDF$.

In case a), at least one ancestor of some element in RDF is not included in A. This missing ancestor is within RDT'. Thus $< A, RDF >$ cannot be a closed family within $< RDT', RDF >$.

Symmetrically, case b) can be proved. Compared with case a), there is at least one lease general descendent missing in B for some element in RDT'.

In case c), since BEG is connected, there exists at least one element in A which has least general descendent(s) in $RDF - B$ and there exists at least one element in B which has ancestor(s) in $RDT' - A$. (Otherwise BEG is not connected.) If any of them is true, $< A, B >$ is not a closed family within $< RDT', RDF >$.

2. Only if:

We prove the following statement: *If BEG is not connected, $< RDT', RDF >$ is*

not a minimal closed family within $< RDT', RDF >$.

If BEG is not connected, it has isolated parts each of which is a connected subgraph of BEG. Take one of such connected subgraphs G for consideration. Note that, G is a bipartite graph. Suppose the source vertexes of G is set B and the destination vertexes of G is set A. We have $< A, B >$ where $A \subset RDT$ and $B \subset RDF$. Let A' be the set of last runs of elements in A. $A' \subset RDT'$. This pair of $< A', B >$ can be proved to be a closed family within $< RDT', RDF >$.

$\forall e' \in A'$, there is an $e \in A$, which is its next run. From the way backward inconsistent arcs are created and the connectivity of G, e' has a least general descendent in B. Similarly, $\forall e \in B$, there is an $f' \in A'$, such that e is a least general descendent of f'. Therefore, $< A', B >$ is a family within $< RDT', RDF >$.

Furthermore, $\forall e' \in A'$, all of its least general descendents within RDF are in B because G is isolated from other subgraphs in BEG. $\forall e \in B$, all of its general ancestors within RDT' are in A'. Hence $< A', B >$ is a closed family within $< RDT', RDF >$.

Since $A' \subset RDT'$ and $B \subset RDF$, $< RDT', RDF >$ is not a minimal closed family within $< RDT', RDF >$. □

The six constraints on RDT and RDF in terms of EG facilitate the users to determine whether a group of inconsistent arcs is the result of an ad-hoc recovery. For example, the two inconsistent arcs in Figure 4.7 do not belong to one ad-hoc recovery. This can be detected immediately since they are not connected with each other. In fact, the DG in Figure 4.7 is the abbreviation of the order processing workflow. T_1 is the *choose-supplier* task and the others follow T_1 correspondingly. Each of the two

inconsistent arc represents the redirection of the two ad-hoc recoveries explained in section 4.2.3 respectively.

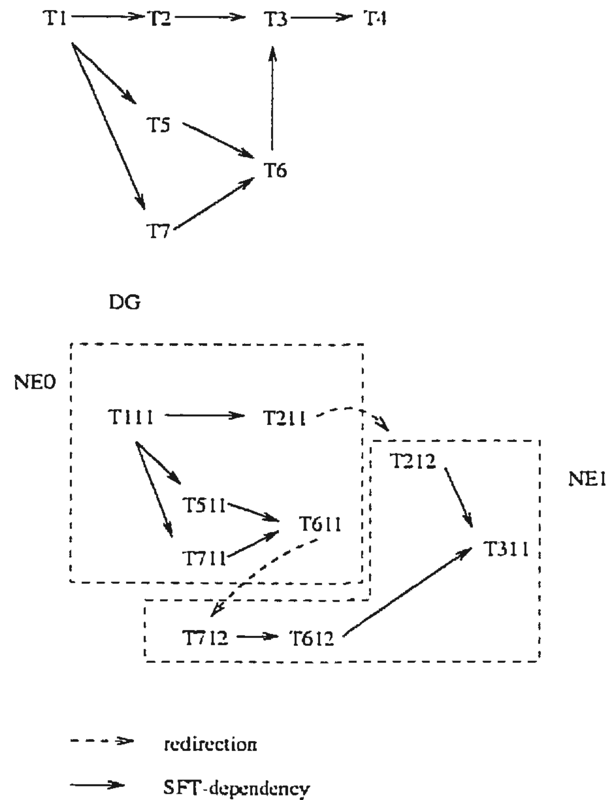


Figure 4.7: Constraints on *RDT* and *RDF* in terms of *EG*

On the other hand, if the *EG* is used as a tool to keep track of the workflow execution, it can help the users to choose the right set of tasks to be re-executed during an ad-hoc recovery. In case the constraints are violated on the chosen *RDT*, a controller should be able to find the right *RDT* and *RDF* for the recovery.

Chapter 5

A WFMS prototype supporting backward ad-hoc recovery

The purpose of prototyping a workflow management system is two fold. First, it stresses the redesign of the key components in the WFMS in order to support ad-hoc recovery. Second, it demonstrates an application of the model with a hospital workflow. A protocol is given to facilitate the cooperation among WFMS components during the ad-hoc recovery. The architecture of the prototype system is open, extensible, and feasible.

Our efforts are mainly made in the following four directions. a) Extend the workflow specification proposed by the WfMC to provide ad-hoc properties for tasks; b) Extend the functionalities of the workflow server for ad-hoc recovery; c) Define a standard task manager user interface to provide ad-hoc recovery related operations; And d) propose a backward ad-hoc recovery protocol. These contents will be discussed in the following sections.

5.1 Design and runtime representation

Table 5.1 and 5.2 are the snapshots of the workflow definition generated by the Graphical Workflow Designer in the prototype. Table 5.3 is the worklist generated at run time. These three tables contain the key workflow control data. Table 5.4 is used for role management. Account of each agent is maintained in this table, including the user name, password and role mapping. It should be generated by the role management tool which is not in the scope of this prototype. The meaning of each field in the tables and the relations among them are as follows.

Table 5.1: Task specification

w-id	t-id	name	type	AH-property	host	join	role	input	output	x	y
01	01		Start							48	89
01	02	Register	NT	undoable	dove		register			131	91
01	03	Nurse	NT	undoable	lark		nurse		flag	212	93
01	04	Doctor	NT	undoable	eagle		doctor			293	218
01	05	Payment	T	undoable	garfield	or	cashier			383	94
01	06		Stop							465	96

Table 5.2: Inter-task dependencies specification

from-task	to-task	condition	anchor-x	anchor-y	end-x	end-y
0101	0102		86	85	136	85
0102	0103	done	173	86	220	86
0103	0104	done and flag=1	237	107	305	203
0103	0105	done and flag=0	256	86	339	86
0104	0105	done	344	206	407	108
0105	0106	commit	427	85	485	85

Task specification

Table 5.3: Run time worklist

w-id	wi-id	t-id	ti-id	name	agent	state	status	oper	output	host	get	pickup
01	01	02	01	Tom	reg1	Done	pre-AH			dove	0	0
01	01	03	01	Tom	nur1	Done	pre-AH		1	lark	1	1
01	02	02	01	Mike	reg1	Done	pre-AH			dove	0	0
01	02	03	01	Mike	nur1		pre-AH			lark	0	0
01	01	04	01	Tom	doc1	Active	pre-AH			eagle	0	1
01	01	03	01	Tom	nur1	Done	in-AH	undo			1	1
01	01	03	01	Tom	nur1	Done	post-AH	redo	1		1	1
01	01	04	01	Tom	doc1	Done	in-AH	undo			1	1
01	01	04	01	Tom	doc1	Done	post-AH	redo			1	1
01	01	05	01	Tom			pre-AH			garfield	0	0

- **Workflow identifier** (w-id) identifies different workflows.
- **Task identifier** (t-id) identifies different tasks in one workflow. Two tasks in a workflow have different t-id.
- **Task type**

  1. Transactional task (T)

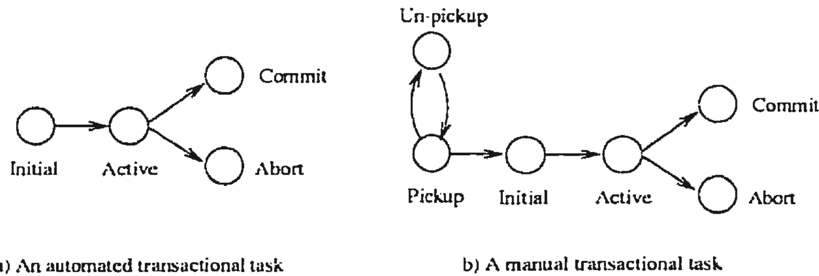


Figure 5.1: Structures of transactional tasks

Transactional and the non-transactional tasks are the two smallest atomic units of activity each of which forms one logical step within a process.

There are two kinds of transactional tasks: automated ones and manual ones. An automated transactional task can be performed by the computer directly. Its states are changed automatically from *Initial* to *Active* then *Commit* or *Abort*. A manual transactional task has to be picked up before it is initiated. Structures of the two kinds of transactional tasks are shown in Figure 5.1(a) and Figure 5.1(b). A typical transactional task is a database update which satisfies the ACID properties. The recovery of a transactional task is done by the system.

2. Non-transactional task (NT)

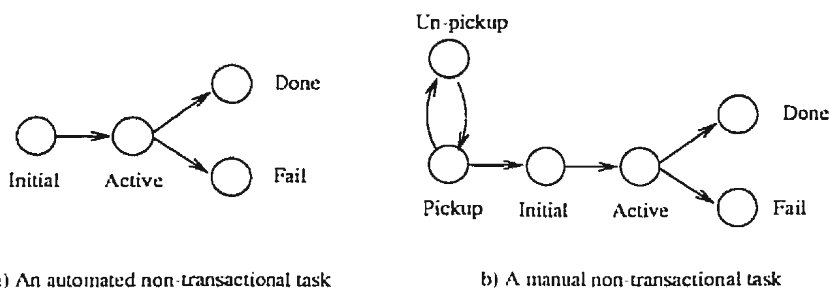


Figure 5.2: Structures of non-transactional tasks

The only difference between the structure of this kind of tasks and that of the manual transactional tasks is that the terminating states are changed to *Done* and *Fail* respectively. The non-transactional task can either be an automated one or a manual one. A typical example maybe a nurse checking a patient's pulse and record the data in a MS Word file. It does not have any transactional properties and hence requires some human assistance on failure.

3. Start icon (Start)

If there is an edge from the start icon to a task in the workflow map, that task is a starting task of the workflow. Usually there is only one start icon in the workflow map. When a workflow instance is created, new task instances of the workflow instance will be created for those starting tasks. Start icon occupies a row in table 5.1. But only five fields are used, namely, w-id, t-id, type, x and y.

4. Stop icon (Stop)

   If there is an edge from the stop icon to a task in the workflow map, that task is a terminating task. There maybe a condition associated with the edge. The combination of all of these conditions forms the terminating condition of the workflow. Like the Start icon, stop icon occupies a row in table 5.1.

- **AH-property**

  This property of a task relates to the actions that can be taken during the ad-hoc recovery of a task. Its value can be one of the following: {*undoable*, *comp (compensatable)-for-redo*, *comp-for-undo*, *redoable*, *null*}.

  1. undoable[1]: undesirable effects of the task can be eliminated as if the task has never been executed.
  2. comp-for-undo: undesirable effects can be eliminated semantically.
  3. comp-for-redo: desirable effects can be generated semantically.

[1] Undo here means the elimination of the effects of a task. It captures a wider spectrum than the meaning in the transaction processing.

4. redoable: any task instance can be repeated.

5. null: none of the above.

The AH-property does not depend on the task type. That is to say, a transactional task and a non-transactional task can have the same value of AH-properties. If a task is comp-for-undo(redo), a compensating task should be specified. AH-properties are used to validate the decisions made by the agents during the ad-hoc recovery.

- **Host** specifies where the task can be executed.

- The value of **Join** can be of $\{and,\ or,\ null\}$. 'And'('or') corresponds to the and-joint(or-joint) in the generic workflow model. If the value is null, it implies that the indegree of the task node is 1, or the task has only one precedent task. This precedent task must be finished successfully in order to invoke it.

- **Role**

  Role is a conceptual categorization of agents. It comes from the organizational management in an enterprise. The use of role simplifies the control of the accessibility rights to tasks. For example, suppose Jane and Mary are both nurses in a hospital, and role 'nurse' is allowed to perform a nurse task. Thus Jane and Mary both get the access rights to the nurse task. Another advantage of using role is that work can be allocated to agents dynamically. If Jane takes a leave for a few hours, some of her work items can be transfered to Mary.

- **Input** and **output** hold data objects' names or identifiers. Input data objects

Table 5.4: Role

w-id	uname	passwd	role
01	reg1		register
01	nur1		nurse
01	doc1		doctor

should be available before the task is invoked. Values of output data objects should be set before the task terminates successfully.

- Symbol **x** and **y** are the coordinates of the top left corner of the task icon drawn on the designer's canvas. These two fields together with the task type help to redraw a task icon on canvas when the workflow definition is loaded into the Graphical Workflow Designer.

Inter-task dependencies

The inter-task dependencies are stored in another table since it is a set of data relatively independent of the task specification. Each row of Table 5.2 represents an edge(inter-task dependency) between two tasks in the workflow map.

- **From-task** and **to-task** refer to the source and destination tasks of an edge. Their values are the concatenation of the values of w-id and t-id. The modification of w-id and t-id in Table 5.1 will change from-task and to-task in Table 5.2 automatically in the graphical workflow designer.
- The **condition** field is a logical expression associated with an edge.
- **Anchor-x** and **anchor-y** are the x-coordinate and the y-coordinate of the

source task. **End-x** and **end-y** are the x-coordinate and the y-coordinate of the destination task.

Run time data

While Table 5.1 and 5.2 concern the workflow definition, Table 5.3 and 5.4 store the workflow run time data. We extend the concept of worklist mentioned in WfMC to refer to task instances in the prototype system. In fact, the worklist in WfMC is only a subset of ours.

Worklist

- **Workflow instance** is represented by <w-id, wi-id>. The value of wi-id is generated automatically when a new workflow instance is created.

- **Task instance** is represented by <t-id, ti-id>. Once a new instance of a task is created, the value of ti-id is incremented.

  Tuple <w-id, wi-id, t-id, ti-id> identifies a unique task instance in workflows.

- **Name** is the name of the task instance (work item). Users recognize a work item from its name, which is translated into the actual identifier, <w-id, wi-id, t-id, ti-id> by the system.

- **Agent** is the login name taken when the agent logs onto the system. Each agent is responsible for processing his/her chosen work items, during both normal scheduling and ad-hoc recovery.

- **State** corresponds to the nodes in task structures (Figure 5.1 and 5.2) except pickup and un-pickup. Its value can be taken from {Initial, Active, Done,

Commit, Fail, Abort}. The pickup and un-pickup states are reflected by the 'pickup' field. (Refer to the **Pickup** part for more detail.)

- **Status** can be *pre-AH*, *in-AH*, or *post-AH*. When a work item is initialized for the first time in a normal execution, its' status is pre-AH. During an ad-hoc recovery, if the work item is being undone, its status is in-AH. After the undo procedure, its status is post-AH.

- **Output** keeps a list of data objects generated by a task instance.

- **Host** specifies the computer where the task instance is executed. The host must be one of those specified in the field 'host' of the task specification table (Table 5.1).

- **Get** is a flag indicating whether a work item has been fetched into the task manager window or not. Once $get = 1$, the item will not be fetched again. Therefore, this field helps to allocate work items among several agents of the same role. After an item is done or committed, this field will keep the value 1. When an agent logs off, all his/her unpicked items will be returned to the worklist for re-allocation by means of setting $get = 0$.

- **Pickup** is a flag indicating whether a work item has been picked up or not. The pick up operation is only provided for the Todo list.

There are three types of work items in the worklist: *pre-AH* work items, *in-AH* work items, and *post-AH* work items. Pre-AH work items can appear in the Todo list and the Done list. In-AH work items can only appear in the Todo list. Post-AH

work items appear in the Redo list or Comp-for-redo list, depending on the ad-hoc recovery decision. If the decision is <null, comp-for-redo>, the work items will be put into the Comp-for-redo list. Otherwise it will be put into the Redo list.

Decision

Decision is a table storing the recovery decisions made by the agents. Each work item involved in the ad-hoc recovery has an entry in the decision table. Agents are responsible for undoing(redoing) work items that they picked up. The decision table is shown in Table 5.5. The last field 'leaf' indicates whether an item is a least general descendent or not.

Table 5.5: Decision

w-id	wi-id	t-id	ti-id	tname	state	iname	agent	in-AH-oper	post-AH-oper	flag	leaf
01	01	03	01	Nurse	Done	Tom	nur1	undo	redo	1	0
01	01	04	01	Doctor	Active	Tom	doc1	undo	redo	1	1

5.2 Outline of a client-server architecture supporting ad-hoc recovery

Shown in Figure 5.3 is an architecture of the WFMS which supports ad-hoc recovery. It can be separated into build time and run time parts each of which conform to client-server computing architecture. The build time part includes a Graphical Workflow Designer (client) and the Workflow Server. The run time part is a nested client-server architecture. The higher level is the Task Manager(client)-Workflow Server.

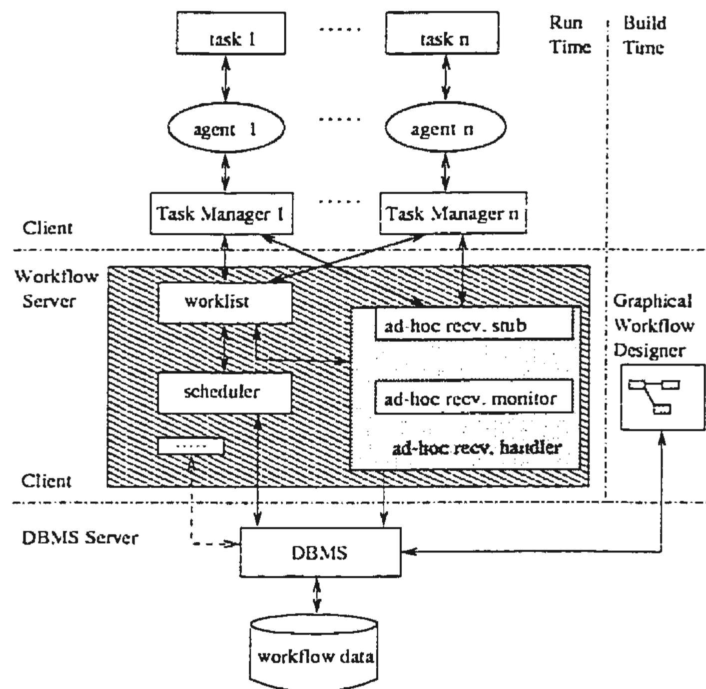


Figure 5.3: A client-server architecture supporting ad-hoc recovery

The lower level is the Workflow Server(client)-DBMS(server). The Workflow Server provides services for Task Manager's requests. There are mainly two sets of services, normal scheduling and the ad-hoc recovery, handled by the scheduler and the ad-hoc recovery handler, respectively. Key components in the architecture will be introduced in section 5.3.

Client-server is a software architecture in which one set of software components (the clients) use messages to ask another set of software components (the servers) to do things. The servers carry out the required actions and return their results to the clients, again using messages. Both the clients and the servers send their messages not using addresses, but instead using names. The clients, in particular, send their requests to named services rather than to specific machines, relying on some form

of *name resolution* to determine the physical server to be used. A breakdown (see Figure 5.4) was proposed by the Gartner Group to show the variety of ways in which the workload can be divided between the client and the server [15].

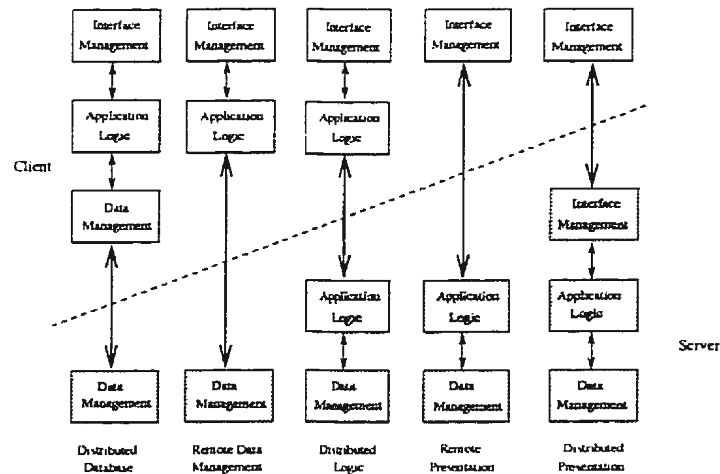


Figure 5.4: Types of client-server computing

The build time architecture belongs to the remote data management type. The DBMS server only deals with the data management, and the Graphical Workflow Designer (client) handles everything else. This includes the interface management and the issues related with workflow definition. In the run time part, the two client-server levels both belong to the distributed logic type. Besides the interface management, the clients have some intelligence of the workflow enactment such as where to put the work items and how to invoke the real task, etc..

The infrastructure of the prototype system is shown in Figure 5.5. Graphical Workflow Designer uses a graphic tool, Java AWT(Abstract Window Toolkit), to generate a workflow map. While the map is drawn, information about tasks and inter-task dependencies are extracted into a database. The communication between

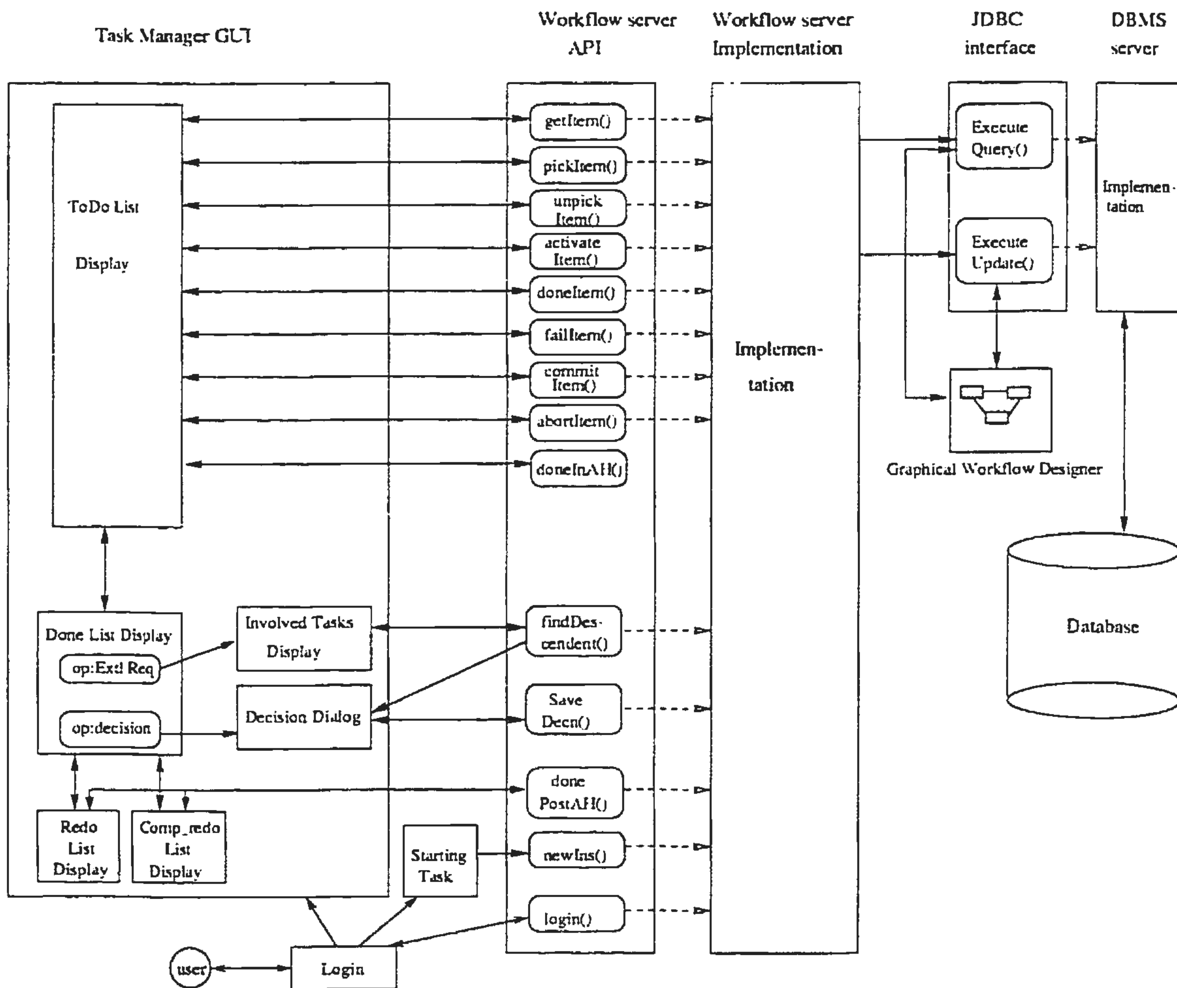


Figure 5.5: The infrastructure of the prototype

AWT and the database is through JDBC™ (Java Database Connectivity).

There may exist multiple Task Managers for one user task. They all have the standard interfaces. The Task Manager interface contains four list displays each of which is responsible to manipulate work items in different status and state. Items move from list to list. Operations are grouped and attached to each list. These operations once performed require services from the Workflow Server through Java RMI™ (Remote Method Invocation). Task Manager is not the real task. It can either

invoke an automated task or report to the workflow that a manual task is activated.

Workflow Server provides services to task Managers by means of a set of APIs and their implementations. It is a registered object and bound to a unique name. Task Managers locate Workflow Server from its name. If the Scheduler and Ad-hoc Recovery Handler are separated into two registered objects, they have different names and contain different APIs. Since Task Managers do not access the database directly at run time, the data integrity is maintained by the Workflow Server.

DBMS Server implements JDBC APIs to provide Java applications access to databases. The JDBC APIs are provided by JDK(Java Development Toolkit). The DBMS Server we use is a commercial product which implements the JDBC APIs.

5.3 Components

5.3.1 Graphical workflow designer

Figure 5.6 is the screen snapshot the the Graphical Workflow Designer. It provides basic drawing functions like creating, modifying, and deleting drawing objects. Central to the designer is a canvas. All the drawing objects are listed in the toolbar to the left of the canvas. In our prototype, the toolbar includes start icon, stop icon, transactional task icon, non-transactional task icon, and the arrow icon. By clicking the left mouse buttons on an object in the toolbar and dragging it onto the canvas, a task or an inter-task dependency will be created. Clicking the middle mouse button on an object will delete it both from the screen and the database. When clicking the right mouse button on tasks or edges, two different kinds of windows will be popped

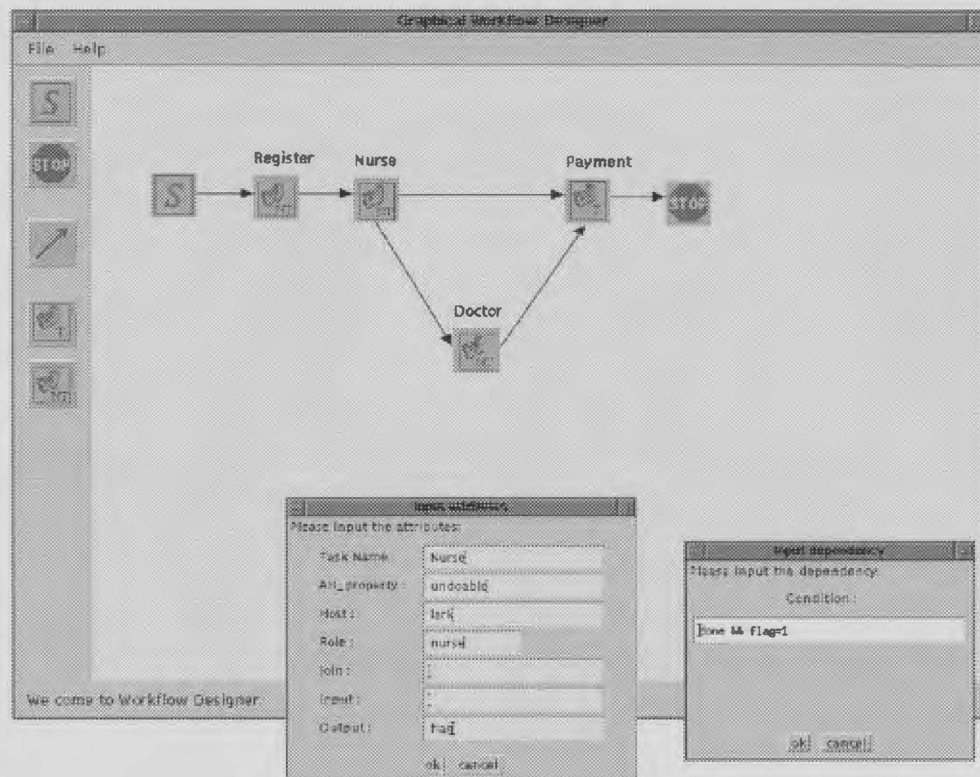


Figure 5.6: The snapshot of the Graphical Workflow Designer

up as shown in Figure 5.6. The larger one corresponds to the modification of the task attributes. The smaller one corresponds to the modification of an inter-task dependency.

The workflow designer is capable of modeling the normal structures including sequence, branch and loop. The workflow definition is stored into the workflow database during the design. When the 'open' operation is selected in the 'File' menu, the map will be re-constructed and shown on the canvas. Advantages of using database as the storage of the workflow definition are as follows.

1. Saving storage. Instead of saving the image of the map, graphical information like the coordinates and task types are stored into a database. These occupy

less space than an image file.

2. Standard access methods and easy manipulation. Database queries and updates are standardized as SQL.

3. Information sharing. Several workflow definitions can coexist in one database.

4. Concurrent design. Transactional properties of the database management may facilitate the concurrent design of the workflow definition.

5.3.2 Workflow server

Central to the server is a Scheduler which is responsible for scheduling tasks according to the workflow definition. It monitors the progress of task instances and decides the next tasks to run by examining the conditions attached to the relevant transitions.

The ad-hoc recovery related services include finding the affected tasks and agents who are responsible for the task executions, checking the validity of the decisions, undoing (redoing) work items, etc. Most of the interactions between the Task Manager and the Ad-hoc Recovery Handler are done by the *Ad-hoc Recovery Stub.* When the DBMS is being rolled back, *Ad-hoc Recovery Monitor* keeps track of the progress of the DBMS and reports to the Ad-hoc Recovery Handler when the rolling back is finished.

The workflow server is the implementation of backward ad-hoc recovery model described in Chapter 4. The scheduler schedules the tasks according to the workflow definition in case there is no ad-hoc recovery request. This is exactly the normal execution in the model. On the other hand, if there is ad-hoc recovery request,

some tasks will be repeated by the server and the repetition is still governed by the dependency described by the workflow definition. So the two conditions in Definition 4.3 are naturally met. This implies that the workflow instance in the presence of ad-hoc recovery is a well defined sequence of redirected executions.

The concept of general descendency in the model is nothing more than the normal SFT-descendency with the peers included. In our prototype, before any tasks are repeated, all their general descendents must be terminated. This means the new peer follows the general descendents of the old peer. This is actually the RD-order in the model.

In the architecture, there is a user interface to specify the tasks from which the new execution starts. When the recovery stub gets this set of tasks, it searches for all the least descendents along the execution paths to form a candidate RDF. It then creates a characteristic graph for the set specified by the user and the candidate RDF. By evaluating the connectivity of the characteristic graph, we can get a set of minimum RDF, RDT pairs. This result can be returned to the user to assist him/her in making a final decision on the pairs of RDF and RDT to initiate ad-hoc recoveries. The current version of the prototype has not yet implemented the graph algorithm.

The workflow server APIs defined in this prototype are as follows.

```
public interface WFServer extends Remote {
    public String [] getItem (String task_id, String agent) throws java.rmi.RemoteException; //Get new items from the worklist, and put it in the corresponding lists for display.
    public void pickItem (String name, String t_id, String agent) throws java.rmi.RemoteException; //Set pickup=1 in the worklist, activated when user performs pick up operation.
    public boolean validUnPick (String name, String t_id) throws java.rmi.RemoteException; //Check whether the
```

Unpick up operation is valid or not.

public void UnPickItem (String name, String t_id) throws java.rmi.RemoteException; //Set pickup=0 in the worklist, unpick up a work item.

public void activateItem (String name, String t_id, String agent, String status) throws java.rmi.RemoteException; //Change the state to 'active' in the worklist, performed after the work item is picked up.

public void doneItem (String name, String t_id, String output) throws java.rmi.RemoteException; //Change the state of a normal work item to 'done' in the worklist, performed after user clicks done option.

public String doneInAH (String name, String t_id, String output) throws java.rmi.RemoteException; //Change the state of an item whose status is 'in-AH' to 'done', and schedule the next work items to be undone.

public void donePostAH (String name, String t_id, String output) throws java.rmi.RemoteException; //Change the state of an item whose status is 'post-AH' to 'done', and schedule the next task instances.

public String [] findID (String name, String t_id) throws java.rmi.RemoteException; //Find the id tuple for a specified item.

public String [] fetchAttrs (String [] item_id, String[] attrs_wanted) throws java.rmi.RemoteException; //Fetch the attributes wanted for an item whose id tuple is given by item_id, which is returned by findID().

public void failItem (String name, String t_id) throws java.rmi.RemoteException; //Change the state of a normal work item to 'fail' in the worklist, indicating the corresponding task is unsuccessful.

public boolean login (String uname, String passwd, String taskid) throws java.rmi.RemoteException; //Validate the login of an agent to a task, performed when an agent is trying to login.

public void newInstance(String pname, String agent) throws java.rmi.RemoteException; //Create a new workflow instance, including the first task instances (followed 'start').

public SerialObj findDescendent(String [] taskSet, int length, String[] ids) throws java.rmi.RemoteException; //Find the descendents of tasks in taskSet of workflow instance specified by ids.

public String[] getAllTask(String w_id) throws java.rmi.RemoteException; //Get all the task names of workflow 'w_id' from the task specification table.

public void saveDecn(SerialObj seObj) throws java.rmi.RemoteException; //Save the decisions of agents into the decision table. The agents are specified in seObj.

public void quit(String t_id, String agent) throws java.rmi.RemoteException; //Make all unpicked items available if an agent quits.

public void undo() throws java.rmi.RemoteException; //Start the undo phase of the ad-hoc recovery. Undo the least general descendent tasks. Their ancestors which need to undo will be scheduled by doneInAH() and performed by activateItem().

}

5.3.3 Task manager

Task manager is the synonym of the *worklist handler*. It is responsible for worklist manipulations such as selecting a work item, reassigning a work item, notifying completion of a work item, and invoking a tool or client application as part of the work item [37]. The implication of the last function is that the task manager is the wrapper of the real task. In our prototype, the functions of the task manager have been extended to provide interface for the ad-hoc recoveries.

There are four list display in the Task Manager GUI (Figure 5.8). They are Todo list, Done list, Redo list, and Comp-for-redo list. The transition of one work item between these lists is shown in Figure 5.7.

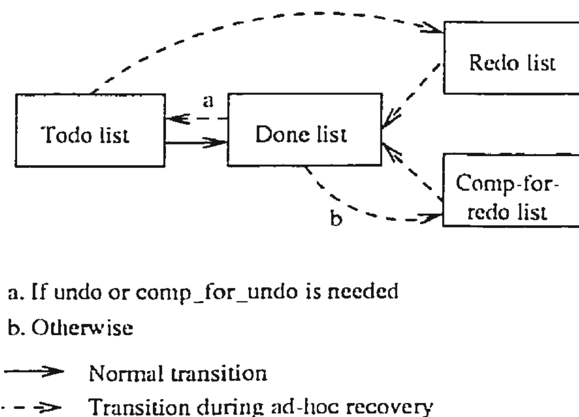


Figure 5.7: Transition of a work item between lists

Each list provides a set of operation interface where the user can perform state transition for each work item.

- **Todo list**

  It holds the pre-AH work items and the in-AH items whose states are Unpickup,

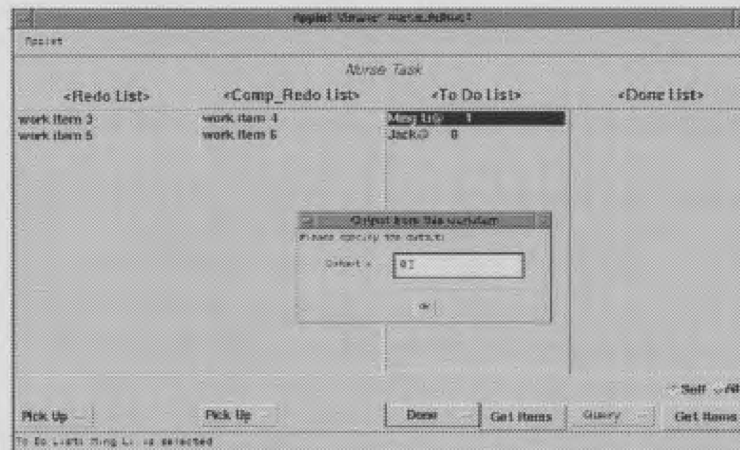


Figure 5.8: Task manager GUI

Pickup, Initial or Active. Operations provided for Todo list are: Pick Up, UnPick Up, Activate, Done, Fail, Commit, Abort.

- Done list

  It holds the pre-AH items whose states are 'done' or 'commit'. This is also where ad-hoc recovery requests are submitted. Operations provided for done list are: Query, Req Extl, Req IntA, Req IntN, Decision.

- Redo list

  It holds the post-AH items which are to be redone. Operations provided for redo list are: Query, Activate, Done, Fail, Commit, Abort.

- Comp-for-redo list

  It holds the post-AH items which are to be compensated for redo. These items are not undone. Operations provided for comp-for-redo list are: Query, Activate, Done, Fail, Commit, Abort.

5.3.4 Database server

The workflow schema, control data and the work items are stored in a relational database, Mini SQL[2] database in this prototype. Imaginary JDBC driver is used to access the mSQL database.

5.4 Backward ad-hoc recovery protocol

Three ad-hoc recovery protocols are proposed in [26]. They are *external collaboration*, *internal independent* and *internal automatic* protocols. The differences among these three protocols are the degree of machine involvement. External collaboration is the most flexible one with least machine involvement. In internal automatic protocol, after the workflow server receives the recovery request, it handles all the remaining processing. A practical system should provide all these three protocols from which the user can choose a preferred one in processing ad-hoc recovery requests. In this prototype system, only external collaboration protocol is implemented.

Initiator

1. Make an ad-hoc recovery request, specifying the set of the tasks to start from, then wait for response;
2. Upon receiving the response, consult all the agents whose ids are returned, and collect their decisions;
3. Make a decision-handling request, specifying the decisions collected at step 2,

[2] Mini SQL is a product from Hughes Technologies Inc.

and wait for the response;

4. If the response is *decision-invalid* then consult with other agents again, make a new decision, and go to step 3, else if the response is *recovery-impossible* then either go to step 1 or exit, else initiation is successful;

Agent

1. Pick up a work item w, assuming it is for task i;
2. If $w.status$ = *in-AH/post-AH*, perform operation $w.oper$;
3. Else activate task i;
4. When the operation at 2 or 3 is finished, do $w.state \leftarrow$ *done/commit*;

Workflow Server

- **ad-hoc recovery stub**

  1. If an ad-hoc recovery request is received then;

     a. lock the process instance;

     b. search for the task instances that will be affected by the ad-hoc recovery, and the agents' ids in charge of these instances; The affected task instances are those that are still active or have been done, which are the descendents of the tasks in the recovery request.

     c. return the agent's ids obtained at step b to the initiator;

  2. If a handling-decision request is received then;

a. if the decision conflicts with the tasks' attributes, return *decision-invalid*, else if at least one decision is *not-exist* return *recovery-impossible* and unlock the process instance, else;

i. store all the decisions in the decision-list in the database;

ii. create two entries in the worklist for every decision, an *in-AH* work item and a *post-AH* work item.

iii. inform the DBMS to do backward scanning (undoing the representations of the task instances obtained at step 1.b);

iv. activate ad-hoc recovery monitor;

v. return *recovery-starts* to the initiator;

- **ad-hoc recovery monitor**

1. Wait for the message from the DBMS (either T_{ij}*_undone* or *all_undone*);
2. If the message is T_{ij}*_undone*:

a. create in the worklist a work item w for task instance T_{ij};

b. $w.status \leftarrow in\text{-}AH$;

c. $w.oper \leftarrow T_{ij}.Dec.in_AH_oper$;

d. go to step 1;

3. If the message is *all-undone*:

a. if there exists a work item in the worklist with status *in-AH* then wait until the antecedent becomes false and go to the next step, otherwise go to the next step;

b. unlock the process instance;

- **scheduler**

1. Determine the next task T_i to run;

2. If there is an entry for T_{ij} in the worklist then;

   a . $w.status \leftarrow$ *post-AH*;

   b . $w.oper \leftarrow T_{ij}.Dec.post_AH_oper$;

3. Else;

   a . create in the worklist a work item w for T_{ij};

   b . $w.status \leftarrow$ *pre-AH*;

In this protocol, the ad-hoc recovery stub must find the required information about the affected task instances. Since the affected task instances are the ones on the paths from the tasks in RDT' to the tasks in RDF, their ancestors are the tasks in RDT'. In addition, they must either terminate or are ongoing. (A task instance not started yet is never involved in ad-hoc recovery.) Since RDT' is supplied by the initiator, the stub can retrieve the required information by issuing queries to the DBMS.

Chapter 6

Implementation and an example

The prototype is implemented in a pure Java environment. The Graphical Workflow Designer, the Workflow Server, and the Task Managers all run on RedHat Linux (Distribution 5.1 with Linux kernel 2.0.34). The database server runs on V3.2 62 alpha. The JDK version used is 1.1.6 and the JDBC driver is mSQL-JDBC 1.0b3.

6.1 Java features for enterprise computing

In May 1996, Java celebrated the inaugural JavaOne conference. The conference's underlying theme was Java's transition from an applet language to a hard-core computing environment. Since that conference, that theme has been growing into a reality: Java as a language for enterprise computing.

Enterprise computing traditionally refers to the mission-critical systems on which a business depends. At the heart of Java's enterprise computing philosophy lie the distributed computing and database access APIs–RMI and JDBC, respectively. Older

languages require third-party APIs to provide this kind of support. Java, on the other hand, includes these features into the central Java distribution that can be found on every Java platform.

6.1.1 Java DataBase Connectivity (JDBC™)

JDBC allows developers to write applications that access relational databases without considering which particular database they are using. When they write a Java database program, that same program will run against Oracle, Sybase, Ingres, mSQL, or any other database that supports this API [22]. The following is a piece of code showing the JDBC connection from the Workflow Server.

```
public String [] getItem(String my_task_id, String agent) throws RemoteException {
    String [] display;
    display = new String[50];
    try {
        Class.forName("com.imaginary.sql.msql.MsqlDriver");
        String url = "jdbc:msql://www.cs.mun.ca:1114/workflow_db";
        Connection con = DriverManager.getConnection(url, "xuemin", "");
        Statement stmt = con.createStatement();
        ResultSet rs =
        stmt.executeQuery("SELECT w_id, wi_id, t_id, ti_id, name, status, state, oper, pickup FROM worklist
WHERE t_id='"+my_task_id+"' AND (agent='"+agent+"' OR agent='') AND get=0");
        while(rs.next()) {
            String workflow_id = rs.getString("w_id");
            String workflow_ins_id = rs.getString("wi_id");
            int pickup = rs.getInt("pickup");
            ....
        }
    } catch( Exception e ) {
        System.out.println("error in msql: "+e.getMessage());
        e.printStackTrace();
```

```
        }
        return display;
    }
```

In the above example, the Workflow Server asks the JDBC *DriverManager* to hand it the proper database implementation based on a database URL. The database URL looks similar to other Internet URLs. The actual content of the URL is loosely specified as *jdbc:subprotocol:subname*. The subprotocol identifies which driver to use, and the subname provides the driver with any required connection information. For the imaginary JDBC implementation for mSQL that we used in the prototype, the URL is *jdbc:msql://www.cs.mun.ca:1114/workflow_db*. In other words, this URL says to use the mSQL JDBC driver to connect to the database workflow_db on the server running at port 1114 on www.cs.mun.ca. After the connection is set up, the Java code creates a *statement* object *stmt* and executes a query on the database. The query result is returned and stored in a class called *ResultSet*. ResultSet class provides a set of methods to extract the information from the query. For example, *getString("w_id")* extracts field *w_id* of the first row of table *worklist* during the first iteration of the loop..

6.1.2 Remote Method Invocation (RMI™)

RMI allows Java programs to call certain methods on a remote server. Remote server implements a remote interface that specifies which of its methods can be invoked by clients. Clients can invoke the methods of the remote server almost exactly as they invoke local methods.

Remote Procedural Calls (RPC) is an older technology Sun developed that does much the same thing as RMI. RPC is language- and processor-independent; RMI is processor-independent by nature, but limited to programs written in Java. RPC will eventually be made available in Java.

To get the cross-platform portability that Java provides, RPC requires a lot more overhead than RMI. RPC has to convert arguments between architectures, so that each computer can use its native data types. Furthermore, RPC can only send primitive data types, while RMI can send objects.

In short, RMI is a good solution for communication between Java programs on different hosts. However, if connection with programs written in other languages is needed, it is better to investigate RPC, or look into CORBA [17]. A piece of code containing RMI in the prototype is as follows.

```
try {
    WFServer wf_server = (WFServer) Naming.lookup("rmi://jay.cs.mun.ca/WFServerImpl");
    String [] message = new String[50];
    for (i=0; i<50; i++) message[i]="";
    message = wf_server.getItem("03",parent.parent.parent.parent.uname);
} catch (Exception e1) {
    System.out.println("Exception in getNewItem: " + e1);
}
```

In this example, the client looks for the workflow server registered and bound to *WFServerImpl* on host *jay.cs.mun.ca* using protocol rmi. Then it invokes the method *getItem()* on the server side and captures the returned value in *message*.

6.2 Some implementation details

In the ad-hoc recovery procedure, the two most important functions are doneInAH() and donePostAH(). Function doneInAH() finds the next item(s) to be undone, while donePostAH() finds the next item(s) to be invoked. They are called when the user performs Done operation on the selected work items in the Task Manager GUI. DoneInAH() is called for an item with tag <undo> in Todo list; DonePostAH() is called for an item in Redo list or a Comp-for-redo list. Figure 6.1 and 6.2 are the flow charts of doneInAH() and donePostAH().

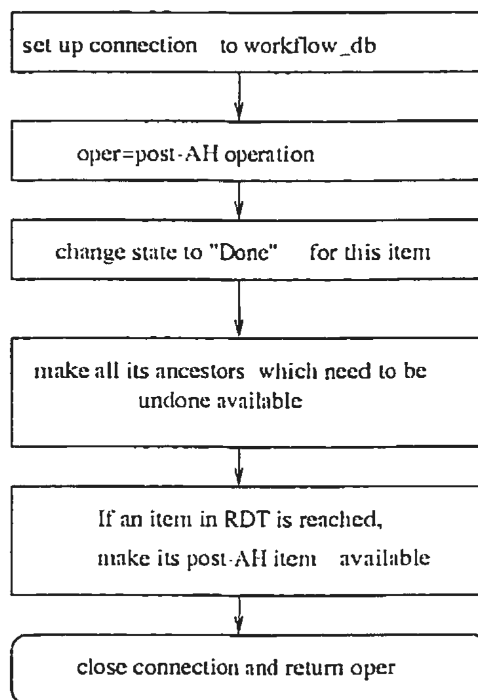


Figure 6.1: Flow chart of doneInAH()

When the decision is made, the Workflow Server looks for the involved tasks and generates all the undo(redo) items in the worklist. However, only the least general descendent tasks are made available to the user. They are elements in *RDF*. After

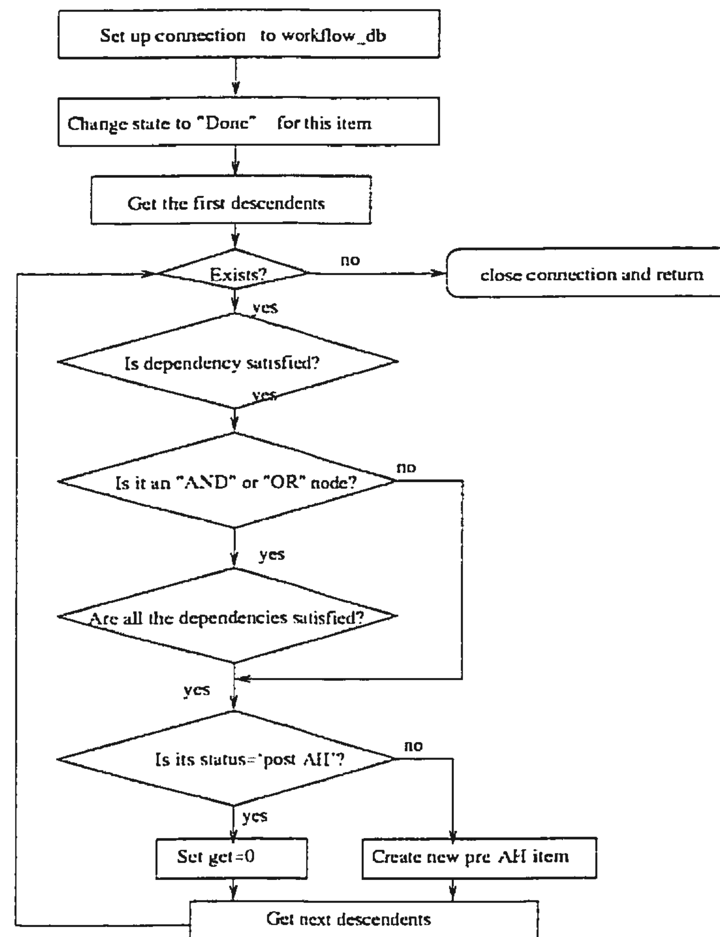


Figure 6.2: Flow chart of donePostAH()

the undo phase is finished, the items in *RDT* whose status are 'post-AH' become available. These items are fetched to the Task Manager GUI during the getItem() call.

6.3 Ad-hoc recovery in a hospital workflow

The example workflow has four tasks as depicted in Figure 5.6. The Register task is responsible for registering the patients' personal information and initiating a workflow

instance. The Nurse task and the Doctor task perform examinations on the patients and record the results into the patient database. The patients pay their examination fees at the cashier's desk where the Payment task is executed.

Suppose there are two patients coming to the hospital, Tom and Mike. After they are registered, two work items are created for Nurse task. The patients' names are used to identify the work items. When a nurse logs on to the Nurse Task Manager, the two new items are fetched into the Todo list, as can be seen from Figure 6.3. The structure of each entry is $< itemname >$@$< status >< state >< pickup >$. The default value for $< state >$ is null which represents initiate state. The default value for $< status >$ is null too which represents pre-AH status in Todo list. Value of pickup can be 0 or 1.

The nurse picks up Tom and activates the real task, examining pulse and blood pressure. After the examination, he records the results and submits his work by selecting "Done" operation. The output window is popped up where he enters 1, indicating the doctor's examination is required. The scenario of the above operations is shown in Figure 6.4.

Now a new work item is created for the Doctor task. When the doctor gets new items from the Doctor Task Manager, he observes Tom waiting for his examination and activates the real task, reading the nurse's testing results and examines, prescribes, etc. The doctor's screen looks like Figure 6.5.

Before the doctor finishes his examination on Tom, the nurse finds out he entered the wrong pulse rate on Tom's record. Then he requests for an external ad-hoc recovery for Tom. The workflow server receives the request and is supposed to lock the

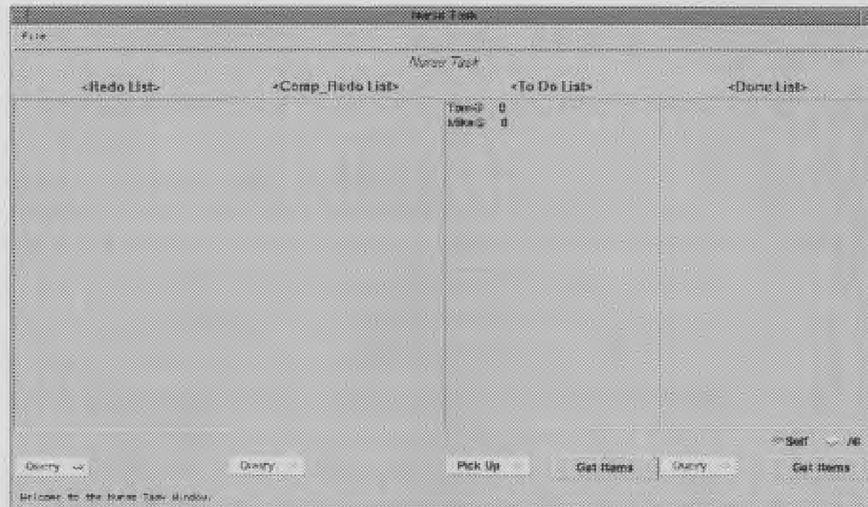


Figure 6.3: The initial Nurse Task Manager window

workflow instance. (The lock mechanism is not implemented in the current version.) Then the server asks the nurse for the restart-from (RDT') tasks, which is Nurse task in the example. Then the Workflow Server detects all the affected tasks, which are Nurse task and Doctor task, and returns the agents'ids to the nurse. The sequence of steps are depicted in Figure 6.6.

Now, the nurse talks to the doctor about how to recover the two tasks. They may decide to undo their work then redo them. Thus the decision window looks like Figure 6.7.

After the decision is sent to the workflow server, the undo phase of the recovery starts. First, undoing the work item Tom in the Doctor task. Then undoing it in the Nurse task. The undo items appear in the Todo list with tag <undo>. The undo procedure of each item includes two steps. First, the DBMS performs a backward scan and restores the database states to the previous point. Then the agents undo the real work if it is not covered by the DBMS recovery and it is deemed necessary.

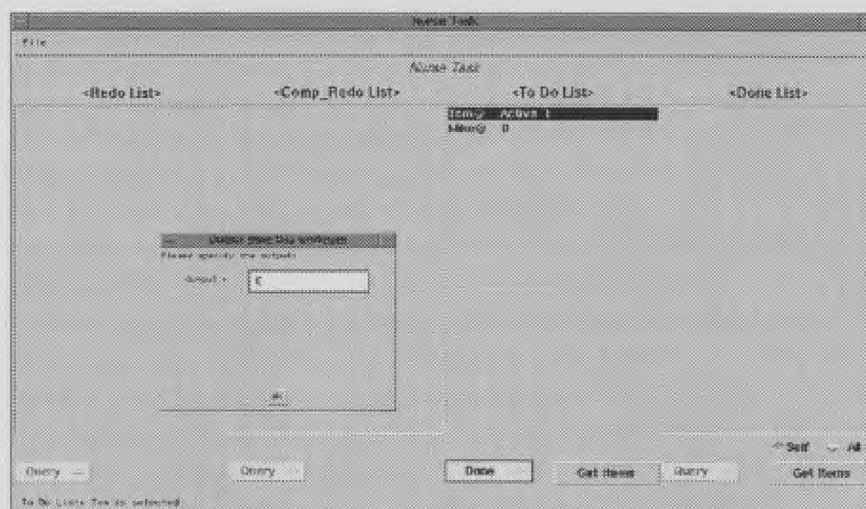


Figure 6.4: Processing work item Tom in the Nurse Task Manager window

Undoing Doctor Task is shown in Figure 6.8. After the doctor finishes undo, an undo item is available in Nurse task. This roll back procedure will go on until a restart-from task is reached. The undo process for Tom in the Nurse Task Manager window is shown in Figure 6.9.

After the undo phase, a redo work item Tom in the Nurse Task manager window is created. It is now put into the Redo list. After the nurse records the correct testing results again for Tom, the scheduler finds the next task to be run according to the new output, which might be '1' or '0'. In this example, the output is the same as before. Thus the next task is Doctor task. Redoing Nurse task and Doctor task is shown in Figure 6.10 and 6.11.

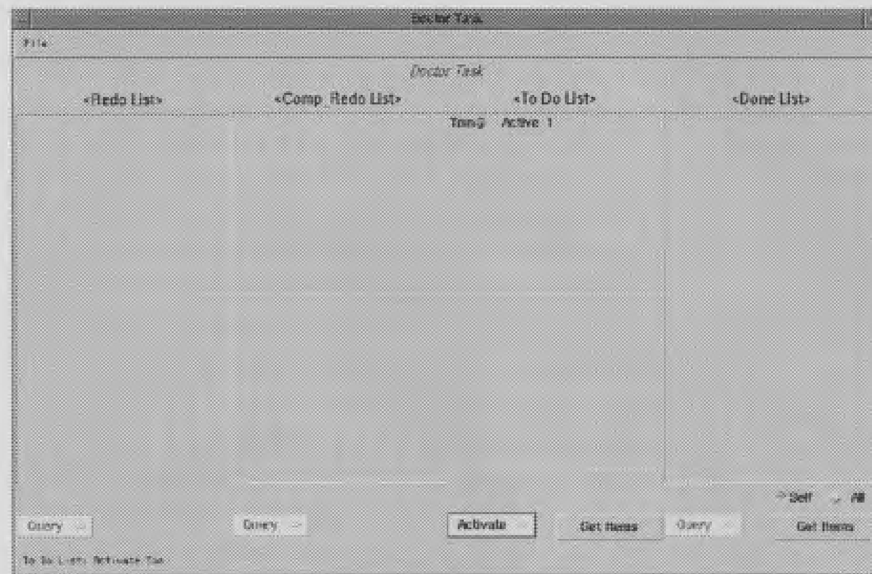


Figure 6.5: Processing work item Tom in the Doctor Task Manager window

6.4 Discussion about availability and scalability

In the client-server architecture, the server is always thought as the bottleneck of communication. If the server is down or overloaded, it is possible that the clients are blocked in processing. Although not implemented completely, we tried multiple Workflow Server structure in the prototype. When the initial server is down, another server takes its place. The piece of code for the two-server version is as follows.

```
public class Adhoc1 extends Applet {
    public String uname,passwd,main_server=“lark”,back_server=“auk”;
    public void init() {
        ...
        try {
            WFServer wf_server = (WFServer) Naming.lookup(“rmi://”+main_server+“.cs.mun.ca/WFServerImpl”);
            ...
        }
        catch (Exception e1) {
            System.out.println(“Server ”+main_server+
            “ is down, try ”+back_server+“ ...”);
```

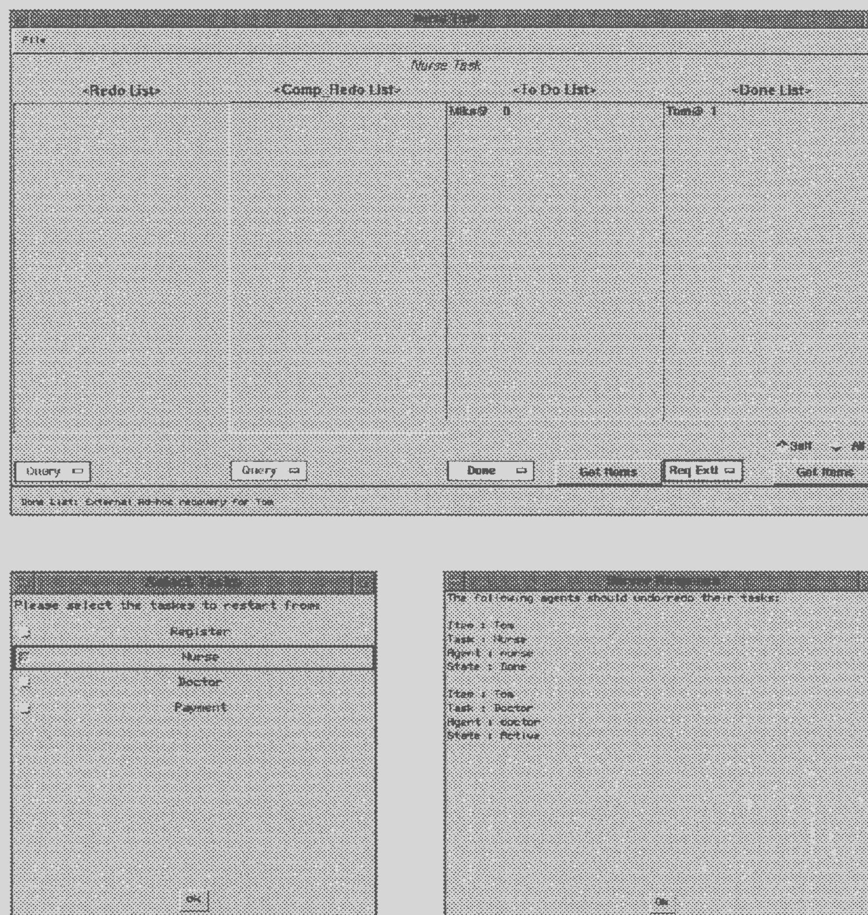


Figure 6.6: External ad-hoc request and choose restart-from tasks

```
            try {
                WFServer wf_server = (WFServer) Naming.lookup("rmi://"+back_server+".cs.mun.ca/WFServerImpl");
                ...
            }
            catch (Exception e2) {
                System.out.println("Exception in login: "+e2);
                e2.printStackTrace();
            }
        }
    }
}
```

Input Decisions

Please input decisions:

Task : Nurse   Agent : nurse

in-AH operation : undo

post-AH operation : redo

Task : Doctor   Agent : doctor

in-AH operation : undo

post-AH operation : redo

ok

Figure 6.7: The decision

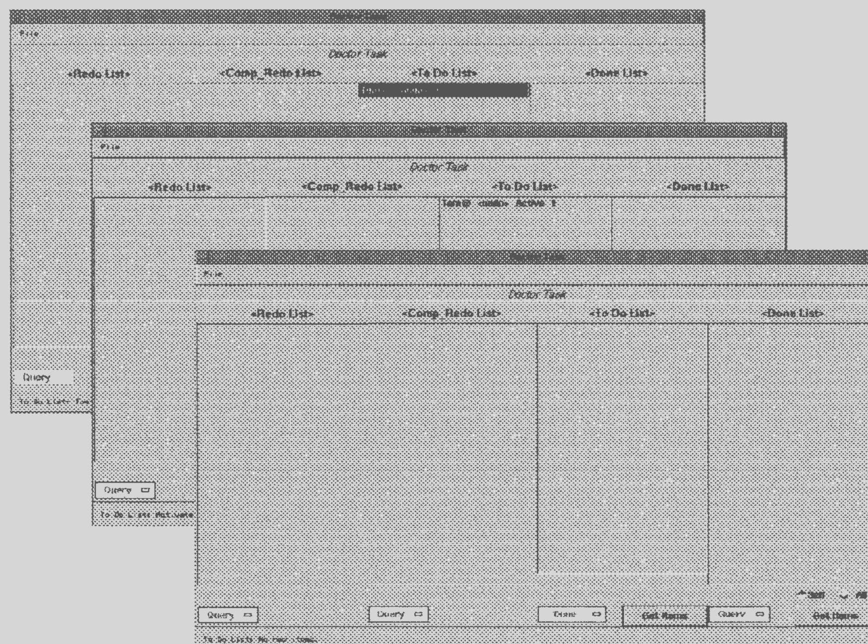


Figure 6.8: Undo Doctor task

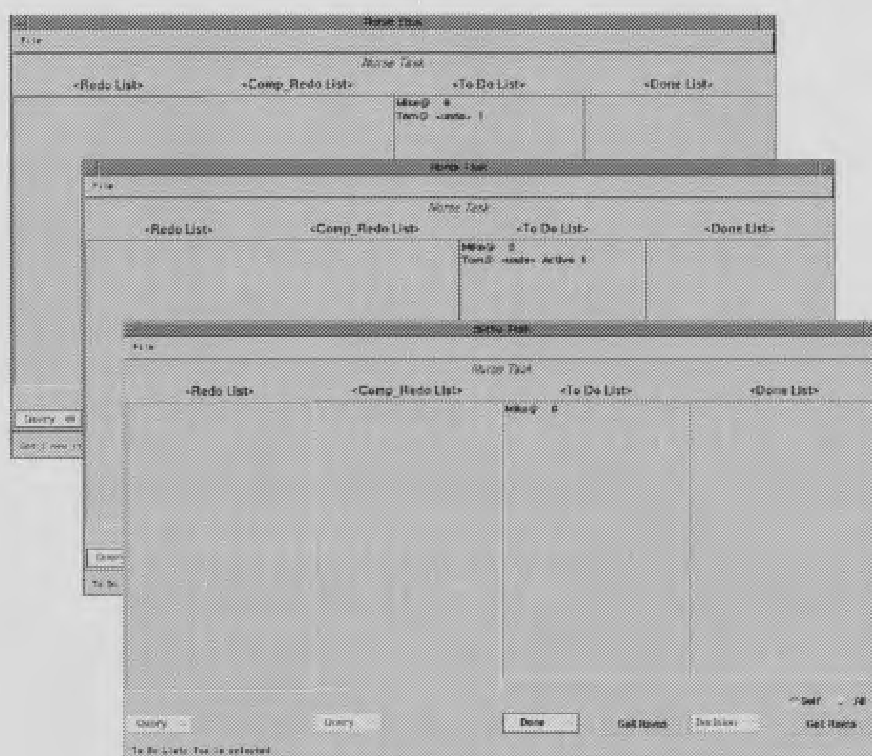


Figure 6.9: Undo Nurse task

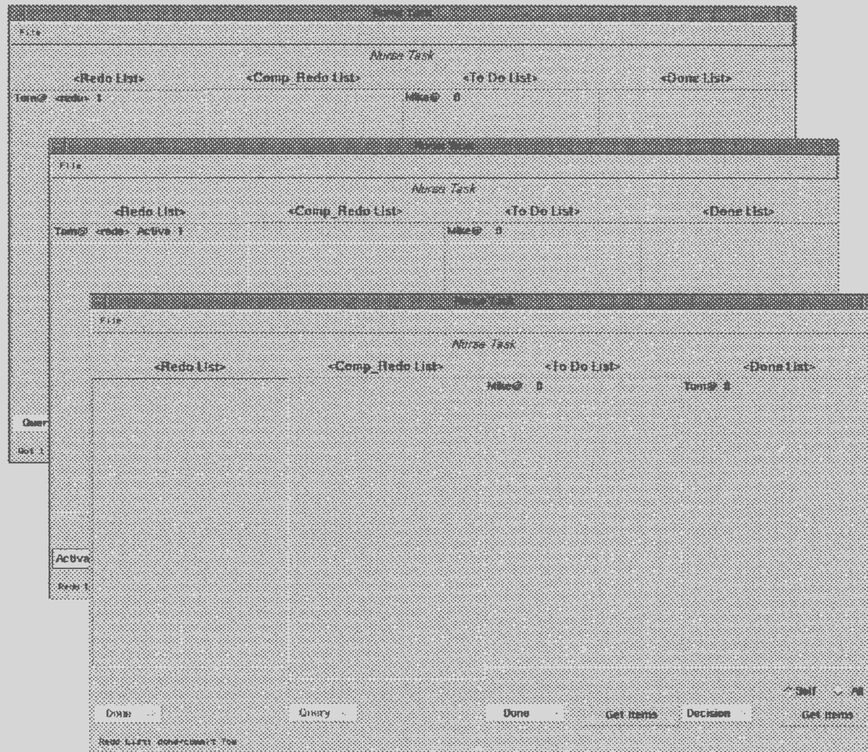


Figure 6.10: Redo Nurse task

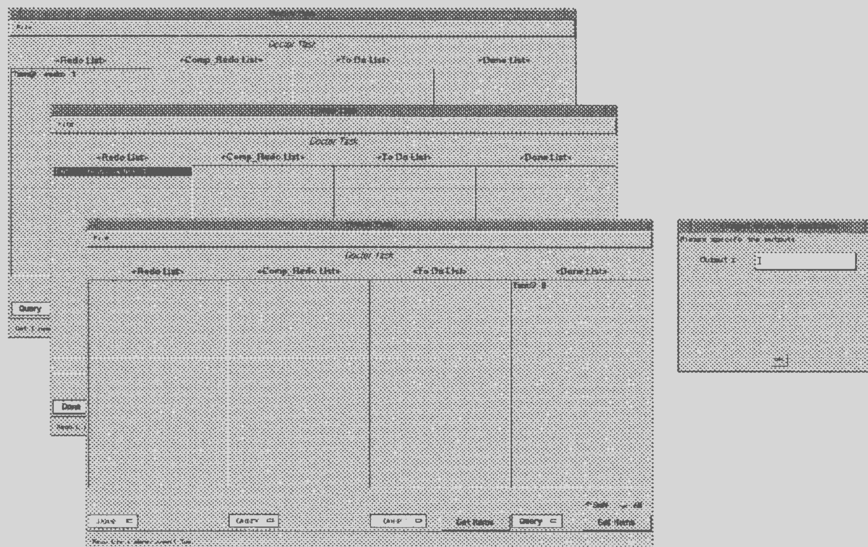


Figure 6.11: Redo Doctor task

```
garfield > show_task; show_arrow

Welcome to the miniSQL monitor.  Type \h for help.

mSQL >     ->
Query OK.  6 row(s) modified or retrieved.

+------+------+--------------------+-----------+-------------------+-------------------+-----+----------+----------+----------+--------+--------+
| w_id | t_id | name               | type      | AH_prop           | host              | join| role     | input    | output   | x      | y      |
+------+------+--------------------+-----------+-------------------+-------------------+-----+----------+----------+----------+--------+--------+
| 01   | 01   |                    | Start     |                   |                   |     |          |          |          | 48     | 89     |
| 01   | 02   | Register           | NT        | undoable          | dove              |     | register |          |          | 131    | 91     |
| 01   | 03   | Nurse              | NT        | undoable          | lark              |     | nurse    |          | flag     | 212    | 93     |
| 01   | 04   | Doctor             | NT        | undoable          | eagle             |     | doctor   |          |          | 293    | 218    |
| 01   | 05   | Payment            | T         | undoable          | garfield          | or  | cashier  |          |          | 383    | 94     |
| 01   | 06   |                    | Stop      |                   |                   |     |          |          |          | 465    | 96     |
+------+------+--------------------+-----------+-------------------+-------------------+-----+----------+----------+----------+--------+--------+


mSQL >     ->
Bye!

Welcome to the miniSQL monitor.  Type \h for help.

mSQL >     ->
Query OK.  7 row(s) modified or retrieved.

+-------------------+-------------------+----------------------------+----------+----------+--------+--------+
| from_task         | to_task           | condition                  | anchor_x | anchor_y | end_x  | end_y  |
+-------------------+-------------------+----------------------------+----------+----------+--------+--------+
| 0101              | 0102              |                            | 85       | 106      | 131    | 106    |
| 0102              | 0103              | done                       | 168      | 106      | 211    | 106    |
| 0103              | 0104              | done && flag=1             | 230      | 130      | 291    | 232    |
| 0103              | 0105              | done && flag=0             | 249      | 107      | 382    | 107    |
| 0104              | 0105              | done                       | 330      | 234      | 399    | 132    |
| 0105              | 0106              | commit                     | 420      | 109      | 464    | 109    |
| 0102              | 0102              |                            | 149      | 116      | 149    | 116    |
+-------------------+-------------------+----------------------------+----------+----------+--------+--------+


mSQL >     ->
Bye!

garfield >
```

Figure 6.12: Build time data

garfield > show_worklist ; show_decision

Welcome to the miniSQL monitor. Type \h for help.

mSQL >   ->
Query OK. 10 row(s) modified or retrieved.

w_id	wi_id	t_id	ti_id	name	agent	state	status	oper	output	host	get	pickup
01	01	02	01	Tom	register	Done	pre-AH			dove	0	0
01	01	03	01	Tom	nurse	Done	pre-AH		1	lark	1	1
01	02	02	01	Mike	register	Done	pre-AH			dove	0	0
01	02	03	01	Mike	nurse		pre-AH			lark	0	0
01	01	04	01	Tom	doctor	Active	pre-AH			eagle	0	1
01	01	03	01	Tom	nurse	Done	in-AH	undo			1	1
01	01	03	01	Tom	nurse	Done	post-AH	redo	1		1	1
01	01	04	01	Tom	doctor	Done	in-AH	undo			1	1
01	01	04	01	Tom	doctor	Done	post-AH	redo			1	1
01	01	05	01	Tom			pre-AH			garfield	0	0

mSQL >   ->
Bye!

Welcome to the miniSQL monitor. Type \h for help.

mSQL >   ->
Query OK. 2 row(s) modified or retrieved.

w_id	wi_id	t_id	ti_id	tname	state	iname	agent	in_AH_oper	post_AH_oper	flag	leaf
01	01	03	01	Nurse	Done	Tom	nurse	undo	redo	1	0
01	01	04	01	Doctor	Active	Tom	doctor	undo	redo	1	1

mSQL >   ->
Bye!

garfield >

Figure 6.13: Run time data

Chapter 7

Conclusions and future work

This thesis focuses on the modeling in workflow applications and architectural aspects of the ad-hoc recovery of workflow management systems. The contributions of the research presented in this thesis at the modeling level are as follows:

- A generic workflow model is refined based on the workflow reference model. Besides the typical concepts like task and task instance, in-depth concepts like the local state, legal set, and normal execution are formalized. A definition graph is also given to represent the workflow schema graphically.

- Based on the generic workflow model, the ad-hoc recovery model is proposed. A third dimensional concept in the workflow execution, the run, is abstracted and its behavior studied during redirections.

- A set of concepts are defined, such as peers, redirected executions, well defined sequence of redirected executions, etc.. Two sets, RDT and RDF and the six

constraints on them are given to characterize the redirections involved in ad-hoc recoveries.

- An execution graph, concise representation of the workflow execution is proposed for the first time. Compared with other literatures, our model is more accurate in depicting normal executions of a workflow and is original in depicting workflow executions when exceptions occur.

The contributions of this thesis at the system development and implementation level include the following:

- A client-server WFMS architecture is given and the components handling ad-hoc recovery are embedded into the architecture. It can be used to manage both normal workflow executions and backward ad-hoc recoveries.
- A protocol is given to handle backward ad-hoc recoveries. It is also demonstrated with an example hospital workflow.
- Java features for enterprise computing are experimented in the system design and implementation. Availability and scalability issues are discussed briefly.

The current model and implementation can be refined in many dimensions. The forward ad-hoc recovery is not formalized in the model. It is almost symmetric with the backward recovery model except that there are no peers in the forward redirected executions. The RDT and RDF associated with forward recovery should have different constraints. We feel that the constraints on RDT and RDF in forward recovery are simpler than those of the backward recovery.

The ad-hoc recovery model does not deal with the dynamic changes made to the workflow definition. It is not a fully dynamic model. In order to provide more flexibility, other modeling concepts can be incorporated.

In the system implementation, we dealt only with External Collaboration Protocol. The other two are not implemented and their run time performance is not evaluated. Also the whole system is not implemented in a fully scalable fashion. These work are to be continued in the future.

Bibliography

[1] A. Bonner, A. Shruf, and S. Rozen. LabFlow-1: A Database Benchmark for High Throughput Workflow Management. *Proc. of the 5th. Intl. Conference on Extending Database Technology*, pages 25–29, Avignon, France, March 1996.

[2] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. *Proc. of the 16th. Intl. Conf. on Very Large Data Bases*, pages 507–518, August 1990.

[3] A. Forst, E. Kuhn, and O. Bukhres. General Purpose Workflow Languages. *Distributed and Parallel Databases*, 3(2):119–154, April 1995.

[4] A. Reuter and H. Wachter. The ConTract Model. *IEEE Data Engineering Bulletin*, 14(1), March 1991.

[5] A. Sheth, D. Worah, K. Kochut, J. Miller, K. Zheng, D. Palaniswami, and S. Das. The METEOR Workflow Management System and its Use in Prototyping Significant Healthcare Applications. *Proc. of the 13th. Towards an Electronic Patient Record (TEPR '97) Conference*, Medical Records Institute, April 1997.

[6] A. Sheth, K. J. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shevchenko. Supporting State-Wide Immunization Tracking Using Multi-Paradigm Workflow Technology. *Proc. of the 22nd. Intl. Conference on Very Large Data Bases*, Bombay, India, September 1996.

[7] A. Sheth, Krys J. Kochut. Workflow Applications to Research Agenda: Scalable and Dynamic Workflow Coordination and Collaboration Systems. *http://lsdis.cs.uga.edu/workflow/index.html*, 1997.

[8] A. Sheth, M. Rusinkiewicz. On Transactional Workflows. *ACM SIGMOD Record*, 22(3), September 1993.

[9] Action Technologies. Collaborative Business Application. *Presentations, http://www.actiontech.com/default.cfm?area=Products/Developers'Center/*, 1998.

[10] Amit Sheth and Krys J. Kochut. Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems. *Large Scale Distributed Information Systems Lab, http://lsdis.cs.uga.edu/workflow/index.html*, 1997.

[11] D. Barbara, S. Mehrotra, and M. Rusinkiewicz. INCAs: Managing Dynamic Workflows in Distributed Environments. *Journal of Database Management, Special Issue on Multidatabases*, 7(1):5–15, Winter 1996.

[12] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–154, April 1995.

[13] D. Worah, A. Sheth. What Do Advanced Transaction Models Have to Offer for Workflows? *Proc. of Intl. Workshop on Advanced Transaction Models and Architectures*, Goa, India, 1996.

[14] D. Worah, A. Sheth. Transactions in Transactional Workflows. *Advanced Transaction Models and Architectures*, Kluwer Academic Publishers, S. Jajodia and L. Kerschberg, editors, 1997.

[15] Dave Ensor and Ian Stevenson. Oracle Design. *O'Reilly Associates, Inc.*, First edition, March 1997.

[16] David Hollingsworth. The Workflow Reference Model. *http://www.aiim.org/wfmc/* , TC00-1003(Issue 1.1), November 1994.

[17] Elliotte Rusty Harold. JAVA Network Programming. *O'Reilly Associates, Inc.*, First edition, February 1997.

[18] F. Leymann, H. J. Schek, and G. Vossen. Transactional Workflows. *Dagstuhl seminar 9629*, 1996.

[19] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan. Functionality and Limitations of Current Workflow Management Systems. *IEEE Expert*, 12(5), October 1997.

[20] G. Alonso, H.J. Schek. Research Issues in Large Workflow Management Systems. *Proc. of the NFS Workshop on Workflow and Process Automation in Information Systems*, University of Georgia, Athens, GA, May 1996.

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

[22] George Reese. Database Programming with JDBC and JAVA. *O'Reilly Associates, Inc.*, First edition, June 1997.

[23] Hector Garcia-Molina, Kenneth Salem. Sagas. *ACM 0-89791-236-5/87/0005/0249*, pages 249–259, 1987.

[24] J. Tang, J. Veijalainen. Enforcing Inter-task Dependencies in Transactional Workflows. *Proc. of the 3rd Intl. Conf. on Cooperative Information Systems*, pages 72–86, May 1995.

[25] Jian Tang and Jari Veijalainen. Transaction-oriented Work-flow Concepts in Inter-organizational Environments. *Proc. of 4th Intl. Conference on Information and Knowledge Management*, November 1995.

[26] Jian Tang and Xuemin Xing. A Workflow Management Systems Architecture that Supports Ad-hoc Recoveries. *1999 International Database Engineering and Applications Symposium*, pages 332–340, August 1999.

[27] L. Fischer. The Workflow Paradigm - The Impact of Information Technology on Business Process Reengineering. Future Strategies, Inc., Alameda, CA, Second edition, 1995.

[28] M. Ansari, L. Ness, M. Rusinkiewicz and A. Sheth. Using Flexible Transactions to Support Multi-system Telecommunication Applications. *Proc. of the 18th Intl. Conf. on Very Large Data Bases*, pages 65–76, Vancouver, Canada, August 1992.

[29] M. Duitshof. Workflow Automation in Three Administrative Organizations. *Master's thesis*, University of Twente, The Netherlands, July 1994.

[30] M. Hsu, C. Kleissner. Objectflow: Towards a Process Management Infrastructure. *Technical Report, Digital Equipment Corporation*, 1995.

[31] M. Kamath, K. Ramamritham. Bridging the Gap Between Transaction Management and Workflow Management. *Proc. of the NFS Workshop on Workflow and Process Automation in Information Systems*, University of Georgia, Athens, GA, May 1996.

[32] N. Krishnakumar, A. Sheth. Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations. *Distributed and Parallel Databases*, 3(2):155–186, April 1995.

[33] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. *Proc. of the 19th Intl. Conf. on Very Large Data Bases*, pages 134–145, Dublin, Ireland 1993.

[34] P. Chrysanthis, K. Ramamritham. Acta: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 194–203, 1990.

[35] S. Joosten, G. Aussems, M. Duitshof, R. Huffmeijer, and E. Mulder. WA12: An Empirical Study about the Practice of Workflow management. University of Twente, Enschede, The Netherlands, July 1994.

[36] W. Jin, L. Ness, M. Rusinkiewicz, and A. Sheth. Concurrency Control and Recovery of Multidatabase Workflows in Telecommunication Applications. *Proc. of ACM SIGMOD Conference*, May 1993.

[37] Workflow Management Coalition. Workflow Management Coalition Terminology and Glossary. *http://www.aiim.org/wfmc/*, WFMC-TC-1001(Issue 2.0), June 1996.

[38] Y. Breitbart, A. Deacon, H. Schek, and A. Sheth. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multisystem Workflows. *SIGMOD Record*, 22(3):23–30, September 1993.