DISSERTATION

A FRAMEWORK FOR PROCESSING GENERALIZED ADVANCED TRANSACTIONS

Submitted by

Tai Xin

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2006

COLORADO STATE UNIVERSITY

February 8, 2006

WE HEREBY RECOMMEND THAT THE **DISSERTATION** PREPARED UNDER OUR SUPERVISION BY **TAI XIN** ENTITLED **A FRAMEWORK FOR PROCESSING GENERALIZED ADVANCED TRANSACTIONS** BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Charles Anderson

Indrajit Ray

Leo Vijayasarathy

Indrakshi Ray

Adviser

Department Head/Director

ABSTRACT OF DISSERTATION

A FRAMEWORK FOR PROCESSING GENERALIZED ADVANCED TRANSACTIONS

The traditional transaction processing model, though useful in database systems, does not perform well in many specialized applications. Researchers have addressed this problem by proposing various advanced transaction models. The advanced transactions are characterized by having a number of cooperative *subtransactions* whose execution is controlled by *dependencies*. There is few research work on unified transaction model and the analysis for correct specification of dependencies. The issues like dependency conflicts and redundancies are not addressed. For instance, dependencies can conflict with each other. A pair of conflicting dependencies will require contradicting operations in execution, and will in turn lead to deadlock and denial-of-service problems.

Moreover, existing work does not specify how to enforce dependencies and ensure the correct execution of advanced transactions. The dependencies pose new challenges which must be addressed in execution. A reliable scheduler is needed to ensure satisfaction of all dependencies in execution. In case of system crash, the system will be left in inconsistent state. We need effective failure recovery techniques, not only to restore the data items to consistent states, but also to preserve the dependencies. Malicious attacks cannot be prevented all the time. Correct and complete damage assessment are crucial for survivability of advanced transactions.

In this dissertation, I proposed a reliable transaction processing model for advanced transactions. It can be used to develop complex and powerful advanced transactions correctly and uniformly, and more important, it will be reliable in execution and have the power to survive system crashes and malicious attacks.

Tai Xin
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Spring 2006

ACKNOWLEDGEMENTS

I am very grateful and want to deliver my thanks to Dr. Indrakshi Ray for being my adviser. The scientific guidance and constructive criticism she provided has helped me greatly in conducting my research and completing this dissertation. Her inspiring thoughts and support has always assisted me in my personal and professional growth. In particular, it have been her problem-oriented view, her concentration on essentials, and her accuracy and attitude in research that crucially contributed to my scientific publications, courseworks, and especially this dissertation.

I sincerely thank Dr. Indrajit Ray for his continuous help during my PhD study and guidance in developing this dissertation. He helped me understand the topics in security and transaction precessing as well as work out this dissertation. I also want to thank Dr. Charles Anderson and Dr. Leo Vijayasarathy for their willingness to serve on my committee and be co-reviewers of this dissertation. Their kind and enthusiastic help, suggestions, and comments also made the success of this dissertation. They gave me the opportunities to present preliminary results and gave substantial remarks which helped to improve the dissertation significantly. My thanks also go to Dr. Darrell Whitley, Dr. Dale Grit, Dr. Sudipto Ghosh, Dr. Yashwant Malaiya, Dr. Bruce Draper and all faculty members in Computer Science department. During my study in Fort Collins, they helped me on subject knowledge and my professional growth.

Thanks also to my colleague Yajie Zhu who worked together with me in the advanced transaction processing project and provided many valuable suggestions. Further more, many friends have helped me significantly especially during the time of writing this dissertation, either by their mental support, appreciation, or practical help, namely Fang Chen, Na Li, Lin Jiang, Dazhi Wu, Monte Lunacek, Shuxin Yin, Nayot Poolsappasit, Vidya Waykole, Trung Dinh Trong, and Eunjee Song.

Last, but not least, I am indebted to my parents Qingyu Kong and Bolin Xin for giving me tremendous support, help, and appreciation during my study and this work. Without their love and support, I cannot finish my PhD study here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background on Advanced Transactions

The traditional transaction processing model has been widely used in database systems. These transactions preserve the ACID properties [16, 46], namely, properties of *atomicity*, *consistency*, *isolation* and *durability*. The ACID properties are necessary in order to ensure the reliability and correctness of transaction execution for traditional database applications.

With the growth of specialized applications consisting of complex co-operating activities, the need for *advanced transactions* increased rapidly in recent years. The traditional transaction processing model, though useful in database systems, does not perform well and is not suitable for these advanced applications. Researchers have addressed this problem by proposing various new extended transaction models [10, 24, 28, 32, 42, 73, 78, 80, 85, 106]. Two special properties of these applications are that, they normally contain highly cooperative activities and have very long running durations. Researchers have also proposed corresponding technologies supporting transaction processing for these advanced transactions [7, 15, 18, 24, 26, 28, 30, 31, 32, 33, 37, 43, 49, 54, 73, 75, 78, 95, 98, 99, 109, 114, 115, 116, 119].

Advanced transactions are typically composed of a set of component *subtransactions*, whose execution are coordinated by specially designed structures called *dependencies*. A subtransaction performs a logical unit of work. Subtransactions are also referred to as *tasks* or *activities* in many papers. I will use the term subtransaction in the rest of this dissertation. Dependencies are used to coordinate the cooperation of the subtransactions. Existing research work in advanced transactions, such as ACTA [30], SAGAS [42] and ASSET [24], have described dependencies as a means to control the interactions among subtransactions. A dependency specifies how the execution of one subtransaction T_{wa} causes (or affects) the execution of another subtransaction T_{wb} in an advanced transaction. For example, a *strong commit* dependency specifies that if T_{wa} commits successfully then T_{wb} must also commit. Dependencies can also specify the ordering of execution for two

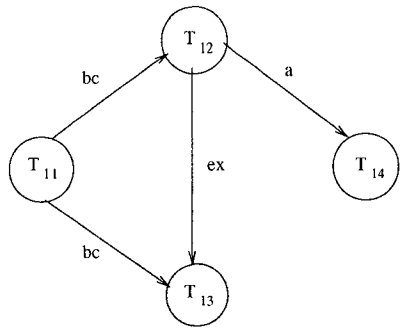

Figure 1: Advanced Transaction and Dependencies

subtransactions. For example, a *commit* dependency specifies that if both T_{wa} and T_{wb} commit, then T_{wa} must commit before T_{wb} does so. A *begin-on-commit* dependency specifies that T_{wb} cannot begin until T_{wa} has committed successfully. Below we show an example of an advanced transaction.

Example 1

Consider an example of a travel agency application. The advanced transaction ADT_1 consists of the following set of subtransactions $\{T_{w1}, T_{w2}, T_{w3}, T_{w4}\}$ described below.

subtransaction T_{w1} Reserve a ticket on Airlines A

subtransaction T_{w2} Purchase the Airlines A ticket

subtransaction T_{w3} Cancel the reservation

subtransaction T_{w4} Reserve a room in Resort C

The various kinds of dependencies that exists among the subtransactions are shown in Figure 1. There is a *begin-on-commit* dependency between T_{w1} and T_{w2} and also between T_{w1} and T_{w3}. This means that neither T_{w2} or T_{w3} can start before T_{w1} has committed. There is an *exclusion* dependency between T_{w2} and T_{w3}. This means that either T_{w2} can commit or T_{w3} can commit but not both. Finally, there is an *abort* dependency between T_{w4} and T_{w2}. This means that if T_{w2} aborts then T_{w4} must abort.

From this example, we can see that dependencies are powerful in supporting the cooperation feature of complex applications. The structure of dependencies are fundamental and crucial to advanced transactions.

1.2 Problem Description and Motivation

Advanced transactions are powerful extensions to traditional transaction models. They have been applied in complex applications, like EJB and OMG CORBA object services [85, 96]. However, there are a number of open issues to be solved with the existing advanced transaction models.

To address the shortcoming of traditional transaction model on advanced applications, many extended transaction processing models have been proposed [7, 28, 29, 32, 37, 43, 49, 78, 95, 99, 116]. These models are capable for executing different types of complex applications. Nevertheless, most of these models only work on specific environments, and their applicability is very limited. To make the proposed reliable transaction processing model applicable in variety environments, a unified model to express different applications is greatly needed. Therefore, we will propose the notion of generalized advanced transaction. It will enable us to represent different advanced transactions uniformly, and provide a system that will allow execution of different advanced transactions.

Dependencies are the key components of advanced transactions. Different types of control-flow dependencies are specified by the application developers to construct various advanced transactions. However, dependencies can be specified incorrectly. For example dependencies may conflict with each other. A pair of conflicting dependencies requires contradicting operations in deploying, and they will result in denial-of-service and/or unavailability issues for advanced transactions. For instance, a *strong-commit* dependency $T_{wi} \rightarrow_{sc} T_{wj}$ and a *exclusion* dependency $T_{wi} \rightarrow_{ex} T_{wj}$ on the same pair of subtransactions is conflicting, since the first one requires the subtransaction T_{wj} to commit while the other one requires T_{wj} to abort. This could lead to unpredictable situations in execution, and will further cause a deadlock. The advanced transactions must be structured in correct form. Unfortunately, there is no existing work on the correct structure of advanced transactions. Properties of various kinds of dependencies, classification of dependencies, different forms of dependency conflicts, interactions of dependencies, and the impacts of dependencies on the reliable execution of advanced transactions are not discussed in literature.

Correct execution of advanced transaction is extremely crucial. ACID properties cannot guarantee the correct execution of an advanced transaction, since they do not address the dependencies existing in advanced transactions. The scheduler for advanced transactions must ensure satisfaction

of every dependency in execution. This is a big challenge compared to the traditional transaction model. How to correctly enforce different dependencies and properly deploy the advanced transactions becomes a major concern. Building a *reliable scheduler* for the advanced transaction system is very necessary. The reliable scheduler takes correctly structured advanced transaction as input. By correct structure, we mean there are no conflicting dependencies. It should be able to execute the advanced transactions correctly – choose the proper actions and sequences to schedule all the subtransactions, and at the same time preserve all the dependencies. System crashes will occur in all real-world systems, and they cannot be neglected. In the event of a system crash, the advanced transaction system must be brought back to a consistent state. We will need some effective failure recovery techniques, not only to restore the data items to consistent states, but also to ensure the satisfaction of dependencies among all subtransactions. A number of existing papers consider the failure recovery problem for workflow systems, however, they failed on how to ensure satisfaction of all dependencies during recovery for advanced transaction systems.

Although intrusion detection and prevention mechanisms are effective in some applications, malicious attacks cannot be completely prevented. Attackers can intrude into the advanced transaction system and inject some damage for personal benefits. Damage assessment and repairing techniques are crucial for survivability of advanced transaction applications. How to survive the malicious attacks is a very serious problem that must be solved. Damage assessment and repair algorithms are needed to remove the damage caused by malicious attacks. In repair process, appropriate actions need to be taken to ensure the satisfaction of all dependencies. Existing papers in literature do to address these issues. Techniques for damage assessment and recovery in database systems are not adequate for advanced transactions. An advanced transaction consists of subtransactions that have control-flow dependencies specified among them. The presence of these dependencies require new techniques for damage assessment and recovery. An example will help illustrate the idea. Suppose a malicious attacker creates a subtransaction T_{wj} in an advanced transaction that decrements an account balance $acct_x$ to \$100. On successful completion of T_{wj}, a new subtransaction T_{wk} is initiated that charges a maintenance fee because the account balance in $acct_x$ has fallen below the designated limit of \$200. A subtransaction T_{mn} of a different transaction reading the incorrect value in $acct_x$ and using this to compute $interest_x$, $total\_bal$, corrupt these other data items as well.

The traditional database survivability techniques will restore the balance in $acct_x$ and compute the correct values of data items that depended on $acct_x$. But it will not remove the effects of affected subtransactions, such as T_{wk}, that got initiated because of the malicious subtransaction.

Access control policies are security policies that govern access privileges to resources. Users executing a subtransaction must have the adequate privileges to perform the execution. Real-time update of access control policies is necessary for many security-critical and dynamic applications. For the advanced transaction system to continue functioning in such situations, the policies must be changed immediately and the modified policies automatically enforced. Updating policy objects in ad-hoc manner can lead to potential security problems like information breach; moreover, violating of dependency during policy update will cause information integrity and unavailability problems. It is important to control the manner in which security policies are updated. Real-time updates of access control policies is needed in commercial environments like business workflow systems. Suppose the user *John*, by virtue of some policy P, has the privilege to execute a long-duration subtransaction T_{wx} (which can be seen as a task in a workflow) that prints a large volume of sensitive financial information kept in file I. While *John* is executing this subtransaction, an insider threat is suspected and the policy P is changed such that *John* no longer has the privilege of executing this subtransaction. Since existing access control mechanisms check *John*'s privileges *before John* initiates the subtransaction and not *during* the execution of the subtransaction, the updated policy P will not be correctly enforced causing financial loss to the company. In this case, the policy was updated correctly but not enforced immediately resulting in a security breach. There is no existing paper on the issues of real-time update of policies in advanced transaction system.

The research described in this document is motivated by the following limitations of the existing advanced transaction models.

1. Most existing advanced transaction models are only suitable for a specific environment. We need a generalized transaction model to express different advanced applications in an uniform manner.

2. Dependency is the most important feature of advanced transactions. Nevertheless, properties of dependencies, conflict of dependencies, interaction of different dependencies, and impacts

of dependencies on execution are not completely studied in literature. Advanced transactions must be correctly structured and the dependency properties must been analyzed.

3. Due to the existence of dependencies in coordinating the subtransactions, correctly execution of advanced transactions becomes a challenging task. A reliable transaction scheduler is needed, and it should provide mechanisms to ensure the satisfaction of all dependencies and correctly deploy all the subtransactions.

4. Recovery from system crash and failures is a serious problem that cannot be neglected. System crash will leave the system in a inconsistent state. We need reliable failure recovery techniques to restore the system to consistent states and ensure the satisfaction of dependencies after recovery.

5. Advanced transactions could suffer from malicious attacks. Survivability must be considered in a reliable transaction model. Effective damage assessment and repair algorithms are needed to solve the malicious attack problems.

6. Real-time update of access control policies are important for dynamic environments. Update policies in ad-hoc manner will cause information breach and integrity problems. Effective mechanisms are needed to update policies in a secure manner and to provide high level of concurrency control.

Existing papers in literature have not addressed these issues listed above.

1.3 Research Goals and Tasks

Motivated by the open issues discussed previously, I propose a reliable and survivable transaction processing model for advanced transactions in this dissertation. The goal of this research is to build a reliable transaction processing model, which will enable us to express advanced transactions correctly and uniformly; and more importantly, it will be able to execute and deploy the advanced transactions in a reliable and survivable manner.

To achieve this goal, I decompose this research work into a set of tasks –

1. propose a generalized advanced transaction model to express different advanced transactions uniformly and correctly;

2. study the properties of dependencies, analyze the correct structure of advanced transactions, and propose well-structured advanced transactions as a criteria to construct error-free advanced transactions;

3. build reliable scheduler to deploy the advanced transactions in a reliable manner and correctly enforce the dependencies; propose reliable recovery strategy to restore from failures, bring the system back to consistent states and ensure satisfaction of dependencies;

4. propose repair algorithm to survive malicious attacks, in face of attacks, provide efficient damage assessment and repair the system effectively;

5. provide means for real-time update of access control policies in advanced transactions, not only to update the policy in real-time, but also ensure satisfaction of dependencies.

The set of tasks are described in details in the following sections.

1.3.1 Task 1. Defining Generalized Advanced Transaction as an Unified Model

The first task in this research is to define generalized advanced transactions to represent advanced transactions uniformly. The proposed generalized advanced transaction needs capture the properties of most existing advanced transaction models and it should be suitable for different environments. To achieve this goal, I will start from studying existing advanced transaction models. Then I will analyze the common features of these models, and propose the notion of generalized advanced transactions. It should be not only suitable for existing advanced applications, and also be able to express yet-to-develop advanced transactions.

The work in this task is listed below briefly.

1. Analyze the properties of different advanced transactions,

2. Propose notion of generalized advanced transaction,

3. Define the set of dependencies that we will use in the model

How to express different extended transactions uniformly is the major problem in this task. Various advanced transaction models extend traditional transactions by new concepts which relax the ACID properties and add new primitives to transactions. For example, nested transactions and SAGAS are the earliest advanced transaction models. In nested transaction models, top-level transactions have all the ACID properties of traditional transactions; subtransactions, on the other hand, are atomic but they share data with their parent, so they are not fully isolated. Moreover, subtransactions are not durable, because abort of their parent will cause their own abort. Based on these analysis, we can summarize the two important properties of most extended transactions: (i) contain long duration computations, and (ii) subtransactions are interactive with each other in different ways. Dependencies are the effective means to capture both these two properties and they generate different advanced transaction models. We will define different kinds of control-flow dependencies that will be used in the generalized advanced transaction models. The control-flow dependencies define the relationships that may exist between different events (such as, begin, commit or abort) of two subtransactions in a generalized advanced transaction.

Based on these analysis, I will propose the notion of generalized advanced transactions. In the generalized advanced transaction model, subtransactions are low-level coordinated activities, and they form the smallest unit in coordination of the advanced transaction. A generalized advanced transaction also contains a set of dependencies to coordinate the execution of these subtransactions. And, it will also contain a set of completion sets, which specify the execution termination states for advanced transactions.

Technical challenges in this task are summarizing the common properties of all existing advanced transaction models, and modeling the generalized advanced transaction as a unified form. How to organize the subtransactions and control-flow dependencies as key components in different advanced transactions uniformly is the major concern in this task.

1.3.2 Task 2. Analysis for Well-structured Advanced Transactions

Before we can start to build a reliable execution model for advanced transactions, properties of dependencies need to be thoroughly investigated. Some important features of dependencies have not been studied in existing papers. For example, dependencies could be transitive and symmetric.

Moreover, one dependency can interact with other dependencies and they can imply some implicit relationships. An important goal in this research is to analyze the properties of dependencies and define the correct structure for advanced transactions. Some simple analysis are the basic properties of dependencies and classifications of dependencies based on their features. For example, a dependency of type x is said to be *symmetric* if $T_{wi} \rightarrow_x T_{wj}$, then $T_{wj} \rightarrow_x T_{wi}$ for all T_{wi}, T_{wj}. Similarly, a dependency of type x is said to be *transitive* if $T_{wi} \rightarrow_x T_{wj}$ and $T_{wj} \rightarrow_x T_{wk}$ implies that $T_{wi} \rightarrow_x T_{wk}$. Moreover, one dependency can include the effects of another dependency and the inclusion relationship is established. For instance, the *begin-on-commit* dependency includes the constraints of the *begin-on-begin* dependency. We should avoid to specify these two dependencies over the same pair of subtransactions, because they will cause *dependency redundant* problems in execution.

Based on the different semantics of the dependencies described above, we will classify them into two major categories: *event enforcement* type, and *event ordering* type. An event ordering dependency $T_{wi} \rightarrow_x T_{wj}$ is one in which the execution of some event in subtransaction T_{wi} must precede the execution of some event in subtransaction T_{wj}. An event enforcement dependency $T_{wi} \rightarrow_x T_{wj}$ is one in which the execution of some event in subtransaction T_{wi} requires the execution of some event in subtransaction T_{wj}. Different event ordering and event enforcement dependencies in an advanced transaction often interact in subtle ways resulting in conflicts and redundancies.

A more serious problem is that, dependencies on advanced transactions could be conflicting and in turn cause unpredictable behaviors of system. All types of dependency conflicts must be studied and removed from the advanced transaction specifications. For instance, a *begin* dependency $T_{wa} \rightarrow_b T_{wb}$ will be conflicting with another *begin* dependency $T_{wb} \rightarrow_b T_{wa}$. Due to the constraints of these two dependencies, neither T_{wa} nor T_{wb} can start earlier than the other. Hence they have to wait for each other to start infinitely. This pair of dependencies will lead to deadlock in execution. This is only one cause for the conflicts. Conflicts can arise in different forms:

1. the operations they required may be contradicting each other;

2. the order of operations may also conflict.

In either of these cases, it will lead to problems in execution of the advanced transactions.

Often times, one single dependency is not powerful enough in specifying the relationship between two subtransactions. For example, if we want to specify that T_{w1} must begin after T_{w2} has committed; and, if T_{w2} aborts then T_{w1} must also abort, at this time, one single dependency is not sufficient in expressing the coordination between T_{w1} and T_{w2}. We can use *composite dependency*, which is more powerful than a single dependency, under these situations. I formalize the notion of composite dependency in this dissertation. A composite dependency between a pair of subtransactions T_{wi}, T_{wj} in a generalized advanced transaction model, denoted by $T_{wi} \rightarrow_{d_1,d_2,\ldots,d_n} T_{wj}$, is obtained by combining two or more primitive control-flow dependencies $d_1, d_2, \ldots, d_n$. The effect of the composite dependency is the conjunction of the constraints imposed by the individual dependencies $d_1, d_2, \ldots, d_n$. When multiple dependencies are specified between a pair of transactions, we need to ensure that no dependency is redundant and the dependencies do not impose conflicting requirements.

I analyze all forms of dependency conflict. There are several situations that conflicts could possibly rise:

1. a pair of primitive dependencies can be conflicting to each other when composing a composite dependency;

2. conflicts can occur between event ordering dependencies – infeasible ordering required by different event ordering dependencies;

3. conflicts can also occur between event enforcement dependencies – conflicting enforcement requirements placed by event enforcement dependencies;

4. finally, conflicts can also occur between event ordering dependencies and event enforcement dependencies – because some event enforcement dependencies imply an event ordering, and some event ordering dependencies prohibit some event.

Detection of conflicts require us to draw the graph for the generalized advanced transactions. We draw all the edges in the graph including both the explicitly given edges and all the implicit edges. These implicit edges can be implied due to the symmetric or transitive nature of dependencies, they can also be implied by the interactions among different dependencies. In this study, all forms of dependency interactions will also be analyzed.

Advanced transactions must be constructed in correct structures. The example of deadlock problem disclosed that improper dependency structure will cause problem for execution of advanced transactions. Prior to building a reliable scheduler, we must have a clear definition for the correct structure of advanced transactions. In this task, I formally define and analyze the features for *well-structured* transactions. A well-structured advanced transaction should have all its dependencies structured properly – contains no conflicting dependencies and no redundant dependencies. Moreover, a well-structured transaction needs to have clearly defined completion states. I propose well-structuredness properties as criteria to specify dependencies correctly.

In task two, properties of dependencies and well-structured advanced transactions will be analyzed. I also classify control-flow dependencies into subclasses based on their features, and discuss the interactions among these dependencies. More importantly, I study all kinds of dependency conflicts and propose ways to detect and remove them. The work in this task is listed below.

1. Analyzed dependency classification, simple property, composite dependencies, and dependency interactions,
2. Investigate dependency conflicts,
3. Propose algorithms to detect and remove dependency conflicts,
4. Propose well-structured advanced transactions

The biggest technical challenge in this task is how to identify all forms of conflicts on different kinds of dependencies, and build algorithms to detect and remove all dependency conflicts and redundancies.

1.3.3 Task 3. Building a Reliable Scheduler and Recovery Manager

Prior to constructing the reliable scheduler, impacts of dependencies on execution of advanced transactions must be thoroughly analyzed. A reliable scheduler for advanced transactions must be able to enforce all dependencies correctly during normal execution. It accepts well-structured generalized advanced transactions, and should be able to execute the subtransactions correctly and ensure the satisfaction of different dependencies.

Preserving the constraints for every single dependency in execution is an interesting problem. Very few existing paper in literature addressed the effects of dependencies in execution of advanced transactions. I will start by analyzing the executing states of subtransactions. A subtransaction can take different states in execution – namely, unscheduled (un_{wi}), initiation (in_{wi}), execution (ex_{wi}), prepare (pr_{wi}), committed (cm_{wi}), and aborted (ab_{wi}). I will analyze the proper actions needed to schedule a pair of subtransactions that are associated with different control-flow dependencies. For each dependency between any two subtransactions T_{wi} and T_{wj}, we need to check out what actions the scheduler necessarily to adopt under the current execution states. I will associate every control-flow dependency with a *scheduling action table*. This table will show the execution actions that need to be adopted by the scheduler, based on states of the two subtransaction. The table entry in scheduling action tables can be presented in different types, like: (I) *no constraint*, means that both of the two subtransactions can go into next state without any restriction; (II) *delay, abort, reject, execute*, gives the specific actions need to be taken by the scheduler on one subtransaction to satisfy the dependency constraints; (III) *prohibited*, means that these two states are not possible for this control-flow dependency; and, (IV) *final state*, means that both of the two tasks have arrive the final states, and the two final states are acceptable. For each primitive control-flow dependencies, I will analyze the dependency execution table. These tables help to specify the execution order and the actions for the scheduler to take.

How to enforce a group of dependencies and/or composite dependencies is a more difficult problem. When considering the joint affects of several dependencies, we need to combine several scheduling action tables and consider the correct actions to be taken. These four types of table entries will be associated with different *priorities* in combination. Entry *no constraint* will have the lowest priority, and entry *constraint* has higher priority. When a scheduler finds *no constraint* entry in one table and a *constraint* entry in another table, it will take the special constraint in the *constraint* entry. If a scheduler meets *non-exist* entry, it can decide that these execution states are not possible for these two subtransactions with this set of dependencies. Moreover, *non-exist* entry will take a higher priority than the *final state* entry. In this task, we will analyze the correct priority order and show how it works. Above all, I will analyze the different execution stages of subtransactions, and propose a reliable execution model to deploy the dependencies correctly.

In designing a reliable transaction processing model, failure recovery issues must be taken into consideration. The scheduler proposed is supposed to enforce the dependencies during regular execution. However, enforcing dependencies during normal execution is not enough. In the event of failure or system crash, the transaction processing model needs to take effective failure recovery techniques. It must be able to restore the data items to consistent states, and more important, it should also ensure the satisfaction of all dependencies during recovery. In this task, I elaborate on different kinds of control-flow dependencies and analyze how these dependencies could impact the recovery process. I discuss what information is needed and how the information is stored to perform an effective recovery. An effective recovery algorithm will be given. I also provide correctness proof for the proposed recovery algorithm.

Detailed work needed in this task is listed below.

1. Study the impacts of dependencies on reliable execution of advanced transactions,
2. Discuss the complete execution criteria,
3. Enforce primitive dependency, composite dependency and a set of dependencies in execution,
4. Propose algorithms for reliable scheduler,
5. Propose reliable recovery manager, recovery algorithm and correctness proof

Technical challenges in this task lie in the analysis for correct scheduling action tables for every different dependency. The constraints of different dependencies in execution must be taken into consideration. Moreover, enforcing composite dependencies and ensure satisfaction of a group of dependencies is another difficult problem. The priorities specified among the scheduling actions must preserve all component dependencies.

1.3.4 Task 4. Survivability and Security Issues for Advanced Transactions

1.3.4.1 Damage Assessment and Repair for Malicious Attacks

Malicious attackers can break in to the advanced transaction system and cause serious damage. The malicious attackers can force transactions to behave erroneously. They can also forge some subtransactions, and trigger the execution of more unwanted subtransactions. As a consequence,

the malicious attackers can earn some personal benefit and and cause lose to the systems. Advanced transactions contain control-flow dependencies. The presence of these dependencies require new mechanism for damage assessment and recovery.

In this dissertation, I present algorithms to show how we can assess and repair the effects of damage caused by malicious subtransactions. Our algorithm focuses not only on restoring the consistency of data items by removing the effects of malicious subtransactions, but also takes appropriate actions to ensure the satisfaction of dependencies during repairing.

I start on analyzing the nature of various types of control-flow dependencies on damage spread and repair. Our repair mechanism focuses on the different impacts of these dependencies on attack recovery. The survivability algorithm that will be proposed in this dissertation provides more specific techniques in generating redo and undo subtransactions lists, based on the analysis of suspect and affected subtransactions. Discriminating the various types of control-flow dependencies in recovery is a key point in our survivability algorithm. For malicious attack repair purpose, I assume that we will have data item read record as well as write record in the log file. The read records will be useful in building up the read-write dependencies among subtransactions in the repair algorithm. In repairing, we will distinguish malicious subtransactions, read-write affected subtransactions (subtransactions that read from malicious subtransactions), control-flow dependency affected subtransactions (subtransactions that are affected by undo actions, referred as the undo-action affected subtransactions), and innocent subtransactions (execution and states have not been altered by malicious ones). We only consider the committed subtransactions in repairing. The aborted subtransaction and unfinished subtransaction will not be considered since they did not change the state of the database system. In this task I also provide correctness proof for the repair algorithms and show that they can remove all effects of malicious attacks – restore data to their correct values and preserve the dependencies.

The research work in these task is briefly listed below.

1. Study the impacts of malicious attacks on advanced transactions,
2. Propose damage assessment and repair algorithms,
3. Provide the correctness proof for effective repair

The major technical challenge in this task is to identify which dependencies spread damage in the advanced transactions. How to effectively assess damage and perform the repair, and to ensure the satisfactions of dependencies in repair are also important problems that must be solved.

1.3.4.2 Real-time Update of Access Control Policies in Advanced Transactions

Access control policies protect information resources from unauthorized access. In security-critical applications and dynamic environments, the access control policies may need real-time update due to security requirement.

In this task, I consider real-time policy updates in the context of an advanced transaction system. The advanced transaction system consists of a set of data objects that are accessed and modified through different subtransactions. Subtransactions performing operations on the data objects must have the privilege to execute those operations. Access control policies are stored in the form of *policy objects*. Subtransactions executing by virtue of the privileges given by a policy object is said to *deploy* the policy object. In addition to being deployed, a policy object can also be accessed and modified by some subtransactions. We are considering an environment in which different kinds of subtransactions execute concurrently some of which are policy update subtransactions. In other words, a policy may be updated while subtransactions are executing by virtue of this policy. Allowing one subtransaction to continue execute without adequate privilege will result in security breach.

Updating access control policies in database systems has been addressed in several earlier papers. However, in advanced transactions, the situation is more complicated. After a policy update, if the subject does not possess enough privileges to continue executing a subtransaction any longer, this subtransaction must abort. This strategy works well with traditional database systems, since the transactions in database systems are independent and the execution of one transaction does not affect any other transactions. However, the advanced transactions are different – subtransactions in advanced transactions are cooperative and their executions are coordinated by dependencies. In face of a policy restriction update, aborting the one subtransaction is not enough. After the abort event on one subtransaction, proper actions may be needed on more subtransactions in the same advanced transaction, in order to ensure the satisfaction of the dependencies defined among these subtransactions. In this task, I propose algorithms that update policies in real-time and also ensure

satisfaction of all dependencies.

1.4 Significance and Contributions

In this research, I propose a reliable transaction processing model that can be used to develop complex and powerful advanced transactions, and be able to deploy these advanced transactions correctly. This research is significant that it appears to be the first work in analyzing the dependency conflict properties; and it provides a generalized framework for analyzing dependencies and reliable execution of advanced transactions. This work will help the advanced transaction designer to build correctly structured transactions. The discussion on real-time update of access control policies is very important for dynamic environments. The advanced transactions in security critical applications will be benefit from this study.

Contributions of this research work are that it:

- enables the advanced transaction developers to express complex applications in form of generalized advanced transaction correctly and uniformly;

- provides means to analyze dependency conflicts in advanced transactions. It also gives algorithms to detect and remove the conflicts as well as redundancies in dependency specifications;

- provides mechanisms to specify well-structured advanced transactions, and provide criteria for completion execution;

- provides reliable scheduler and recovery manager to execute advanced transactions and enforce dependencies correctly in execution;

- provides survivability mechanism to survive malicious attacks;

- discusses the real-time update of access control policies in database systems and advanced transaction systems.

1.5 Structure of Dissertation

The rest of this dissertation is organized as follows.

Chapter 2 describes existing research work in the area of advanced transactions and secure transaction processing for advanced transaction models.

Chapter 3 presents the generalized advanced transaction model and different types of dependencies.

Chapter 4 discusses the properties of control-flow dependencies, like dependency inclusion relationship, composite dependencies, minimality properties etc, and also studies the interaction of control-flow dependencies. This chapter will also discuss different types of dependency conflicts that could exist in specifications, and proposed algorithm to detect and remove these conflicts.

Chapter 5 presents the reliable execution model and the scheduling algorithms for advanced transactions.

Chapter 6 describes the failure recovery strategies and the recovery algorithms.

Chapter 7 discusses the damage assessment and repair mechanisms for advanced transactions.

Chapter 8 presents the algorithms for real-time updating of security access control policies in database and advanced transaction systems.

Chapter 9 concludes this dissertation.

CHAPTER II

LITERATURE REVIEW

2.1 Traditional Database Transactions

Reliable and correct execution of program is a key concern in the computing and engineering world. A transaction is normally made up of a set of operations, which are closely related to each other, as one single unit of work. The reliability and correctness properties must hold for a transaction. In the traditional transaction model, the ACID properties ensure the correctness in execution [16, 46]. The acronym ACID stands for atomicity, consistency, isolation, and durability. To ensure predictable behavior, all transactions must possess these basic properties, reinforcing the role of mission-critical transactions as all-or-none propositions.

Atomicity: A transaction must be atomic, either all of the work is done or none of it is. Operations within a transaction usually share a common intent and are interdependent. By performing only a subset of these operations, the system could compromise the overall intent of the transaction. Atomicity eliminates the chance of processing only a subset of operations.

Consistency: A transaction must preserve the consistency of data, transforming one consistent state of data into another consistent state of data. The responsibility for maintaining consistency lies with the application developer.

Isolation: A transaction must be a unit of isolation, which means that concurrent transactions should behave as if each were the only transaction running in the system. Because a high degree of isolation can limit the number of concurrent transactions, some applications reduce the isolation level in exchange for better throughput.

Durability: A transaction must be recoverable and therefore must have durability. If a transaction commits, the system guarantees that its updates can persist in spite of system failures. Durability is ensured by the recovery mechanisms.

The atomicity and durability properties define the notion of reliability in the context of transaction. They require that a group of operations is either executed as a whole unit, or none of the operations are executed; and, the completed state change made by the group of operations are guaranteed to be persistent. The consistency and isolation properties define the notion of correctness in execution of transactions. A transaction must transform the system from one consistent state into another consistent state; and, any intermediate states of an ongoing transaction are not visible to any other transactions that are concurrently executing.

2.2 *Advanced Transaction Models*

In the past two decades, a variety of transaction models and technologies supporting advanced transaction processing have been proposed [7, 15, 18, 24, 26, 28, 29, 30, 31, 32, 33, 37, 42, 43, 49, 54, 67, 73, 75, 78, 95, 99, 109, 114, 115, 116]. Examples are ACTA [28, 29, 30, 31, 32], ConTracts [93], nested transactions [78], ASSET [24], EJB and CORBA object transaction services [80], workflow management systems [2, 8], concurrency control in advanced databases [16] etc. These models consider dependencies among the component transactions, and they are capable for complex applications.

ACTA

Chrysanthis and Ramamrithan introduce ACTA [28, 29, 30, 31, 32], as a formal framework for specifying extended transaction models. ACTA allows intuitive and precise specification of extended transaction models by characterizing the semantics of interactions between transactions in terms of different dependencies between transactions, and in terms of transaction's effects on data objects. During the past few years, ACTA has become a standard for formal specifying and reasoning about advanced transaction models.

In ACTA, Chrysanthis and Ramamrithan organize the notion of history into object events and significant events executed on behalf of transactions. An object event is an invocation of an operation upon a data object. A significant event is an invocation of a transaction processing primitive – transaction begin, commit or abort.

ACTA is a comprehensive transaction framework for specifying extended transactions. It can be

used to formally define the properties of extended transaction models. Using ACTA, one can specify the effects of the transaction on objects, and one can analyze the interactions between transactions. Chrysanthis and Ramamritham discussed the ACTA framework with five building blocks: history, dependencies between transactions, the view of transaction, the conflict set of a transaction, and delegation. They express several extended transaction models with the ACTA model, including SAGA, nested transactions, chain transactions, and split/join transactions [87].

In recent years, ACTA has become a standard for formal specification and analysis of advanced transaction models. In ACTA, advanced transaction is described as a set of operations on objects that execute atomically in a predefined order, or, a set of extended transactions with an explicitly given control related to the notions of visibility, consistency, recovery, and permanence. ACTA model is so powerful in defining extended transactions. Its power stems from the ability to represent the structural behaviors of transactions and the dependencies between them. In paper [32], the authors present ACTA as a tool for the synthesis of different extended transaction models. They show that the ACTA framework can support the development and analysis of new extended transaction models in a systematic manner. Synthesis of advanced transaction models is demonstrated by deriving new transaction definitions, either by modifying the specifications of existing transaction models, or by combining the specifications of existing models. The authors also gave several examples, showing how new models can be synthesized from atomic transactions and join transactions.

Unfortunately, properties of various kinds of dependencies, conflict of dependencies, and impacts of dependencies on reliable execution of advanced transactions are not discussed in ACTA. Dependencies in advanced transactions must be structured in correct form. ACTA fails to identify these issues.

SAGAS

Garcia-Molina and Salem discussed the SAGAS model in an early paper [42]. The SAGAS model offers a linear structure for organizing a set of transactional activities. The model is very useful especially when the subtransactions are relatively independent to each other, and if each subtransaction can be successfully compensated. In SAGAS, the dependency structure is very simple. They are used for the compensation of an unsuccessful execution. Moreover, dependency properties are

not studied in ACTA.

ASSET

ASSET [24] provides a set of transaction primitives extending a programming language. Beyond the traditional transaction primitives (e.g., begin, commit, abort, get_parent) it introduces new primitives allowing creation of dependencies between transactions, resource delegation, and giving permissions for an access to acquired resources. However, no implementation based on a real architecture is shown and interfaces of the proposed primitives are not defined precisely. None of the provided examples uses dependencies; instead, dependencies are implicitly modeled by flow control constructs.

Multiform Transactions

Mancini, Ray, Jajodia and Bertino have proposed the multiform transactions [73]. A multiform transaction consists of a set of transactions and includes the definition of a set of termination dependencies among these transactions. The set of dependencies specifies the commit, abort relationship among the component transactions. The multiform transaction is organized as a set of coordinate blocks. Each coordinate block will have its transaction component and its protocol component, which include a definition of the dependencies among the transactions. The coordinate block, along with the corresponding coordinator module (CM) can manage the execution of the transactions.

The multiform transaction model separates the execution order from the completion dependencies of the transactions. It gives more expressive power to the model in specifying dependencies.

Various extended transaction models can be specified as multiform transactions, including SAGAS, workflows, long lived activities, secure distributed transactions, contingent transactions, nested transactions, and the ACTA framework. In the paper, they also give a high level declarative language that can be used for specifying transactions and completion dependencies.

This is a generalized form of advanced transactions. But, properties of different kinds of dependencies, classification of dependencies, conflict of dependencies, and impacts of dependencies on reliable execution of advanced transactions are not discussed. The discussion on well-structured transaction in this paper only focused on orphan transactions. They did not consider other issues in defining a well-structured advanced transaction.

Workflow Management Systems

A large body of research exists in the area of workflows [10, 38, 44, 60, 62]. Some research [13, 44] has focused on how to specify workflows and how to ensure the correctness of the specification. Other researchers [10, 13] studied how to control the execution of tasks in a workflow so as to satisfy the dependencies.

Workflows are composite activities typically involving a variety of computational and business activities. Workflow is a type of advanced transaction model, since all the workflow models support coordination of sub-units of activities. The importance of workflow models is increasing rapidly due to the business application forces. For the above reasons, Workflow Management Systems (WFMS) recently have gained a lot of attention [2, 8, 9, 11, 44, 52, 53, 55, 56, 59, 60, 63, 68, 92, 95, 97, 98, 99]. They are responsible for coordinating the execution of multiple tasks performed by different entities within an organization. A group of such tasks that form a logical unit of work constitutes a workflow. To ensure the proper coordination of these tasks, various kinds of dependencies are specified between the tasks of a workflow. The scheduler of a workflow is expected to enforce these dependencies during normal execution of a workflow.

Workflow can be easily modeled with the advanced transaction models, like ACTA framework [28, 31, 32], and the multiform transactions [73]. The common properties between workflow and advanced transaction models are that they have cooperating sub-tasks as the subtransactions, and they use dependencies to coordinate the execution of the sub-tasks. Thus, we can see workflow as an instance or a special form of an advanced transaction model.

EJB, CORBA OTS Models

The paper by Prochazka and Robinson [84, 85, 86, 96] discussed the EJB transaction model and proposed Bourgogne transactions. The authors also identified a number of weakness of the model. Prochazka claims that the EJB model has the weakness in transaction coordination and component architecture, as inherited from the traditional transaction model. For example, Object Management Group (OMG) has discussed the OTS specification for CORBA activities and services [80]. The OTS specification defined the structure of the activities and is an attempt to standardize extended transaction models. In [106], Shwarz, and Spector discussed type specific concurrency control.

They defined *Synchronizing Shared Abstract Types* in their paper, to loose the requirement of concurrent accessing to data. Concurrent read/write and write/write operations can be permitted on a data item from different transactions as long as these operations can be shown to be non-interfering. Barghouti and Kaiser [16] discuss the concurrency control issues of the advanced database applications. Nevertheless, these approaches are limited to a specific environment, and they are not aiming towards a generalized transaction model for advanced transactions.

2.3 Dependencies in Advanced Transactions

A number of papers in literature have discussed the inter-task dependencies for workflow systems and also about the issues in enforcing dependencies in execution.

Form of Dependencies

Chrysanthis and Ramamritham have pointed out that, in generalized advanced transactions, dependencies could be formed due to several causes [28, 30, 31]. Dependencies between subtransactions may be: (i) direct result of the structural properties of advanced transactions, in order to coordinate the cooperation of sub-tasks; (ii) indirectly developed as a result of interaction of subtransactions over shared objects.

The structure of an advanced transaction defines its component subtransactions and the relationship between them. Dependency is an effective means to express and coordinate these relationships. Various types of dependencies can be used to specify the inter-task structures of an advanced transaction. They can also be used to syntheses the structure of many existing advanced transaction models. For instance, in the *nested transaction* model, the *parent/children* relationship is established when a new child is *spawned.* This relationship can be captured with control-flow dependencies very easily. We can express this relationship with the following set of dependencies: (1) a child subtransaction t_c establishes a *weak-abort* dependency on its parent t_p, as $t_p \rightarrow_{wa} t_c$. (2) the parent subtransaction establishes a commit dependency on this child, as $t_c \rightarrow_c t_p$.

The weak abort dependency guarantees that the abortion of an uncommitted child if its parent aborts in history. Note that in nested transactions, when a child transaction commits, its effects are not made permanent in database but only visible to its parent. Thus, the weak abort dependency

does not prevent a child subtransaction from committing and making its effects visible to its parents. The commit dependency of the parent on its child is used to coordinate the order of commit event for parent and child subtransaction. The weak abort dependency and commit dependency together establish the relationship for a nested transaction.

We can see that the nested transactions can be easily structured using the control-flow dependencies. The synthesis of nested transaction relationship is specified in [31] in terms of the postcondition of a *spawn* event: $post(spawn_{t_p}[t_c]) \Rightarrow (((t_p \rightarrow_{wa} t_c) \in D) \bigwedge (t_c \rightarrow_c t_p) \in D)$

Other hierarchically-structured advanced transaction models can also be modeled with the dependencies, though they may have various relationship between parent and its children subtransactions.

Dependencies could also be formed by the interactions of subtransactions over a shared data item. Two operations are conflicting if they are to access the same data item and at least one of them is a write operation. A conflicting relationship can be modeled with dependencies, as that, if transaction T_{wi} invokes an operation p and later on another transaction T_{wj} invokes conflicting operation on the same data item x, then we could develop a dependency to coordinate the two operations.

Singh has discussed the semantical inter-task dependencies on workflows [109]. The author used algebra format to express the dependencies and analyze their properties and semantics in workflow systems.

Unfortunately, none of these papers address the dependency interaction, nor do they discuss the composite dependencies and problems arising out of conflicting dependencies.

Dependency and Deadlock in Advanced Transactions

In Bertino, Chiola, and L. V. Mancini's recent paper [22], the authors discussed the deadlock detection issues in advanced transaction models. Deadlock detection is fairly well-understood for the traditional transaction model. When dealing with advanced transaction models, the deadlock issues become more complex due to the existence of different types of transaction dependencies and data dependencies. Commit and abort dependencies specifying constraints on transaction termination order, and data dependencies arise when transactions concurrently access common data items under

conflicting modes. In the paper, the authors show that in the face of these dependencies, deadlocks may arise that the conventional deadlock detection algorithms are not able to detect. They discussed transaction waiting states characterized by AND-OR graphs, and proposed an algorithm for detecting deadlocks in these graphs. The authors claimed that their algorithm has a computational complexity linear in the number of nodes and edges of the AND-OR graphs. This paper is an early work in discussing deadlock raised due to dependencies in advanced transactions. However, the author only paid attention to termination dependencies (including commit and abort dependency), they did not discuss the impact of different kinds of control-flow dependencies in forming of deadlocks in advanced transactions. Moreover, properties of dependencies are not studied in this paper; how the interaction of different kinds of dependencies will affect the deadlock issues is not discussed.

Enforce of Dependencies and Execution of Workflows

There are also a number of papers appearing on analysis and execution of workflow systems.

Adam, Atluri and Huang [1] have discussed modeling and analysis of workflows using Petri Nets. The authors show how to use Petri Nets to model the workflow system at a conceptual level. They identify some structural properties of PN (Petri Nets) and demonstrate the use of PN on analysis and verification of workflow specifications. The authors discussed different control-flow dependency types – strong-causal type, weak-causal type and precedence type. They also mentioned value dependencies and temporal dependencies. The authors show how to use PN to represent control-flow, value and temporal dependencies; they also demonstrate how to use PN to conduct analysis on workflow structures – like inconsistent dependency specifications, terminable with an acceptable state, feasible to complete execution in temporal constraints. They give the algorithm to build a TCPN (temporal constraint PN) and show how to use it to perform the analysis. This work is very useful, it demonstrates PN as an effective tool for modeling and analysis of workflows. However, control-flow dependencies can be conflicting to each other, and they have more impacts on execution of workflow. The authors did not address these issues. How to reliably schedule an advanced transaction and enforce the dependencies are not studied as well.

J. Tang and J. Veijalainen [111] proposed a way to enforce intertask dependencies in transactional workflows. In an earlier work, Paul C. Attie addressed the same problem. Attie at el. [13]

discussed means to specify and enforce intertask dependencies. They illustrate each task as a set of significant events (start, commit, rollback, abort). Intertask dependencies connect these events of various tasks and limit the occurrence and temporal order of them. In [59], Mohan Kamath has discussed the correctness issues in workflow execution and workflow system management.

2.4 Recovery and Failure Handling for Advanced Transactions

Although there is large body of research existing in the area of workflows, workflow recovery has received very little attention. Very few existing paper studied the issues of failure recovery in advanced transaction models. Most of the research [20, 26, 33, 38, 60, 62, 63] in workflow recovery focuses on how to restore a consistent state after a failure and they do not address the issue of enforcing task dependencies.

In the paper [78], J. Moss proposed ways for a nested transaction to support better control over recovery and concurrency. In the traditional transaction model, when a transaction fails, the system normally provides only a simple rollback of the entire transaction. In a long-running extended transaction environment, the author try to avoid the costing work to be performed in case of failures. In nested transaction model, the outermost transaction is referred as the top-level transaction. The permanent effect property is possessed only by the top-level transaction. The commit of nested transactions (can be seen as an example of the subtransactions) are provisional upon the commit/abort of the enclosing top-level transaction. In advanced transaction model, a transaction could have several options in case of one of its subtransactions fails. It can ignore the abort if the subtransaction is non-vital; it can retry the same transaction; it can submit an alternative; or, it can self-abort. This brings more advantage in failure handling.

In P. Attie et al. paper [13], the authors addressed the scheduling and recovery issues for transactions with dependency structures. They illustrate each task as a set of significant events (start, commit, rollback, abort). Intertask dependencies connect these events of various tasks and limit the occurrence and temporal order of them. They present formal specification of intertask dependency by using Computation Tree Logic (CTL). The enforceability of dependencies is determined by event attributes (forcible, reject-able, and delayable). Scheduler can allow, delay, reject or force occurrence of events to enforce intertask dependencies. The model's recoverability did not get much

attention. They briefly point out what data is needed to store for recovery of scheduler from a failure. A persistent communication mechanism is also needed to store messages between scheduler and tasks. But this approach does not describe how and where these data can be stored and how different data can be represented at log. They mention to use checkpoint mechanism, but do not give the more information about the implementation. To handle the system problems, such as disk crashes, etc. they use timeouts. The timeouts technique is more suitable for distributed workflow. The authors do not address recoverability issues in details. With respect to recovery, the authors mention what data is needed to recover from failures but do not provide details. They also identify that checkpointing and timeouts are needed for the recovery process but omit the details. Another important issue has been ignored – intertask dependencies still need to be enforced in the recovery procedure of workflow.

In paper [38], J. Eder and W. Liebhart classify workflow as document-oriented workflow and process-oriented workflow, and identify potential different types of failure. Based on various failure and different types workflow, they discuss the concepts for workflow recovery. All these concepts are used to achieve one goal, which is to restore the most recent consistent process state after a failure, so that as little long-duration work as possible is lost and that process execution can be continued. This paper gives an overview of workflow recovery concepts. It analyzes several recovery processes, such as forward recovery, backward recovery. It also points out the necessary requirement of WFMS for implementing workflow transactions and supporting workflow recovery. However, no detailed approach for workflow recovery processes is provided in this paper. When workflow recovery takes place, how to use these related information is not described in the paper. The constraints of tasks in a workflow, such as different kinds of control-flow dependencies are not considered here either.

Some approach provides systematic way of partially rolling back a workflow using cascaded rollback of a group of tasks. In the context of the FlowBack [62], the authors discussed backward recovery by using compensation process. Compensation can be applied to tasks and group of tasks (spheres) to support partial backward recovery. The creation of the compensation workflow can be static and dynamic. The static approach means the generation of the compensation is done by the workflow designer during the design phase. The dynamic approach means building the

compensation process as the original workflow executes step by step. When user wants to abort the original process, the compensation process will be executed. That paper provides an architectural design as well as implementation of FlowBack. This approach provides some solution for handling exception and cancellation of business processes. It only discusses the difficulties of the dynamic approach. Since the compensation process is predefined by the workflow designer, it could be very complex. It is not transparent and elegant solution to the user. Moreover, some transactions tasks may not be compensated due to the semantics of applications.

An integrated workflow failure handling approach called *opportunistic compensation and re-execution* was proposed by Kamath and Ramamritham [60] to handle failures and coordinate execution of concurrent workflows. In addition to the traditional workflow specification, it adds more specifications for failure handling, such as: the failure handling specification, re-execution specification, step conflict specification, etc. These specifications help to specify how to handle the failure at each individual task, how to re-execute each task after workflow partially rolls back, and how to handle task conflict and compensation dependency during workflow recovery. It eliminates unnecessary compensation and re-execution based on the task re-execution specification and current workflow state. It allows a workflow designer to customize workflow recovery with respect to both correctness and performance. This also requires the workflow designer has comprehensive knowledge about both the business process and the workflow model. To implement this strategy, there are some overheads that must to be considered. During the recovery process, it needs to check corresponding condition based on each task and the results of the execution of the previous tasks, to decide partial or complete compensation and incremental or complete re-execution. These overheads depend on the type of workflow and the type of tasks. These overheads add extra burden to the workflow designer. Another drawback is that their approach does not take into consideration of the effects of different types of control-flow dependencies during recovery.

Indrakshi Ray, Tai Xin and Yajie Zhu [92] have proposed a way to ensure the satisfaction of inter-task dependencies during recovery for workflow systems. The algorithm works not only on restoring consistency by removing the effects of partially executed tasks, and also addressing how to ensure task dependencies during workflow recovery. In that paper, each type of task is specified by (1) the types of inputs and outputs, if any, needed by tasks of this type, (2) recovery attributes

for this type of tasks, (3) compensation-flow for this type of tasks, and (4) alternate-flow for this type of tasks. Each type of task may need some input data and produce some outputs as the data flow dependency. The specification should include what types of inputs are needed and what types of outputs are produced by tasks of this type. Each type of task is described by attributes. Those attributes that are needed by the recovery process are known as the recovery attributes. Recovery attributes can be *compensatable*, *re-executable* and *alternate executable*. Compensatable identifies whether tasks of this type can be compensated or not. Re-executable indicates whether the tasks of this type can be re-executed or not. Alternate executable signifies whether tasks of this type can be substituted by alternate tasks or not. The compensation-flow of a type of task indicates what types of tasks need to be executed to compensate for tasks of this type. The alternate-flow specifies what type of tasks can be executed instead of this type of task. Each type of task has attributes which describe the nature of the task. The task attributes that are important for recovery are *compensatable*, *re-executable* and *alternate-executable*. The attribute *compensatable* identifies whether this type of task can be compensated or not. The attribute *re-executable* identifies whether this type of task can be re-executed or not. The attribute *alternate-executable* identifies whether this type of task can be substituted by some alternate tasks or not. For any type of task $ty(t_{ij})$ that is compensatable, the *compensation-flow* includes the set of types of tasks $\{ty(t_{ik}), ty(t_{il}), \ldots, ty(t_{in})\}$ that can compensate $ty(t_{ij})$, the task dependencies between the tasks in this set, and the inputs to the compensation process. For any type of task $ty(t_{ij})$ that is alternate executable, the *alternate-flow* includes the set of types of tasks $\{ty(t_{ik}), ty(t_{il}), \ldots, ty(t_{in})\}$ that can substitute $ty(t_{ij})$, the task dependencies between the tasks in this set, and the inputs to the alternate execution process. In the paper, the authors consider the workflow recovery problems and propose a recovery scheme that ensures the satisfaction of dependencies in a workflow and restores consistency as well. The authors elaborate on the different kinds of task dependencies present in a workflow and show how these task dependencies impact the recovery process. They also discuss what information is needed and how this information is stored to perform an effective recovery. A recovery algorithm was given to restore consistency and enforce task dependencies.

2.5 *Survivability for Advanced Transactions*

Some researchers have proposed Multi-level Secure (MLS) versions of the advanced transaction models [10, 11]. V. Atluri et al. proposed an execution model for multilevel secure workflows [10, 11, 12]. They also discussed the security access control issues in workflow management systems, with different means to enforce mandatory and discretionary security policies in workflow systems [9]. In another earlier paper, Atluri also discussed the authorization models for workflows [8]. Hung and Karlapalem proposed a secure workflow model in a recent paper [56], they focus on the authorization issues on the business workflow system, in authorizing access privilege for subjects to different objects. However, all those emphasis is still on the confidentiality issues in workflow systems. Not much work works towards the availability, information integrity, recovery, and survivability issues for advanced transaction application in commercial systems.

Survivability has received a lot of attentions in the database context. A group of papers in literature has discussed survivability in database and distributed database environments [3, 4, 5, 57, 65, 66, 69, 70, 82, 83, 110, 122].

Ammann et al. propose a two pass repair algorithm for traditional database systems in their paper [4]. The static algorithms consists of two passes. Pass one scans the log forward from the entry where the first bad task starts to locate every bad and suspect tasks. Pass two goes backward from the end of the log to undo all bad and suspect tasks. They also proposed a dynamic repair algorithm. Their paper focuses on the purely syntactic information about the interleaving of read and write operations. The control-flow dependencies existing with workflow systems and extended task models are not considered.

In recent years, the survivability for workflow system and advanced transaction models is gaining increasing concern. These advanced transaction applications are characterized by two properties - the need for complex coordination among different components/subtransactions, and long-lived computational durations. These two properties impose substantial security requirements on the transaction processing model.

In Gore and Ghosh's paper [45], the authors discuss the recovery and rollback problems in

distributed extended transactions. In the proposed model, the transactions communicate and collaborate only by exchanging messages. In recovery of distributed extended transactions, these message logs and message tables are extensively used. The log messages and the message tables are used in the normal processing, as well as in the event of partial and total rollback of the system. The drawback of this approach is that it is based on the special modified log structures, it is not a generalized model. Moreover, the features of transaction dependencies in any extended transaction model are not captured in this paper.

In a recent workflow survivability paper of M. Yu and P. Liu [120], the authors describe an algorithm for on-line attack recovery of workflows. The algorithm tries to build the list of redo and undo tasks, after an independent Intrusion Detection System reports malicious tasks. They also relax the restriction of executing order that exist in an attack recovery system; they introduced multi-version data objects to reduce unnecessary blocks in order to reduce degradation of performance in recovery. The drawback of this paper is that the authors treat all types of control-flow dependencies in the same manner. However, different type of control-flow dependencies require different treatment during recovery.

In this dissertation, I intend to present an algorithm to effectively assess and repair the effects of damage caused by malicious tasks. Our algorithm focuses not only on restoring the consistency of data items by removing the effects of malicious tasks but also takes appropriate actions to ensure the satisfaction of task dependencies among all the committed tasks.

This dissertation differs to the existing survivability papers in that, our algorithm starts by analyzing the nature of various types of dependencies. Our repair mechanism focuses on the impacts of different kinds of control-flow dependencies and read-write dependencies on damage assessment and attack recovery. The survivability algorithm that will be proposed in this dissertation provides more specific techniques in generating redo and undo subtransaction lists. This is based on the analysis for the suspect and affected tasks. Discriminating the various types of dependencies in recovery is a key point in our survivability algorithm.

2.6 *Update of Access Control Policies in Advanced Transactions*

A large body of work appears in the area of access control. Researchers have proposed access control models [19, 21, 101, 102], policy specification languages [17, 25, 36, 48, 51, 58, 81, 94], and techniques for identifying conflicts in policies [47, 71, 77, 107, 108]. Policy updates have received relatively little attention. Ray and Xin [90, 117] proposed algorithms for concurrent and real-time update of access control policies. The algorithms proposed consider only simple kinds of authorization policies where the only one access control policy is specified for any given subject and an object and only the rights associated with the policy is subject to change. In a subsequent work [91] the authors show how such algorithms can be implemented. The use of semantic knowledge for real-time update of access control policies was proposed by Ray [88]. In this work the author considered only simple kinds of authorization policies $P_i =< SS_i, TS_i, RS_i >$ where SS_i, TS_i, RS_i represents the set of subjects, the set of objects, and the set of access rights respectively. A policy can be changed by performing a series of set union or set difference operations. Depending on what kinds of operations were performed, the policy update is classified as policy restriction or relaxation. The author proposed algorithms that uses this knowledge to generate conflict serializable histories. The author also introduced the idea of how the application context can be used to check whether a policy update transaction interferes with a transaction that deploys the policy. However, a formal treatment of this idea is missing from the work. In a subsequent work [89], a more formal treatment of the semantic-based approach is proposed. This work also shows how transactions can be decomposed to minimize the effects of useless aborts. Semantic-based transaction processing comes at a cost – the cost is in terms of analysis needed to ensure correct behavior. Consequently, the current work uses a syntactic approach for policy update. None of the earlier works consider the case where there are multiple policies specified on a given subject and an object, priorities are specified on policies and the priorities themselves are subject to change.

Some work has been done in identifying interesting adaptive policies and formalization of these policies [40, 104]. A separate work [103] illustrates the feasibility of implementing adaptive security policies. The above works pertain to multilevel security policies encountered in military environments; the focus is in protecting confidentiality of data and preventing covert channels. We

consider a more general problem and our results will be useful to both the commercial and military sector.

Although a lot of work appears in the area of security policies (please refer to Damianou's thesis [36] for a survey), policy updates have received relatively little attention. Some work has been done in identifying interesting adaptive policies and formalization of these policies [40, 104]. A separate work [103] illustrates the feasibility of implementing adaptive security policies. rather limited and restricted to multilevel secure policies. The above works pertain to multilevel security policies encountered in military environments; the focus is in protecting confidentiality of data and preventing covert channels. We consider a more general problem and our results will be useful to both the commercial and military sector.

Automated management of security policies for large scale enterprise has been proposed by Damianou [35]. This work uses the PONDER specification language to specify policies. The simplest kinds of access control policies in PONDER are specified using a *subject-domain*, *object-domain* and *access-list*. The subject-domain specifies the set of subjects that can perform the operations specified in the access-list on the objects in the object-domain. This work describes the implementation of a basic toolkit. The toolkit has a high-level language editor for specifying policies, a compiler for translating policies into enforcement components objected to different platforms, a browser to view and manipulate the domains of subjects and objects to which policies apply. Thus, new subjects can be added to the subject-domain or subjects can be removed from the subject-domain. The object-domain can also be changed in a similar manner. But this work does not allow the policy specification itself to change. An example will help illustrate this point. Suppose we have a policy in PONDER that is implementing Role-Based Access Control: *subject-domain* = *Manager*, *object-domain* = */usr/local*, *access-list* = *read, write*. This policy allows all *Managers* to *read/write* all the files stored in the directory */usr/local*. Now the toolkit will allow adding/removing users from the domain *Manager*, adding/deleting files in the domain */usr/local*. However, it will not allow the policy specification to be changed. For example, the subject-domain cannot be changed to *Supervisors*. Our work, focuses on the problem of updating the policy specification itself and complements the above mentioned work.

Some work has been done in the area of dynamic and adaptive workflow systems which is related to this work. Researchers [27, 61, 74, 76, 79, 100, 113] have motivated the need for workflow systems that have the ability to adapt to the changing environments. Kammer et al. [61] have identified different kinds of exception handling capabilities for adaptive workflows. They also describe how some of these capabilities have been incorporated in the Endeavors workflow support system. Van der Aalst et al.[113] have mentioned that when a workflow is changed there are three ways to manage workflow instances that were active during the change: (a) restart, (b) proceed, and (c) transfer. Restart causes abort of all the tasks that were being executed under the original model. Proceed requires continuation of the workflow instance under the original model. Transfer requires the instance to be changed dynamically such that it complies with the new model. Sadiq [100] recognizes that changes to workflow model or workflow instances can lead to serious inconsistencies and errors. He points out that the problem becomes more critical when active instances are involved while the workflow model is being changed. The authors approach this problem by first formalizing how workflow can be modified, and then checking whether the instance initiated under the original workflow complies with the updated workflow. The instances generated from the old workflow may continue to follow the old model or they may be provided with a revised schedule to comply with the new model. The generation of revised schedule may need manual intervention. Mathews [76] has also addressed the problem of changing workflow policies while active workflow instances are in progress. He agrees that aborting active workflow instances prior to changing a workflow policies is inefficient and ineffective. To deal with this situation, he proposes SWAP, an agent-based architecture, that deals with changes in B2B workflow policies. In this architecture, the different agents keep track of the active workflow instances and their state of execution and are responsible for transitioning the workflow instance so that it complies with the new policy. Atluri et al. [8, 23] have proposed authorization models for workflow systems. They have proposed an approach for specification and enforcement of authorization constraints in workflow management systems. They addressed the workflow authorization model and separation of duties constraints in these papers. Muller and Rahm[79] suggested that dynamic workflow modification is needed for many applications, such as, medical domain. In such domains the large number of exceptions cannot

be handled by the domain except. They have developed a rule-based approach for detecting semantic exceptions that occur during instance execution. The control-flow of other instances affected by this exception needs to be modified dynamically. For this purpose, the authors have proposed two algorithms that carry out the dynamic change of control-flow. Although the problems have a similar flavor, the solutions for the problem of policy updates is very different. We are interested in preventing security breaches for dynamic policy updates without requiring any manual intervention. For making a workflow compliant with the new policy some amount manual task is involved.

Concurrency control in database systems is a well researched topic. Some of the important pioneering works have been described by Bernstein et al. [20]. Thomasian [112] provides a more recent survey of concurrency control methods and their performance. The use of semantics for increasing concurrency has also been proposed by various researchers [6, 14, 41, 50, 64, 72, 105].

In this dissertation, we address the problem of how policies can be updated in real-time and how the modified policies can be immediately enforced. We first discuss our work in the context of a database system. Then we extend this work to the context of generalized advanced transactions. We start with the simple case where a single policy is defined over a subject and an object and show how updates to such policy can be performed in real-time. In real-world, multiple policies may be specified over a given subject and an object. Priorities may be specified on these policies. We propose a mechanism that addresses the update of such policies where the priorities are also subject to change.

CHAPTER III

GENERALIZED ADVANCED TRANSACTIONS

The goal in this work is to build a reliable transaction processing model for the advanced transactions. Although a lot of extended transaction models have been proposed in past two decades, like ACTA [29, 31, 32], ConTracts [93], nested transactions [78], and ASSET [24], most of these models only apply to a specific environment. Moreover, complete execution criteria are not discussed in these models. In order to make the proposed transaction processing mechanism more applicable and suitable for various applications, we need a unified transaction model. The first step in this research is therefore to describe the generalized advanced transactions. The proposed generalized advanced transaction is very expressive – it is capable of supporting various complex applications and represent many existing and yet to develop advanced transactions.

Several papers [30, 31] have discussed dependencies as effective means to construct advanced transactions. Our generalized advanced transaction will have subtransaction and dependencies as the key components. We next describe the generalized advanced transaction, and different types of dependencies that can exist in our models.

3.1 Generalized Advanced Transaction

In this section, we propose the concept of *generalized advanced transaction.* The generalized advanced transaction can be customized for different kinds of transaction models by restricting the type of dependencies that can exist between the component transactions. A generalized advanced transaction is specified by a set of subtransactions, the dependencies between these subtransactions, and the completion sets. All subtransactions specified in a generalized advanced transaction may not execute or commit. A completion set gives the set of subtransactions that needs to be committed for completing the generalized advanced transaction. Some of these subtransactions must be committed in a specific order – the completion set needs to specify this ordering relation. Moreover, the

set of subtransactions that commit in a generalized advanced transaction model may vary with different instances of the generalized advanced transaction. Thus, a generalized advanced transaction may have multiple completion sets.

3.1.1 Generalized Advanced Transactions

Definition 1

[Generalized Advanced Transaction]: A Generalized Advanced Transaction AT_w is a triple <S,D,C>, where,

S - is a set of low-level coordinated subtransactions. A subtransaction, represented by T_{wt}, contains a set of related operations and forms a smallest unit in coordination;

D - is the set of dependencies used to coordinate the execution of co-operating subtransactions. Dependencies in D specify how the subtransactions are related to each other.

C - is the set of completion sets in AT_w.

Definition 2

[Subtransaction] A *subtransaction* T_{wi} is the smallest logical unit of work in an advanced transaction. It consists of a set of data operations (read and write) and subtransaction primitives (begin, abort and commit). The read and write operations that subtransaction T_{wi} performs on data item x are denoted by $r_{wi}[x]$ and $w_{wi}[x]$ respectively. The begin, abort and commit primitives of subtransaction T_{wi} are denoted by b_{wi}, a_{wi} and c_{wi} respectively. The execution of these primitives is often referred to as an event. Thus, we can have begin, commit or abort events. The commit and abort events are referred to as termination events.

When a subtransaction T_{wi} is in the committed or aborted state, its state cannot be changed any more and we say that T_{wi} is in the final state.

Definition 3

[Completion Set] Consider the advanced transaction $AT_w =< S, D, C >$. The completion component C defines the set of completion sets in AT_w. Each *completion set* $C_t \in C$ is specified by $(CT_t, \ll_t)$, where CT_t is the set of subtransactions that must be committed and $\ll_t$ is the order in which they must be committed. Formally, $CT_t \subseteq S$ is the set of subtransactions which must be

committed for this advanced transaction to complete, and $\ll_t$ is the ordering relation that specifies the commit order of subtransactions in CT_i.

We refer an advanced transaction in the form of AT_w, and refer any subtransaction like T_{wi} or T_{wj}, implying that it is subtransaction instead of an advanced transaction.

A generalized advanced transaction AT_w can be represented in the form of a directed graph $G_w =< V, E >$. The subtransaction $T_{w1}, T_{w2}, \ldots, T_{wn}$ defined in S correspond to the different nodes of the graph. Each dependency between transactions T_{wi} and T_{wj} is indicated by a directed edge (T_{wi}, T_{wj}) that is labeled with the name of the dependency.

3.1.2 Dependencies

Next, we describe the dependencies in our advanced transaction model. In order to properly coordinate the different subtransactions in an advanced transaction, dependencies are specified on subtransaction primitives, subtransaction operations, and subtransaction input/outputs. Dependencies that exist between subtransactions of the same advanced transaction are called *internal dependencies*. Dependencies between subtransactions of different advanced transactions are called *external dependencies*.

The only kind of external dependency we consider is *read-write dependency*. Dependencies between subtransactions of the same advanced transaction can be read-write dependency as well as *control-flow dependency* and *data-flow dependency*. In this dissertation, I will focus on control-flow dependencies, since they have the biggest impact on coordinating the execution of advanced transactions. In the rest of this dissertation, I use the term 'dependency' to mean 'control-flow dependency'. Read-write dependencies and data-flow dependencies will be mentioned explicitly.

Definition 4

[Read-write Dependency] A subtransaction T_{ij} is *read-write dependent* upon a subtransaction T_{kl} if there exists a data item x such that:

1. T_{ij} reads x after T_{kl} has updated x,
2. T_{kl} does not abort before T_{ij} reads x, and,
3. if any T_{pq} updates x after T_{kl} has updated x but before T_{ij} reads it, then T_{pq} is aborted.

Read-write dependencies are also present in the traditional transaction processing model. Concurrency control mechanisms [20], such as two-phase locking and time-stamping, ensure correct execution in the presence of such dependencies.

Definition 5

[Data-flow Dependency] A subtransaction T_{ij} is *data-flow dependent* upon a subtransaction T_{ik} in an advanced transaction AT_i, if there exists a data item x such that:

1. subtransactions T_{ij} and T_{ik} belong to the same advanced transaction AT_i
2. x is an input data item for T_{ik}
3. x is an output data item of T_{ij}
4. T_{ik} accepts x as an input from T_{ij}

Note that if there is a data-flow dependency between subtransactions T_{ik} and T_{ij}, there will also be a control-flow dependency of the form *begin-on-commit* between the two subtransactions. This is because for the output of T_{ik} to be available to T_{ij}, T_{ik} must commit before T_{ij} starts execution. Thus, we do not consider data-flow dependencies separately in this dissertation.

Definition 6

[Control-flow Dependency] A *control-flow dependency* specified between a pair of subtransactions T_{ij} and T_{ik} expresses how the execution of a primitive (begin, commit, and abort) of T_{ij} causes (or relates to) the execution of the primitives (begin, commit and abort) of another subtransaction T_{ik}.

Control-flow dependencies are also referred as *transactional dependencies* in many papers. A set of control-flow dependencies has been defined in the work of ACTA [28]. A comprehensive list of transaction dependency definitions can be found in [10, 24, 28]. Summarizing all these dependencies in previous work, we collect a total of fifteen different types of dependencies. These are given below. In the following descriptions T_{ij} and T_{ik} refer to the subtransactions and b_{ij}, c_{ij}, a_{ij} refer to the begin, commit, abort event of T_{ij} respectively that are present in some history H, and the notation $e_{ij} \prec e_{ik}$ denotes that event e_{ij} precedes event e_{ik} in the history H.

[Commit dependency] ($T_{ij} \rightarrow_c T_{ik}$): If both T_{ij} and T_{ik} commit then the commitment of T_{ij} precedes the commitment of T_{ik}. Formally, $c_{ij} \Rightarrow (c_{ik} \Rightarrow (c_{ij} \prec c_{ik}))$.

[Strong commit dependency] ($T_{ij} \rightarrow_{sc} T_{ik}$): If T_{ij} commits then T_{ik} also commits. Formally, $c_{ij} \Rightarrow c_{ik}$.

[Abort dependency] ($T_{ij} \rightarrow_{a} T_{ik}$): If T_{ij} aborts then T_{ik} aborts. Formally, $a_{ij} \Rightarrow a_{ik}$.

[Weak abort dependency] ($T_{ij} \rightarrow_{wa} T_{ik}$): If T_{ij} aborts and T_{ik} has not been committed then T_{ik} aborts. Formally, $a_{ij} \Rightarrow (\neg (c_{ik} \prec a_{ij} \Rightarrow a_{ik})$

[Termination dependency] ($T_{ij} \rightarrow_{t} T_{ik}$): Subtransaction T_{ik} cannot commit or abort until T_{ij} either commits or aborts. Formally, $e_{ik} \Rightarrow e_{ij} \prec e_{ik}$, where $e_{ij} \in \{c_{ij}, a_{ij}\}$, $e_{ik} \in \{c_{ik}, a_{ik}\}$.

[Exclusion dependency] ($T_{ij} \rightarrow_{ex} T_{ik}$): If T_{ij} commits and T_{ik} has begun executing, then T_{ik} aborts. Formally, $(c_{ij} \Rightarrow (b_{ik} \Rightarrow a_{ik})$.

[Force-commit-on-abort dependency] ($T_{ij} \rightarrow_{fca} T_{ik}$): If T_{ij} aborts, T_{ik} commits. Formally, $a_{ij} \Rightarrow c_{ik}$.

[Force-begin-on-commit/abort/begin/termination dependency] ($T_{ij} \rightarrow_{fbc/fba/fbb/fbt} T_{ik}$): Subtransaction T_{ik} must begin if T_{ij} commits(aborts/begins/terminates). Formally, $c_{ij}(a_{ij}/b_{ij}/T_{ij}) \Rightarrow b_{ik}$.

[Begin dependency] ($T_{ij} \rightarrow_{b} T_{ik}$): Subtransaction T_{ik} cannot begin execution until T_{ij} has begun. Formally, $b_{ik} \Rightarrow (b_{ij} \prec b_{ik})$.

[Serial dependency] ($T_{ij} \rightarrow_{s} T_{ik}$): Subtransaction T_{ik} cannot begin execution until T_{ij} either commits or aborts. Formally, $b_{ik} \Rightarrow (e_{ij} \prec b_{ik})$ where $e_{ij} \in \{c_{ij}, a_{ij}\}$.

[Begin-on-commit dependency] ($T_{ij} \rightarrow_{bc} T_{ik}$): Subtransaction T_{ik} cannot begin until T_{ij} commits. Formally, $b_{ik} \Rightarrow (c_{ij} \prec b_{ik})$.

[Begin-on-abort dependency] ($T_{ij} \rightarrow_{ba} T_{ik}$): Subtransaction T_{ik} cannot begin until T_{ij} aborts. Formally, $b_{ik} \Rightarrow (a_{ij} \prec b_{ik})$.

Sometimes one single dependency is not adequate for specifying the relationship between two subtransactions. For example, if we want to specify that (i) T_{11} must begin after T_{12} has committed and (ii) if T_{12} aborts then T_{11} must also abort. In such cases, a single dependency is not sufficient for

expressing the co-ordination relationship between T_{11} and T_{12}. A *composite dependency* is needed under this situation. A composite dependency contains two or more primitive dependencies which are applied towards the same pair of subtransactions. The single dependencies will be henceforth referred to as *primitive dependencies*. For example, the above two primitive dependencies could generate a composite dependency: $T_{12} \rightarrow_{fbc,a} T_{11}$.

Definition 7

[Composite Dependency] A *composite dependency* between a pair of subtransactions T_{ij}, T_{ik} in an advanced transaction, denoted by $T_{ij} \rightarrow_{d_1,d_2,\ldots,d_n} T_{ik}$, is obtained by combining two or more primitive dependencies $d_1, d_2, \ldots, d_n$. The effect of the composite dependency is the conjunction of the constraints imposed by the individual dependencies $d_1, d_2, \ldots, d_n$.

Note that, the constraints placed by the individual primitive dependencies might conflict with each other. This issue will be addressed later.

3.1.3 Representing Different Types of Advanced Transactions

The generalized advanced transaction can be specialized by restricting the number and types of dependencies that can exist between its transactions. For instance a SAGA [42] consisting of three subtransactions T_1, T_2, T_3 and compensating transactions C_2 and C_1 can be expressed as a generalized advanced transaction AT where $S = \{T_1, T_2, T_3, C_1, C_2\}$, $D = \{T_2 \rightarrow_{sc} T_1, T_3 \rightarrow_{sc} T_2, C_2 \rightarrow_{sc} T_2, C_1 \rightarrow_{sc} T_1, C_2 \rightarrow_{sc} C_1\}$, and $C = \{(\{T_1, T_2, T_3\}, \{\}), (\{T_1, T_2, C_2, C_1\}, \{T_1 \ll C_1, T_2 \ll C_2\}), (\{T_1, C_1\}, \{T_1 \ll C_1\})\}$. Other advanced transactions can also be expressed in the form of a generalized transaction using different kinds of dependencies.

The following example shows how a real-world application that can be represented by the generalized advanced transaction model – a business workflow making travel arrangements.

Example 2

The generalized advanced transaction model can be used to express complex real-world applications. An example can be a workflow associated with making travel arrangements. This application contains a number of component activities that are highly cooperating with each other. This business workflow could work in any enterprise to make travel arrangement for the employees. The subtransactions perform the following tasks.

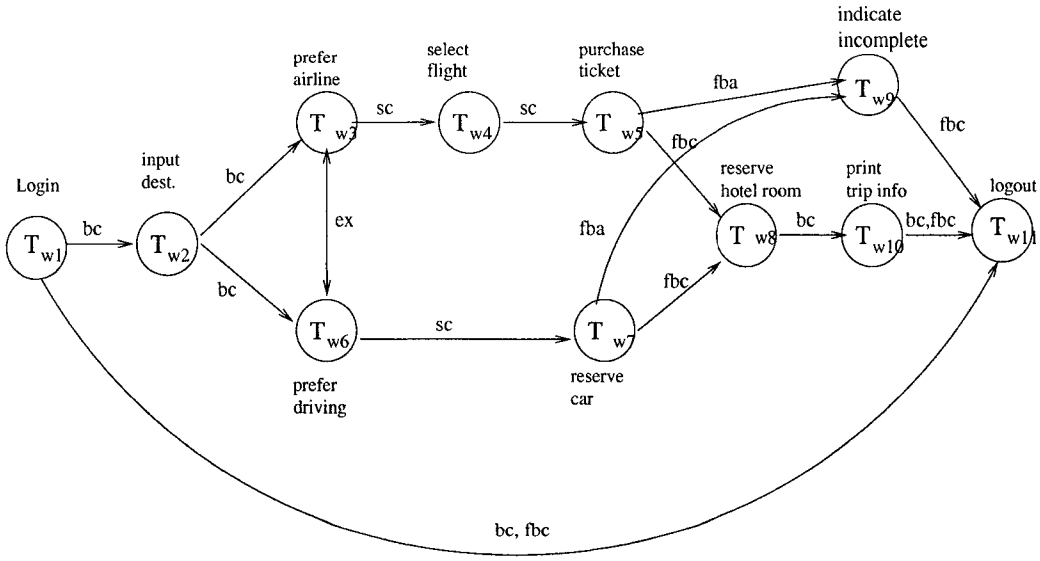

Figure 2: A Real World Example of Generalized Advanced Transaction

(i) Subtransaction T_{w1} – login into the travel arrangement application,

(ii) Subtransaction T_{w2} – input the travel information, including name, source and destination cities,

(iii) Subtransaction T_{w3} – user can choose either flight or ground travel. If the user prefers flight, he need to input information regarding flight,

(iv) Subtransaction T_{w4} – show all possible flights for user to select,

(v) Subtransaction T_{w5} – purchase the air ticket,

(vi) Subtransaction T_{w6} – if the user prefers driving to destination city, he need to input information regard rental car,

(vii) Subtransaction T_{w7} – reserve a rental car,

(viii) Subtransaction T_{w8} – make a reservation of a hotel room,

(ix) Subtransaction T_{w9} – if the flight or rental car reservation failed, indicate an incomplete transaction,

(x) Subtransaction T_{w10} – all reservations are made successfully, print out the travel reservation plan to the user,

(xi) Subtransaction T_{w11} – log out of the travel arrangement application.

There is a *begin-on-commit* dependency between T_{w1} and T_{w2}. This means the travel arrangement cannot start until the user has authenticated to the system successfully. There are also *begin-on-commit* dependencies between T_{w2} and T_{w3}, and T_{w2} and T_{w6} as well. This requires that the user must input the travel source and destination information before he can choose a plan. There are *strong commit* dependencies between T_{w3}, T_{w4} and T_{w5}, and also between T_{w6} and T_{w7} as well. This ensures that the air ticket or rental car reservation is made for the user. There are *force-begin-on-abort* dependencies between T_{w5} and T_{w9}, and between T_{w7} and T_{w9} as well. This means that T_{w9} must start after T_{w5} or T_{w7} has aborted. It indicates that the airlines ticket cannot be purchased. The *force-begin-on-commit* dependency between T_{w5} and T_{w8} ensures that T_{w8} must start if the ticket has been purchased successfully. The *exclusion* dependency between T_{w3} and T_{w6} ensures that either T_{w3} can commit or T_{w6} can commit but not both. In other words, either the airlines ticket is purchased or the rental car reservation is made, but not both. The *begin-commit* dependency between T_{w8} and T_{w10} ensures that the travel plan to be printed out correctly if all the reservations are successfully made. Finally, there are *begin-on-commit* and *force-begin-on-commit* dependencies between T_{w1} and T_{11}, and T_{w10} and T_{w11} as well. This means that the log out process must complete if the workflow terminates.

We can use a generalized advanced transaction $AT_w =< S, D, C >$ to express this application, the dependency graph is shown in Figure 2, where $S = \{T_{w1}, T_{w2}, T_{w3}, \ldots, T_{w11}\}$, $D = \{T_{w1} \rightarrow_{bc} T_{w2}, T_{w2} \rightarrow_{bc} T_{w3}, T_{w2} \rightarrow_{bc} T_{w6}, T_{w3} \rightarrow_{ex} T_{w6}, T_{w3} \rightarrow_{sc} T_{w4}, \ldots, T_{w10} \rightarrow_{bc} T_{w11}, T_{w10} \rightarrow_{fbc} T_{w11}\}$, and $C = \{(\{T_{w1}, T_{w2}, T_{w3}, T_{w4}, T_{w5}, T_{w8}, T_{w10}, T_{w11}\}, \{T_{w1} \ll T_{w2}, T_{w2} \ll T_{w3}\}),$
$(\{T_{w1}, T_{w2}, T_{w6}, T_{w7}, T_{w8}, T_{w10}, T_{w11}\}, \{T_{w1} \ll T_{w2}, T_{w2} \ll T_{w3}\}),$
$(\{T_{w1}, T_{w2}, T_{w9}, T_{w11}\}, \{T_{w1} \ll T_{w2}, T_{w2} \ll T_{w3}\})\}$.

The labels on each edge corresponds to the dependencies that exist between the subtransactions. $L(T_{w1}, T_{w2}) = \{bc\}$, $L(T_{w2}, T_{w3}) = \{bc\}$, $L(T_{w3}, T_{w6}) = \{ex\}, \ldots, L(T_{w10}, T_{w11}) = \{bc\}$. This transaction has three completion sets, they identify the following states respectively – (i) reserved

air tickets successfully, the travel plan is made, (ii) reserved rental car successfully, the travel plan is made, (iii) reservation is failed, no travel plan is made. This generalized transaction can be represented graphically as shown in Figure 2.

As we have shown above, the generalized advanced transaction can be used to represent SAGA transactions, nested transactions, and workflow applications as well, by specifying different dependency structures among the cooperating subtransactions. We can also combine the primitive dependencies to impose stronger constraints and also specify complete execution criteria with the generalized advanced transaction model. Most existing work in extended transactions [8, 16, 24, 78, 80, 93] do not provide such expressive and powerful framework to model and analyze advanced transactions.

The generalized advanced transaction can be specialized by restricting the number and types of dependencies that can exist between its transactions. In this work, however, we do not restrict the dependencies that are possible between transactions of a generalized advanced transaction. Rather our goal is to illustrate which combinations of dependencies cause problems and which do not.

CHAPTER IV

ANALYSIS AND PROPERTIES OF DEPENDENCIES

Control-flow dependencies play an important role in co-ordinating the executions of the subtransactions. Although a few research work discussed how to use dependencies to build advanced transactions and workflow systems [10, 13, 38, 44, 60, 62], not much work has been done on the analysis of dependencies. Incorrect specification of dependencies can result in unpredictable behavior, which in turn, can lead to unavailability of resources and information integrity problems.

In this chapter, we focus on how to correctly specify dependencies in an advanced transaction. We will give the *classification* of the control-flow dependencies, classify them into two broad categories: event ordering and event enforcement dependencies. Different event ordering and event enforcement dependencies in an advanced transaction often interact in subtle ways resulting in conflicts and redundancies. We describe the different types of *dependency conflicts* that can arise due to the presence of multiple dependencies and describe how one can detect such conflicts. An advanced transaction may also contain *redundant dependencies* – these are dependencies that can be logically derived from other dependencies. We show how such extraneous dependencies can be eliminated to get an equivalent set of dependencies that has the same effect as the original set. Our dependency analysis is done in the context of the generalized advanced transaction proposed in previous chapter.

The correct specification of advanced transaction requires that the dependencies are conflict-free and are not redundant; moreover, the completion set must conform to the given set of dependencies. In this work, the correctness of dependency specification is formalized in terms of two properties: *conflict-free property* and *minimality property*. The conflict-free property ensures that a given set of dependencies is physically realizable. That is, the dependencies do not pose constraints on the execution that contradict each other and are therefore impossible to satisfy in a given execution. If a given set of dependencies have conflicts, we cannot give any assurance about the correctness of the behavior of the generalized advanced transaction. Thus, we must ensure that the dependencies do not conflict each other. We show how dependency conflicts occur and thereby help make the

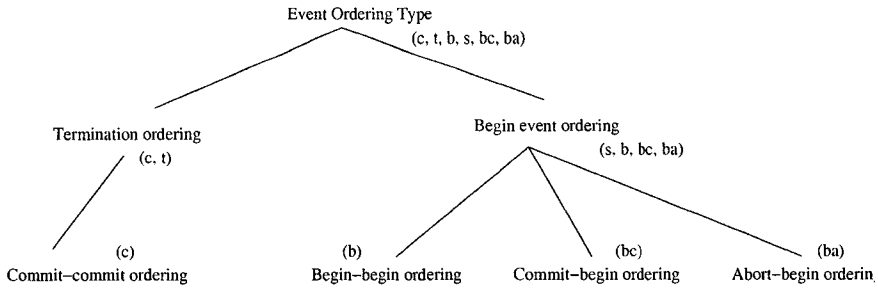

Figure 3: Event Ordering Type of Dependencies

dependency specification conflict-free.

Enforcement of dependencies in advanced transaction models incurs additional overhead. This is because the system must form additional checks and carry out operations to ensure the satisfaction of the dependencies. Thus, the presence of extraneous dependencies in a set of dependencies unnecessarily slows down performance. To address this issue, we propose the minimality property. The minimality property ensures that no dependency specified in a set of dependencies is extraneous. We show how to check for extraneous dependencies and how to minimize a given set of dependencies.

We next discuss different kinds of conflicts that can occur in a set of dependencies and illustrate how to detect such conflicts. Finally, we provide algorithms that check for errors in dependency specification and algorithms for composing dependencies. This work will help the application developer in getting assurance about the correctness of the dependency specification and thereby ensure the correct behavior of the advanced transaction.

4.1 Dependency Classification

One class of dependencies specify ordering relationships among the events of two subtransactions; we call such dependencies *event ordering dependencies.* The other class of dependencies require that some event must happen when certain event has occurred in the other subtransaction; we call such dependencies *event enforcement dependencies.* We also show that in practical systems how certain event enforcement dependencies imply an execution ordering.

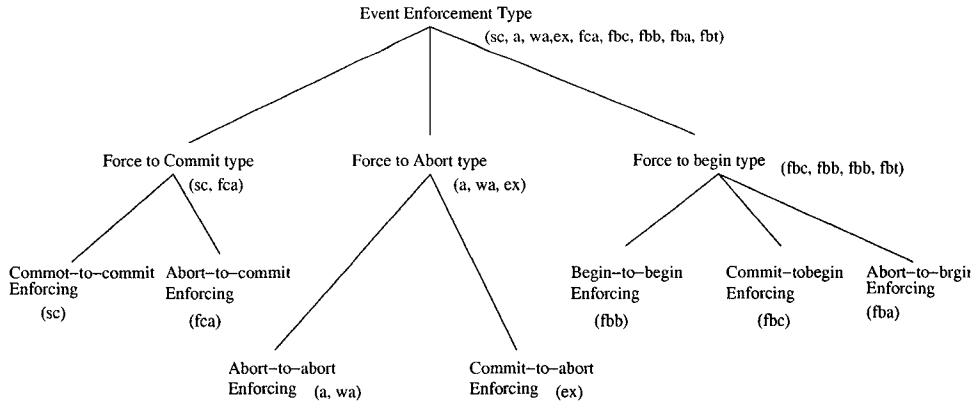

Figure 4: Event Enforcement Type of Dependencies

Definition 8

[Event Ordering Dependency] An *event ordering dependency* $T_{wi} \rightarrow_d T_{wj}$ is one in which the execution of some event in subtransaction T_{wi} must precede the execution of some event in subtransaction T_{wj}.

The commit, termination, begin, serial, begin-on-commit and begin-on-abort dependencies are event-ordering dependencies. The event ordering dependencies can be classified into ones that order the begin events and others that order the termination event. Detailed classification of the event ordering dependencies is given in Figure 3.

Definition 9

[Event Enforcement Dependency] An *event enforcement dependency* $T_{wi} \rightarrow_d T_{wj}$ is one in which the execution of some event (begin, commit or abort) in subtransaction T_{wi} requires the execution of some event in subtransaction T_{wj}.

The strong commit, abort, weak abort, exclusion, force-commit-on-abort, force-begin-on-commit, force-begin-on-abort, force-begin-on-begin, and force-begin-on-terminate are examples of such dependencies. The event enforcement dependencies can be further classified into enforcing commit events, enforcing abort events and enforcing begin events. These again can be classified as commit-to-commit enforcing and abort-to-commit enforcing etc. Detailed classification of the event enforcement type of dependencies are given in Figure 4.

In the preceding discussion we have classified the dependencies into event enforcement and event ordering types. One point needs to be mentioned. Sometimes event enforcement dependencies impose an execution order on transactions. For instance, consider the strong commit dependency $T_{wi} \rightarrow_{sc} T_{wj}$. This dependency requires T_{wj} to commit if T_{wi} commits. It does not specify any order of execution. So if T_{wi} commits and subsequently T_{wj} aborts, the dependency will be violated. Since the commitment of a transaction cannot be guaranteed, this possibility exists. Thus, in order to ensure the satisfaction of the dependency, we require T_{wj} to commit before T_{wi}. Moreover, if T_{wj} aborts for any reason, then to maintain the dependency T_{wi} should not be allowed to commit. In other words, to satisfy this dependency we need to enforce an execution order. Thus, whenever there is a dependency of the form $T_{wi} \rightarrow_{sc} T_{wj}$, there are implicit dependencies of the form $T_{wj} \rightarrow_{c} T_{wi}$ and $T_{wj} \rightarrow_{a} T_{wi}$. Thus, to ensure the satisfaction of the strong commit dependency, an execution order is also imposed.

Similar problems exist with the abort and force-commit-on-abort dependencies as well. The event enforcement dependency $T_{wi} \rightarrow_{a} T_{wj}$, requires T_{wj} to abort if T_{wi} aborts. If an execution order is not specified T_{wj} may commit before T_{wi} aborts, and the dependency will not be satisfied. Here also, we assume there is an implicit commit dependency of the form $T_{wi} \rightarrow_{c} T_{wj}$. We can make similar arguments and show that for the event enforcement dependency $T_{wi} \rightarrow_{fca} T_{wj}$, we need an implicit dependency of the form $T_{wj} \rightarrow_{bc} T_{wi}$.

Conversely, sometimes event ordering dependencies require that some event must occur or some event must not occur. In other words, they may imply the presence or absence of some event enforcement dependencies. An example will help illustrate this point. Consider, for instance, the begin-on-abort dependency $T_{wi} \rightarrow_{ba} T_{wj}$. This requires that T_{wj} cannot begin until T_{wi} aborts. In other words, if T_{wi} commits, T_{wj} should not begin. This is not unexpected because $c_{wi} \Rightarrow \neg b_{wj}$ can be derived from $b_{wj} \Rightarrow (a_{wi} \prec b_{wj})$. Similarly, the event ordering dependency $T_{wi} \rightarrow_{bc} T_{wj}$ requires T_{wi} to commit before T_{wj} can begin. In other words $a_{wi} \Rightarrow \neg b_{wj}$. The event ordering dependency $T_{wi} \rightarrow_{t} T_{wj}$ does not allow T_{wj} to terminate unless T_{wi} has already done so. In other words $\neg e_{wi} \Rightarrow \neg e_{wj}$ where e_{wi}, e_{wj} denote the termination event of T_{wi} and T_{wj} respectively.

4.2 *Composite Dependencies*

Our formal specification of dependencies allows us to combine two or more dependencies between a pair of transactions in the generalized advanced transaction. With the model, one can specify multiple kinds of dependencies between any pair of transaction in a generalized advanced transaction. Such dependencies are called *composite dependencies*. Each individual dependency defined over a pair of subtransaction is referred to as a *primitive dependency*. Composite dependencies are often needed to express more powerful coordination controls over transactions.

To formalize, what dependencies can be combined in a pair of transactions we introduce the notion of *inclusion relation* and *conflict relation* that can exist between dependencies specified on a pair of transactions. A dependency d_x is said to include another dependency d_y if the constraints imposed by d_y are logically implied by the constraints imposed by d_x. A dependency d_x is said to conflict with another dependency d_y if the constraints imposed by d_x violate the constraint imposed by d_y. Dependencies between a pair of transactions can be combined if they do not conflict with each other, that is, they do not impose constraints on the transaction execution that contradict each other. Moreover, if two dependencies related by inclusion relationship is combined, the dependency that is included by the other dependency is eliminated from the combined dependency.

Definition 10

[Composite Dependency] A composite dependency between a pair of transactions T_{wi}, T_{wj} in a generalized advanced transaction model, denoted by $T_{wi} \rightarrow_{d_1,d_2,...,d_n} T_{wj}$, is obtained by combining two or more primitive dependencies $d_1, d_2, \ldots, d_n$. The effect of the composite dependency is the conjunction of the constraints imposed by the primitive dependencies $d_1, d_2, \ldots, d_n$.

When multiple dependencies are specified between a pair of transactions, we need to ensure that no dependency is redundant and the dependencies do not impose conflicting requirements. We impose these requirements as necessary properties of composite dependencies.

4.2.1 Minimality Property

Before describing the necessary properties of composite dependencies, we define two kinds of of dependency relationships that may exist between a pair of dependencies. These are *inclusion* and *conflict*.

Definition 11

[Inclusion Relation] Let $T_{wi} \rightarrow_{d_x} T_{wj}$ and $T_{wi} \rightarrow_{d_y} T_{wj}$ be a pair of dependencies defined over T_{wi} and T_{wj}. The dependency d_x is said to include dependency d_y, denoted by $d_y \subseteq d_x$ if the satisfaction of dependency d_x implies the satisfaction of dependency d_y. The dependency d_y in this case is referred to as included dependency and d_x is the including dependency. The inclusion relationship is reflexive, transitive and anti-symmetric.

For instance, the abort dependency includes the weak abort dependency. This is because whenever the abort dependency is satisfied, the weak abort dependency will also be satisfied. This is formalized by the following theorem.

Theorem 1

The abort dependency $T_{wi} \rightarrow_a T_{wj}$ includes the weak abort dependency $T_{wi} \rightarrow_{wa} T_{wj}$, that is, $(T_{wi} \rightarrow_{wa} T_{wj}) \subseteq (T_{wi} \rightarrow_a T_{wj})$.

Proof 1

To prove this, we must show that the constraints imposed by the abort dependency $(a_{wi} \Rightarrow a_{wj})$, implies the constraints imposed by the weak abort dependency $(a_{wi} \Rightarrow (\neg(c_{wj} \prec a_{wi}) \Rightarrow a_{wj})$. In other words, we need to show that

$$(a_{wi} \Rightarrow a_{wj}) \Rightarrow (a_{wi} \Rightarrow (\neg(c_{wj} \prec a_{wi}) \Rightarrow a_{wj})) = True$$

The left hand side on simplification evaluates to true which equals the right hand side.

Similarly, we can show that the force-begin-on-begin dependency includes force-begin-on-commit, force-begin-on-abort, and force-begin-on-terminate. Similarly, the serial dependency includes the begin dependency and the termination dependency. A complete list of inclusion relations is given in Table 1.

The next relationship that we define is the generalization/specialization relationships between the dependencies.

Definition 12

[Generalization Relations] A dependency $T_{wi} \rightarrow_x T_{wj}$ is a generalization of the dependency $T_{wi} \rightarrow_y T_{wj}$ if the event on T_{wi} required by dependency $T_{wi} \rightarrow_x T_{wj}$ is a generalization of the event on T_{wi} required by the dependency $T_{wi} \rightarrow T_{wj}$.

Dependency	Includes
$T_{wi} \rightarrow_a T_{wj}$	wa
$T_{wi} \rightarrow_{fbb} T_{wj}$	fbc, fba, fbt
$T_{wi} \rightarrow_{fbt} T_{wj}$	fba, fbc
$T_{wi} \rightarrow_t T_{wj}$	c
$T_{wi} \rightarrow_s T_{wj}$	b, t
$T_{wi} \rightarrow_{bc} T_{wj}$	t, b, s, wbc, c
$T_{wi} \longrightarrow_{ba} T_{wj}$	t, b, s

Table 1: Dependencies Inclusion Relationship

Consider, for instance, the dependency $T_{wi} \rightarrow_t T_{wj}$ and the dependency $T_{wi} \rightarrow_c T_{wj}$. The dependency $T_{wi} \rightarrow_t T_{wj}$ is a generalization of $T_{wi} \rightarrow_c T_{wj}$.

Definition 13

[Minimality Property] A composite dependency $T_{wi} \rightarrow_{d_1,d_2,...d_y} T_{wj}$ is said to have the *minimality property* if there are no two dependencies d_m and d_n in $\{d_1, \ldots, d_y\}$ such that some dependency d_m includes dependency d_n.

The minimality property ensures that no unnecessary dependencies are present in a composite dependency.

4.2.2 Conflict-Free Property

Definition 14

[Conflict] Two dependencies d_x and d_y are said to conflict if the constraints imposed by the dependency d_x makes it impossible to satisfy the constraints of d_y.

For instance, consider the dependencies $T_{wi} \rightarrow_{sc} T_{wj}$ and $T_{wi} \rightarrow_{ex} T_{wj}$. The strong commit dependency requires that if T_{wi} commits, then T_{wj} should commit. The exclusion dependency says that if T_{wi} commits, then T_{wj} should abort. Thus, when T_{wi} commits, it is impossible to satisfy both the dependencies. Such conflicts can be found out by using a conjunction on the conditions imposed by the dependencies and evaluating the conjunct to find whether it will be satisfied for all possible executions. A second example of conflict is caused by the dependencies: $T_{wi} \rightarrow_a T_{wj}$ and $T_{wi} \rightarrow_{fca} T_{wj}$. Below, we give a formal proof of why these two dependencies conflict.

Theorem 2

The dependencies $T_{wi} \rightarrow_a T_{wj}$ and $T_{wi} \rightarrow_{fca} T_{wj}$ conflict.

Dependency	Conflicting dependency
$T_{wi} \longrightarrow_{sc} T_{wj}$	ex, bc, s, c, a, t, wbc, ba
$T_{wi} \longrightarrow_{a} T_{wj}$	fca, sc
$T_{wi} \longrightarrow_{wa} T_{wj}$	fca
$T_{wi} \longrightarrow_{ex} T_{wj}$	sc
$T_{wi} \longrightarrow_{fca} T_{wj}$	a, b, ba, bc, wa, s, wbc
$T_{wi} \longrightarrow_{fbc} T_{wj}$	ba
$T_{wi} \longrightarrow_{fbb} T_{wj}$	ba, bc
$T_{wi} \longrightarrow_{fba} T_{wj}$	bc
$T_{wi} \longrightarrow_{fbt} T_{wj}$	ba, bc
$T_{wi} \longrightarrow_{c} T_{wj}$	sc
$T_{wi} \longrightarrow_{t} T_{wj}$	sc
$T_{wi} \longrightarrow_{b} T_{wj}$	fca
$T_{wi} \longrightarrow_{s} T_{wj}$	sc, fca
$T_{wi} \longrightarrow_{bc} T_{wj}$	$sc, fca, ba, fbt, fbb, fba$
$T_{wi} \longrightarrow_{ba} T_{wj}$	$bc, fca, fbt, fbb, fbc, sc$
$T_{wi} \longrightarrow_{wbc} T_{wj}$	sc, fca

Table 2: Conflicting Dependencies over Same Pair of Subtransactions

Proof 2

To prove this we must show that the constraints imposed by the abort dependency ($a_{wi} \Rightarrow a_{wj}$) and the force-commit-on-abort dependency ($a_{wi} \Rightarrow c_{wj}$) cannot be satisfied for all possible executions. The conjunction of the constraints imposed by the two dependencies evaluate to $\neg a_{wi}$. This constraint is only satisfied when T_{wi} is not aborted. Thus if T_{wi} aborts, the conjunction of the constraints imposed by the dependencies will not be satisfied.

Conflicts for the other dependency pairs listed in Table 2 can be proved in a similar manner.

Definition 15

[Conflict-free Property] A composite dependency $T_{wi} \rightarrow_{d_1, d_2, \ldots d_y} T_{wj}$ is said to have the *conflict-free property* if there are no two dependencies d_m and d_n in $\{d_1, \ldots, d_y\}$ such that dependency d_m conflicts with dependency d_n.

4.3 Dependency Minimization

Dependencies specified in a generalized advanced transaction may not be independent. A given set of dependencies may logically imply the existence of other dependencies. For instance, a dependency d_x may include another dependency d_y. In such a case, even if d_y is not in the given set, it is

logically implied. A second example is when dependencies have the *transitive property*. A dependency of type x is said to be *transitive* if $T_{wi} \rightarrow_x T_{wj}$ and $T_{wj} \rightarrow_x T_{wk}$ implies that $T_{wi} \rightarrow_x T_{wk}$. The dependencies having this property are strong commit, abort, termination, force-begin-on-begin, force-begin-on-terminate, begin, serial, begin-on-commit and begin-on-abort.

Implicit dependencies can also exist due to the interaction of a number of different dependencies. Dependencies specified between different pair of subtransactions may interact with each and they may imply other dependencies. For instance, consider the dependencies $T_{wi} \rightarrow_a T_{wj}$ and $T_{wj} \rightarrow_{bc} T_{wk}$. Although there are no explicit dependencies between the transactions T_{wi} and T_{wk}, the interaction of the dependencies $T_{wi} \rightarrow_a T_{wj}$ and $T_{wj} \rightarrow_{bc} T_{wk}$ will put some constraints over the execution of T_{wi} and T_{wk}. $T_{wi} \rightarrow_a T_{wj}$ require that T_{wj} must abort if T_{wi} aborts. $T_{wj} \rightarrow_{bc} T_{wk}$ require that T_{wk} cannot begin until T_{wj} commits. Combining these two constraints, we obtain that T_{wk} cannot begin until T_{wi} commits, that is, $T_{wi} \rightarrow_{bc} T_{wk}$. Similarly, the dependencies $T_{wi} \rightarrow_{fca} T_{wj}$ and $T_{wj} \rightarrow_{fbc} T_{wk}$ imply the existence of $T_{wi} \rightarrow_{fba} T_{wk}$. Also, $T_{wi} \rightarrow_{bc} T_{wj}$ and $T_{wi} \rightarrow_{ex} T_{wk}$ logically imply $T_{wj} \rightarrow_{ex} T_{wk}$. Many more such examples exist. This motivates us to propose the definition of closure of a dependency set. The definitions are in same vein as defined for functional dependencies [39].

Definition 16

[Closure of a Dependency Set] Let **D** be a set of dependencies. The closure of **D**, denoted by $\mathbf{D}^+$, is the set of dependencies that are logically implied by the dependencies in **D**.

Definition 17

[Cover] If every dependency logically implied by the set $\mathbf{D_1}$ is also implied by $\mathbf{D_2}$, then $\mathbf{D_2}$ is said to be a *cover* for $\mathbf{D_1}$. In other words, $\mathbf{D_2}$ is a cover for $\mathbf{D_1}$, if $\mathbf{D_1}^+ \subseteq \mathbf{D_2}^+$.

Thus, the specification of a generalized advanced transaction may contain extraneous dependencies. The presence of extraneous dependency slows down the processing of the transaction because dependencies need to be checked for ensuring correct behavior. In this section, we define how to minimize the given set of dependencies and obtain an equivalent minimal set.

Definition 18

[Redundant Dependency] A dependency $T_{wi} \rightarrow_x T_{wj}$ is said to be *redundant* in a set of dependencies **D**, if $(T_{wi} \rightarrow_x T_{wj}) \in (\mathbf{D} - \{T_i \rightarrow_x T_{wj}\})^+$.

Informally speaking, a dependency $T_{wi} \rightarrow_x T_{wj}$ is redundant in the set **D** if the constraints imposed by $T_{wi} \rightarrow_x T_{wj}$ can be derived from the constraints of the dependencies in the set **D** – $\{T_i \rightarrow_x T_{wj}\}$. A dependency set not having any redundant dependencies is said to be minimal. A minimal dependency set is formally defined below.

Definition 19

[Minimal] A set of dependencies **M** is said to be *minimal* if it satisfies the following condition:

1. for each $(T_{wi} \rightarrow_x T_{wj}) \in M$, the dependency x is a primitive dependency.

2. for each $(T_{wi} \rightarrow_x T_{wj}) \in M$, $(M - \{T_i \rightarrow_x T_{wj}\})^+ \neq M^+$.

Algorithm 1

Finding a Minimal Dependency Set

Input: D – Dependency sets that must be minimized

Output: M – Minimal dependency set

begin

M = D

for each $(T_{wi} \rightarrow_x T_{wj}) \in$ **M**

if $x = x_1, x_2, \ldots, x_n$ is a composite dependency

M = **M** – $\{T_{wi} \rightarrow_x T_{wj}\}$

for $m = 1$ to n **do**

M = **M** $\cup \{T_{wi} \rightarrow_{x_m} T_{wj}\}$

for each $(T_{wi} \rightarrow_y T_{wj}) \in$ **M**

if $(T_{wi} \rightarrow_y T_{wj}) \in$ (**M** – $\{T_{wi} \rightarrow_y T_{wj}\})^+$

M = **M** – $\{T_{wi} \rightarrow_y T_{wj}\}$

end

The first step involves changing each composite dependency $T_{wi} \rightarrow_x T_{wj}$ into a set of primitive dependencies, $T_{wi} \rightarrow_{x_1} T_{wj}$, $T_{wi} \rightarrow_{x_2} T_{wj}$, ..., $T_{wi} \rightarrow_{x_n} T_{wj}$. The second step involves checking whether each dependency $T_{wi} \rightarrow_y T_{wj}$ is logically implied by the remaining dependencies in the set **M** – $\{T_i \rightarrow_y T_{wj}\}$. If so, $T_{wi} \rightarrow_y T_{wj}$ is redundant and it is taken out of the minimal set **M**. The process is repeated for all the dependencies.

Definition 20

[Minimality Property] A set of dependencies D is said to have the *minimality property* if no dependency in D is redundant.

Definition 21

[Equivalence of Dependency Sets] Two dependency sets $\mathbf{D}_1$ and $\mathbf{D}_2$ are equivalent if $\mathbf{D}_1 \subseteq \mathbf{D}_2{}^+$ and $\mathbf{D}_2 \subseteq \mathbf{D}_1{}^+$. In other words, the closure of the two sets are equal, that is, $\mathbf{D}_1^+ = \mathbf{D}_2^+$.

Theorem 3

The minimal set **M** of dependencies obtained by Algorithm 1 is equivalent to the original set **D**.

Proof 3

We need to prove (i) $\mathbf{M} \subseteq \mathbf{D}^+$ and (ii) $\mathbf{D} \subseteq \mathbf{M}^+$. Since $\mathbf{M} \subseteq \mathbf{D}$, (i) is true. Since each dependency in **D** is removed from **M** only if it can be logically implied by the other remaining dependencies in **M**, $\mathbf{D} \subseteq \mathbf{M}^+$.

Algorithm 2

Checking if a Dependency is in the Closure of a Dependency Set

Input: (i) $T_{wi} \rightarrow_x T_{wj}$ – Dependency (ii) **D** – Dependency Set

Output: Boolean – True if they are equivalent, false otherwise

begin

$\mathbf{T} = T_{wi}$

while $((T_{wm} \rightarrow_y T_{wn}) \in \mathbf{D}) \wedge (T_{wm} \in \mathbf{T}) \wedge (T_{wn} \notin \mathbf{T})$

if $d_x \subseteq d_y \wedge d_x$ is transitive

$\mathbf{T} = \mathbf{T} \cup \{T_{wn}\}$

if $T_{wj} \notin \mathbf{T}$

return false

else

return true

end

Algorithm 3

Checking the Equivalence of Dependency Sets

Input: $\mathbf{D}_1$ and $\mathbf{D}_2$ – Dependency sets that are to be checked for equivalence

Output: Boolean – True if they are equivalent, false otherwise

begin

for each $(T_{wi} \rightarrow_x T_{wj}) \in \mathbf{D_1}$

if $(T_{wi} \rightarrow_x T_{wj}) \notin \mathbf{D_2}^+$

return false

for each $(T_{wi} \rightarrow_x T_{wj}) \in \mathbf{D_2}$

if $(T_{wi} \rightarrow_x T_{wj}) \notin \mathbf{D_1}^+$

return false

return true

end

The algorithm checks whether each dependency in $\mathbf{D}_1$ can be logically implied by the dependencies in $\mathbf{D}_2$. If not, the algorithm returns false. Otherwise, it checks whether each dependency in $\mathbf{D}_2$ can be derived from the dependencies in $\mathbf{D}_1$. If not, it returns false. Otherwise the result is true.

4.4 Impact of Dependencies on Completion Sets

Recall that a generalized advanced transaction $T =< S, D, C >$ specifies the set of subtransactions S, the dependencies between these subtransactions D, and the set C that contains the completion sets of T. A completion set of a generalized advanced transaction specifies the subtransactions that need to be committed and the order in which they must be committed for successful execution of some instance of the generalized advanced transaction.

The completion sets specified by a user in a generalized advanced transaction may not conform to the given dependencies. We provide an algorithm that checks whether the completion set complies with the dependencies in an advanced transaction.

Algorithm 4

Check Whether Completion Set Conforms to Dependency

Input: Specification of generalized advanced transaction $T =< S, D, C >$

Output: Returns true if completion set conforms to the dependencies, false otherwise

begin

for each $C_m \in \mathbf{C}$ where $C_m = (CT_m, ll_m)$

if $(T_{wi} \rightarrow_{sc} T_{wj} \in \mathbf{D}) \wedge (T_{wi} \in CT_m) \wedge (T_{wj} \notin CT_m)$

return false

if $(T_{wi} \rightarrow_{bc} T_{wj} \in \mathbf{D}) \wedge (T_{wi} \notin CT_m) \wedge (T_{wj} \in CT_m)$

return false

if $(T_{wi} \rightarrow_{ex|ba} T_{wj} \in \mathbf{D}) \wedge (T_{wi} \in CT_m) \wedge (T_{wj} \in CT_m)$

return false

if $(T_{wi} \rightarrow_{c|t|s|bc} T_{wj} \in \mathbf{D}) \wedge (T_{wi} \in CT_m) \wedge (T_{wj} \in CT_m) \wedge (T_{wi} \not\ll_m T_{wj})$

return false

return true

end

The algorithm looks at each completion set to check whether it complies with the dependencies. Not all dependencies impact a completion set. The algorithm begins by checking whether the completion set complies with the strong commit dependency. The strong commit dependency $T_{wi} \rightarrow_{sc} T_{wj}$ requires that if T_{wi} commits, then T_{wj} must commit. Hence, if the completion set contains T_{wi} and not T_{wj}, then the algorithm reports false signifying that the completion set does not conform to the dependency. Next, we consider the begin-on-commit dependency $T_{wi} \rightarrow_{bc} T_{wj}$. In this case, the algorithm reports false if T_{wi} is not in the completion set but T_{wj} is present. The algorithm then checks for exclusion or begin-on-abort dependency between T_{wi} and T_{wj}. In such cases, if the completion set contains both T_{wi} and T_{wj}, then we have a problem because the completion set violates the dependency. Finally, if there is a commit, termination, serial, and begin-on-commit dependency between T_{wi} and T_{wj}, and the commit order required by these dependencies does not comply with that specified in the completion set, the algorithm returns false. Otherwise the algorithm returns true indicating that the completion set conforms with the dependencies.

4.5 Interaction of Control-flow Dependencies

In the previous section, we outlined two desirable properties of composite dependencies: conflict-free property and minimality property. Even if all the composite dependencies in a generalized

advanced transaction have these properties, the resulting generalized advanced transaction may not have these properties. This is because two or more dependencies specified may interact with each other, and they could imply some implicit dependencies. These implicit dependencies can also result in conflicts and the loss of the minimality properties for the advanced transactions.

Definition 22

[Implicit Dependency] An implicit dependency $T_{wi} \rightarrow_d T_{wj}$ is a dependency implied by other dependencies. The implicit dependency is not directly given by the advanced transaction developer, instead, the constraint of this implicit dependency is implied by some directly given dependencies.

For instance, consider the dependencies $T_{wi} \rightarrow_a T_{wj}$ and $T_{wj} \rightarrow_{bc} T_{wk}$. Although there are no explicit dependencies between the transactions T_{wi} and T_{wk}, the interaction of the dependencies $T_{wi} \rightarrow_a T_{wj}$ and $T_{wj} \rightarrow_{bc} T_{wk}$ will put some constraints over the execution of T_{wi} and T_{wk}. $T_{wi} \rightarrow_a T_{wj}$ require that T_{wj} must abort if T_{wi} aborts. $T_{wj} \rightarrow_{bc} T_{wk}$ require that T_{wk} cannot begin until T_{wj} commits. Combining these two constraints, we obtain that T_{wk} cannot begin until T_{wi} commits. Therefore, there is an implicit edge $T_{wi} \rightarrow_{bc} T_{wk}$ exist in these sets of subtransactions.

Theorem 4

The dependencies $T_{wi} \rightarrow_a T_{wj}$ and $T_{wj} \rightarrow_{bc} T_{wk}$ imply $T_{wi} \rightarrow_{bc} T_{wk}$.

Proof 4

$T_{wi} \rightarrow_a T_{wj}$ implies an dependency $T_{wi} \rightarrow_c T_{wj}$. The logic expression of the dependencies $T_{wi} \rightarrow_a T_{wj}$, $T_{wj} \rightarrow_{bc} T_{wk}$, and $T_{wi} \rightarrow_c T_{wj}$ are $a_{wi} \Rightarrow a_{wj}$, $b_{wk} \Rightarrow (c_{wj} \prec b_{wk})$ and $c_{wi} \Rightarrow (c_{wj} \Rightarrow (c_{wi} \prec c_{wj}))$ respectively. $b_{wk} \Rightarrow (c_{wi} \prec b_{wk})$ is logically implied. Therefore the dependency $T_{wi} \rightarrow_{bc} T_{wk}$ can be derived from $T_{wi} \rightarrow_a T_{wj}$ and $T_{wj} \rightarrow_{bc} T_{wk}$.

As shown in the example above, two dependencies that are connected by an intermediate node could imply an implicit dependency. However, it is not always the case, as another example, two connect dependencies $T_{wi} \rightarrow_{sc} T_{wj}$ and $T_{wj} \rightarrow_a T_{wk}$ do not impose any constraint over T_{wi} and T_{wk}. In this case, T_{wk} can either commit or abort given that T_{wi} commits. One may wonder whether and when there would exist some implicit relationship between subtransactions T_{wi} and T_{wk}.

$T_{wi} \rightarrow_? T_{wk}$	$T_{wj} \rightarrow_{sc} T_{wk}$	$T_{wj} \rightarrow_a T_{wk}$	$T_{wj} \rightarrow_{fca} T_{wk}$	$T_{wj} \rightarrow_{ex} T_{wk}$
$T_{wi} \rightarrow_{sc} T_{wj}$	$T_{wi} \rightarrow_{sc} T_{wk}$	-	-	$T_{wi} \rightarrow_{ex} T_{wk}$
$T_{wi} \rightarrow_a T_{wj}$	-	$T_{wi} \rightarrow_a T_{wk}$	$T_{wi} \rightarrow_{fca} T_{wk}$	-
$T_{wi} \rightarrow_{wa} T_{wj}$	-	-	-	-
$T_{wi} \rightarrow_{fca} T_{wj}$	$T_{wi} \rightarrow_{fca} T_{wk}$	-	-	$T_{wi} \rightarrow_a T_{wk}$
$T_{wi} \rightarrow_{ex} T_{wj}$	-	$T_{wi} \rightarrow_{ex} T_{wk}$	$T_{wi} \rightarrow_{sc} T_{wk}$	-

Table 3: Interaction between Event Enforcement Dependencies

$T_{wi} \rightarrow_? T_{wk}$	$T_{wj} \rightarrow_{fbb} T_{wk}$	$T_{wj} \rightarrow_{fbc} T_{wk}$	$T_{wj} \rightarrow_{fba} T_{wk}$	$T_{wj} \rightarrow_{fbt} T_{wk}$
$T_{wi} \rightarrow_{sc} T_{wj}$	$T_{wi} \rightarrow_{fbc} T_{wk}$	$T_{wi} \rightarrow_{fbc} T_{wk}$	-	$T_{wi} \rightarrow_{fbc} T_{wk}$
$T_{wi} \rightarrow_a T_{wj}$	$T_{wi} \rightarrow_{fba} T_{wk}$	-	$T_{wi} \rightarrow_{fba} T_{wk}$	$T_{wi} \rightarrow_{fba} T_{wk}$
$T_{wi} \rightarrow_{wa} T_{wj}$	-	-	-	-
$T_{wi} \rightarrow_{fca} T_{wj}$	$T_{wi} \rightarrow_{fba} T_{wk}$	$T_{wi} \rightarrow_{fba} T_{wk}$	-	$T_{wi} \rightarrow_{fba} T_{wk}$
$T_{wi} \rightarrow_{ex} T_{wj}$	$T_{wi} \rightarrow_{fbc} T_{wk}$	-	$T_{wi} \rightarrow_{fbc} T_{wk}$	$T_{wi} \rightarrow_{fbc} T_{wk}$
$T_{wi} \rightarrow_{fbb} T_{wj}$	$T_{wi} \rightarrow_{fbb} T_{wk}$	-	-	-
$T_{wi} \rightarrow_{fbc} T_{wj}$	$T_{wi} \rightarrow_{fbc} T_{wk}$	-	-	-
$T_{wi} \rightarrow_{fba} T_{wj}$	$T_{wi} \rightarrow_{fba} T_{wk}$	-	-	-
$T_{wi} \rightarrow_{fbt} T_{wj}$	$T_{wi} \rightarrow_{fbt} T_{wk}$	-	-	$T_{wi} \rightarrow_{fbt} T_{wk}$

Table 4: Interaction between Event Enforcement Dependencies(2)

We next inspect this type of dependency interactions. We first investigate the interaction of two event enforcement type of dependencies. The result is shown in table 3 and 4.

We also examine the interaction of one event enforcement type dependency and one event ordering type of dependency. There is no dependency interactions found in this case. We next investigate the interaction of two event ordering type of dependencies. The result is shown in table 5.

We are specially interested in the interaction relationships among the enforcement type of dependencies, say sc, a, fca and ex. The interactions among these dependencies are very useful in many applications. The interaction relationships among these dependencies are shown in Figure 5.

$T_{wi} \rightarrow_? T_{wk}$	$T_{wj} \rightarrow_b T_{wk}$	$T_{wj} \rightarrow_{ba} T_{wk}$	$T_{wj} \rightarrow_{bc} T_{wk}$	$T_{wj} \rightarrow_s T_{wk}$	$T_{wj} \rightarrow_t T_{wk}$
$T_{wi} \rightarrow_b T_{wj}$	$T_{wi} \rightarrow_b T_{wk}$	$T_{wi} \rightarrow_b T_{wk}$	$T_{wi} \rightarrow_b T_{wk}$	$T_{wi} \rightarrow_b T_{wk}$	-
$T_{wi} \rightarrow_{ba} T_{wj}$	$T_{wi} \rightarrow_{ba} T_{wk}$	$T_{wi} \rightarrow_{ba} T_{wk}$	$T_{wi} \rightarrow_{ba} T_{wk}$	$T_{wi} \rightarrow_{ba} T_{wk}$	-
$T_{wi} \rightarrow_{bc} T_{wj}$	$T_{wi} \rightarrow_{bc} T_{wk}$	$T_{wi} \rightarrow_{bc} T_{wk}$	$T_{wi} \rightarrow_{bc} T_{wk}$	$T_{wi} \rightarrow_{bc} T_{wk}$	-
$T_{wi} \rightarrow_s T_{wj}$	$T_{wi} \rightarrow_s T_{wk}$	$T_{wi} \rightarrow_s T_{wk}$	$T_{wi} \rightarrow_s T_{wk}$	$T_{wi} \rightarrow_s T_{wk}$	$T_{wi} \rightarrow_t T_{wk}$
$T_{wi} \rightarrow_t T_{wj}$	-	-	-	$T_{wi} \rightarrow_s T_{wk}$	$T_{wi} \rightarrow_t T_{wk}$
$T_{wi} \rightarrow_c T_{wj}$	-	-	-	-	-

Table 5: Interaction between Ordering Type of Dependencies

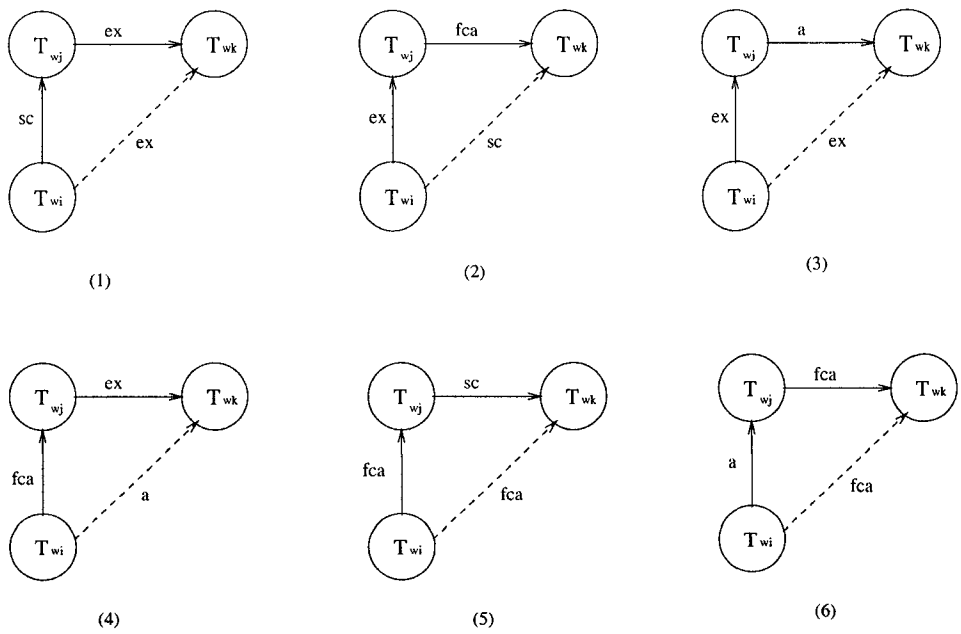

Figure 5: Interaction between Enforcement Type of Dependencies

We can prove the interaction relationship among these dependencies.

Theorem 5

Dependency $T_{wi} \rightarrow_{ex} T_{wj}$ and $T_{wj} \rightarrow_{fca} T_{wk}$ imply $T_{wi} \rightarrow_{sc} T_{wk}$.

Proof 5

$T_{wi} \rightarrow_{ex} T_{wj}$ **and** $T_{wj} \rightarrow_{fca} T_{wk}$ **imply** $T_{wi} \rightarrow_{sc} T_{wk}$.

$T_{wi} \rightarrow_{ex} T_{wj}$ means $(c_{wi} \Rightarrow a_{wj}) \vee (c_{wj} \Rightarrow a_{wi})$, which is equivalent to $(c_{wi} \Rightarrow a_{wj})$; and $T_{wj} \rightarrow_{fca} T_{wk}$ means $a_{wj} \Rightarrow c_{wk}$.

$(c_{wi} \Rightarrow a_{wj}) \bigwedge (a_{wj} \Rightarrow c_{wk}) \Rightarrow (c_{wi} \Rightarrow c_{wk})$, which exactly means the strong commit dependency. From the above, we know that $T_{wi} \rightarrow_{ex} T_{wj}$ and $T_{wj} \rightarrow_{fca} T_{wk}$ imply $T_{wi} \rightarrow_{sc} T_{wk}$.

In addition to the dependency interactions discussed above, there is another form of interactions. Consider the dependencies $T_{wi} \rightarrow_{d_x} T_{wj}$ and $T_{wi} \rightarrow_{d_y} T_{wk}$, one may wonder whether there are some relationships between subtransaction T_{wj} and T_{wk}, and, will the commit, abort or begin events of T_{wj} impose some constraints over T_{wk}. We will refer to this type of interaction as the *non-transitive* type of dependency interaction. Some control-flow dependencies can imply another dependency in the reverse direction. For example, the $T_{wi} \rightarrow_{fca} T_{wj}$ dependency implies a $T_{wj} \rightarrow_{bc} T_{wi}$ dependency. A number of the control-flow dependencies has this property. These

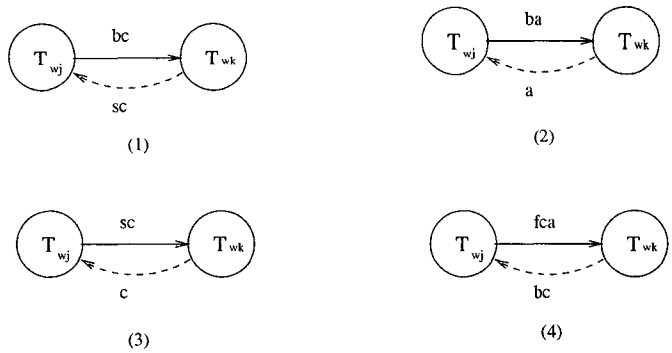

Figure 6: Examples of Implicit Dependencies

$T_{wj} \rightarrow_? T_{wk}$	$T_{wi} \rightarrow_b T_{wk}$	$T_{wi} \rightarrow_{ba} T_{wk}$	$T_{wi} \rightarrow_{bc} T_{wk}$	$T_{wi} \rightarrow_s T_{wk}$
$T_{wi} \rightarrow_{sc} T_{wj}$	-	-	$T_{wj} \rightarrow_{bc} T_{wk}$	-
$T_{wi} \rightarrow_a T_{wj}$	-	-	-	-
$T_{wi} \rightarrow_{wa} T_{wj}$	-	-	-	-
$T_{wi} \rightarrow_{fca} T_{wj}$	$T_{wj} \rightarrow_{bc} T_{wk}$	$T_{wj} \rightarrow_{bc} T_{wk}$	$T_{wj} \rightarrow_{bc} T_{wk}$	$T_{wj} \rightarrow_{bc} T_{wk}$

Table 6: Interaction between Enforcement and Ordering Dependencies

dependencies are shown in figure 6. In this situation, the results obtained from Table 3 and Table 4 can be used to detect the interactions.

We investigate different types of interactions – with two event enforcement type of dependencies, with two event ordering type of dependencies, and also the interactions between one event enforcement and one event ordering type of dependency. There results are shown in table 6 and table 7.

Theorem 6

Dependencies $T_{wi} \rightarrow_{bc} T_{wj}$ and $T_{wi} \rightarrow_{ex} T_{wk}$ imply $T_{wj} \rightarrow_{ex} T_{wk}$ as well as $T_{wj} \rightarrow_a T_{wk}$.

$T_{wj} \rightarrow_? T_{wk}$	$T_{wi} \rightarrow_{sc} T_{wk}$	$T_{wi} \rightarrow_a T_{wk}$	$T_{wi} \rightarrow_{fca} T_{wk}$	$T_{wi} \rightarrow_{ex} T_{wk}$
$T_{wi} \rightarrow_b T_{wj}$	-	-	-	-
$T_{wi} \rightarrow_{ba} T_{wj}$	-	$T_{wj} \rightarrow_{a\|ex} T_{wk}$	$T_{wj} \rightarrow_{fca\|sc} T_{wk}$	-
$T_{wi} \rightarrow_{bc} T_{wj}$	$T_{wj} \rightarrow_{sc\|fca} T_{wk}$	-	-	$T_{wj} \rightarrow_{ex\|a} T_{wk}$
$T_{wi} \rightarrow_s T_{wj}$	-	-	-	-
$T_{wi} \rightarrow_t T_{wj}$	-	-	-	-
$T_{wi} \rightarrow_c T_{wj}$	-	-	-	-

Table 7: Interaction between Ordering and Enforcement Dependencies

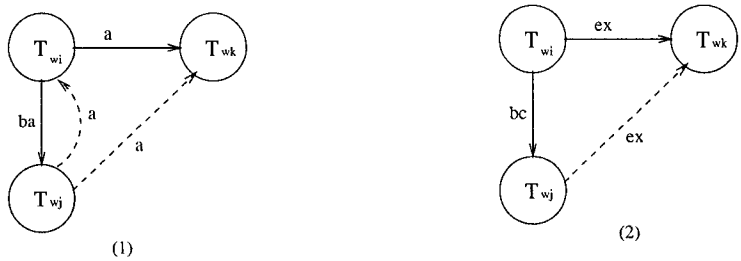

Figure 7: Examples of Non-transitive Types of Interactions

Proof 6

$T_{wi} \rightarrow_{bc} T_{wj}$ means $b_{wj} \Rightarrow (c_{wi} \prec b_{wj})$, it implies $\neg (c_{wi}) \Rightarrow \neg (b_{wj})$, which is equivalent to $b_{wj} \Rightarrow c_{wi}$. $T_{wi} \rightarrow_{ex} T_{wk}$ means $(c_{wi} \Rightarrow a_{wk}) \vee (c_{wk} \Rightarrow a_{wi})$, which is equivalent to $(c_{wi} \Rightarrow a_{wk})$. Commit of subtransaction T_{wj} implies its begin event: $(c_{wj} \Rightarrow b_{wj})$. Therefore,

$(c_{wj} \Rightarrow b_{wj}) \wedge (b_{wj} \Rightarrow c_{wi}) \wedge (c_{wi} \Rightarrow a_{wk}))$

$\Rightarrow (c_{wj} \Rightarrow c_{wi}) \wedge (c_{wi} \Rightarrow a_{wk})$

$\Rightarrow (c_{wj} \Rightarrow a_{wk})$, which is $(c_{wj} \Rightarrow a_{wk}) \vee (c_{wk} \Rightarrow a_{wj})$. This is exactly the *exclusion* dependency between T_{wj} and T_{wk}. From the above, we know that $T_{wi} \rightarrow_{bc} T_{wj}$ and $T_{wi} \rightarrow_{ex} T_{wk}$ imply $T_{wj} \rightarrow_{ex} T_{wk}$.

The abort event of subtransaction T_{wj} also implies the begin event $(a_{wj} \Rightarrow b_{wj})$. Therefore, $(a_{wj} \Rightarrow b_{wj}) \wedge (b_{wj} \Rightarrow c_{wi}) \wedge (c_{wi} \Rightarrow a_{wk})) \Rightarrow (a_{wj} \Rightarrow c_{wi}) \wedge (c_{wi} \Rightarrow a_{wk}) \Rightarrow (a_{wj} \Rightarrow a_{wk})$. And this is exactly the *abort* dependency between T_{wj} and T_{wk}. From the above, we know that $T_{wi} \rightarrow_{bc} T_{wj}$ and $T_{wi} \rightarrow_{ex} T_{wk}$ also imply $T_{wj} \rightarrow_{a} T_{wk}$.

The study of the interaction relationships between connected dependencies is important. The interaction relationships are very useful in two aspects:

1. They are useful in detecting conflicts in dependencies, as the goal for conflict-free property of advanced transactions. Based on these interactions, we can generate imaginary edges in the dependency graph to detect the indirect conflicts.

2. They are useful in detecting redundant dependencies, as the goal for minimality property of advanced transactions. Based on these interactions, we can detect whether some explicitly given dependencies are implies by interaction of other connected dependencies, if so we can

remove these dependencies, since they are redundant to the implicit dependencies and are not necessary.

For example, in figure 7 (1), we can see that $T_{wi} \rightarrow_{ba} T_{wj}$ and $T_{wi} \rightarrow_{a} T_{wk}$ imply a dependency $T_{wj} \rightarrow_{a} T_{wk}$. The appearance of this dependency can prevent ab_{wj} and cm_{wk} happen simultaneously. A $T_{wj} \rightarrow_{a} T_{wk}$ dependency between T_{wj} and T_{wk} guarantees that $T_{wj} \rightarrow_{fca} T_{wm}$ and $T_{wk} \rightarrow_{ex} T_{wm}$ will not be conflicting. As another example, let's take a look at figure 7 (2), we can see that $T_{wi} \rightarrow_{bc} T_{wj}$ and $T_{wi} \rightarrow_{ex} T_{wk}$ imply a dependency $T_{wj} \rightarrow_{ex} T_{wk}$. And, this implicit dependency $T_{wj} \rightarrow_{ex} T_{wk}$ between T_{wj} and T_{wk} will prevent the happening of an indirect conflict between $T_{wj} \rightarrow_{sc} T_{wm}$ and $T_{wk} \rightarrow_{ex} T_{wm}$, since cm_{wj} and cm_{wk} cannot occur together.

4.6 Verification of Dependency Properties using Prolog Language

4.6.1 Prolog Language and Verification

Prolog, which stands for PROgramming in LOGic, is the most widely available language in the logic programming paradigm - Prolog is based the mathematical notions of relations and logical inference. Prolog is a declarative language meaning that rather than describing how to compute a solution, a program consists of a database of facts and logical relationships (rules) which describe the relationships which hold for the given application. In a Prolog program, rather than running a function to obtain a solution, the user asks a question. When asked a question, the run time system searches through the database of facts and rules to determine (by logical deduction) the answer.

Among the features of Prolog are 'logical variables' meaning that they behave like mathematical variables, a powerful pattern-matching facility (unification), a backtracking strategy to search for proofs, uniform data structures, and input and output are interchangeable.

Prolog is widely used in artificial intelligence applications such as inference, verification, natural language interfaces, automated reasoning systems and expert systems. There systems usually consist of a database of facts and rules and an inference engine, the run time system of Prolog provides much of the services of an inference engine.

There is no structure imposed on a Prolog program, there is no main procedure, and there is no nesting of definitions. All facts and rules are global in scope and the scope of a variable is the fact or rule in which it appears. A Prolog program is executed by asking a question. The question is

called a query. Facts, rules, and queries are called clauses. A Prolog program consists of a database of facts and rules, and queries (questions). In Prolog, *Fact* is presented as: ... . *Rule* is presented as: ... :- ... . *Query* is presented as: ?- ... . *Variables* must begin with an upper case letter. *Constants* numbers must begin with lowercase letter, or enclosed in single quotes.

One of the areas that causes the most confusion in Prolog is that the Prolog functions are not like functions in C/C++, Java, or Lisp. Suppose we are to define a function designed to find the square of a number. It might look like this $floatsquare(floata)returna * a;$ in a C/C++ or Java program. We are passing in the value we want squared. The value is squared, then returned. Prolog is different. In Prolog, there are no return values. Values are simply passed into a functor as a statement, and then Prolog declares whether the statement can be proved true. Therefore, both the value to be squared AND the result should be passed in; then we get to know if those values form a valid combination.

To understand, review the following lines of Prolog code, noticing that they now return boolean values: *square(A,B) :- B is A * A.* When query Prolog with a statement like square(3,9) you are asking Prolog if it can prove the statement as being true. Prolog will state any statement that can bind to square(A,B) as being true if it can prove that the statement "B is A * A" is true. Therefore, if one queries "square(3,9).", Prolog tries to prove that "9 is 3 * 3." Since this is a rather trivial mathematical operation for Prolog to prove, it will have no problem proving that statement as true. Because it determines that 9 is in fact equal to 3 * 3, it then displays "yes", meaning that it was able to prove square(3,9) as being true.

In Prolog, *rules* extend the capabilities of a logic program. They are what give Prolog the ability to pursue its decision-making process. Facts and rules may be parameterized to produce programs in predicate logic. The parameters may be variables, atoms, numbers, or terms. Parameterization permits the definition of more complex relationships. The following program contains a number of predicates that describe a family's logical relationships.

4.6.2 Define Rules for Dependency Relationship Verification

To verify the dependency properties and interaction, we need to first build up the database of rules for inferences. We first define all the rules for ordering type of dependencies. Some most straight

forward rules are based on the definition of these dependencies, like 'beginDepen(X,Y) :- precede(begin, X, begin, Y).' means that if begin event of X is required to precede the begin event of Y, we say there is a begin dependency between X and Y. Some rules can be a bit more complex, these are for the recursive. For example, rule 'beginDepen(X,Y) :- precede(begin, X, begin, Z), beginDepen(Z, Y).' means that if there is a begin dependency between Z and Y, and, if the begin event of X is required to precede the begin event of Z, then we say there is a begin dependency between X and Y.

First we define the rules for ordering type of dependencies.

/* Define rules for ordering type of dependencies */

/* begin dependency */

beginDepen(X,Y) :- precede(begin, X, begin, Y).

beginDepen(X,Y) :- precede(begin, X, begin, Z), beginDepen(Z, Y).

beginDepen(X,Y) :- precede(begin, X, commit, Z), bcDepen(Z, Y).

beginDepen(X,Y) :- precede(begin, X, abort, Z), baDepen(Z, Y).

/* begin-on-commit dependency */

bcDepen(X,Y) :- precede(commit, X, begin, Y).

bcDepen(X,Y) :- precede(commit, X, begin, Z), beginDepen(Z, Y).

bcDepen(X,Y) :- precede(commit, X, commit, Z), bcDepen(Z, Y).

bcDepen(X,Y) :- precede(commit, X, abort, Z), baDepen(Z, Y).

/* Supporting rules for ordering dependency */

/* begin event of X is before commit/abort event of X */

precede(begin, X, commit, X).

precede(begin, X, abort, X).

/* begin event of X is before begin event of Y, if */

/* commit(abort) event of X is before begin event of Y */

precede(begin, X, begin, Y) :- precede(commit, X, begin, Y).

precede(begin, X, begin, Y) :- precede(abort, X, begin, Y).

We next gave the rules for enforcement type of dependencies.

/* Define rules for the enforcement dependencies */

/* strong commit dependency */

scDepen(X, Y) :- enforce(commit, X, commit, Y).

scDepen(X, Y) :- enforce(commit, X, commit, Z), scDepen(Z, Y).

scDepen(X, Y) :- enforce(commit, X, abort, Z), fcaDepen(Z, Y).

scDepen(X, Y) :- enforce(commit, X, begin, Z), enforce(begin, Z, commit, Y).

/* abort dependency */

abortDepen(X, Y) :- enforce(abort, X, abort, Y).

abortDepen(X, Y) :- enforce(abort, X, abort, Z), abortDepen(Z, Y).

abortDepen(X, Y) :- enforce(abort, X, commit, Z), exDepen(Z, Y).

abortDepen(X, Y) :- enforce(abort, X, begin, Z), enforce(begin, Z, abort, Y).

/* Supporting rules for enforcement dependencies */

precede(commit, X, commit, Y) :- enforce(abort, X, abort, Y).

precede(commit, Y, commit, X) :- enforce(commit, X, commit, Y).

precede(commit, Y, begin, X) :- enforce(abort, X, commit, Y).

4.6.3 Verification of Dependency Properties and Interactions

With the rules defined for both ordering types of dependencies and enforcement type of dependencies, we can set up some facts to verify the dependency properties and implicit dependencies. In every round of these verifications, we will set up some facts and then query the database of rules to verify the dependency inclusion relationships, dependency interactions, and the implicit dependencies.

4.6.3.1 *Verification of Dependency Inclusion*

With the Prolog program and rules, we can verify the dependency inclusion relationship proposed in table 1. For instance, in order to check whether begin-on-commit dependency includes some other dependencies, we can set up the facts of a begin-on-commit dependency, and then use this fact to query the Prolog program for any dependency included by this begin-on-commit dependency.

Example 3

The fact we set up is a begin-on-commit dependency between t1 and t2:

precede(commit, t1, begin, t2)..

Steps of Verifications:

[DepenVerifyProg compiled, cpu time used: 0.3100 seconds]

[DepenVerifyProg loaded]

— ?- findDep(Dependency, t1, t2).

Dependency = begin on commit.

Dependency = commit.

Dependency = begin.

Dependency = serial.

Dependency = termination.

no

The above example shows that a set of dependencies, namely commit, begin, serial, and termination dependencies, are included by the begin-on-commit dependency. This verification proves the entry in table 1. Similarly, we can verify all the dependency inclusion relationships.

4.6.3.2 *Verification of Dependency Interactions*

We can verify all the dependency interaction relationships, as show in tables 3, 4, and 5. As an example, we show how we can verify the interaction between event enforcement dependencies using Prolog.

Example 4

Take a look at the first column in table 3, $T_1 \rightarrow_{sc} T_2$ and $T_2 \rightarrow_{sc} T_3$ imply the implicit dependency

$T_1 \rightarrow_{sc} T_3$. We can set up the facts as:

/* Give facts of two sc dependencies */

enforce(commit, t1, commit, t2).

enforce(commit, t2, commit, t3).

The query should be like:

— *?- findDep(Dependency, t1, t3).*

and the query result we obtained is:

Dependency = strong commit..

The above query result proved that $T_1 \rightarrow_{sc} T_2$ dependency and $T_2 \rightarrow_{sc} T_3$ dependency imply the implicit dependency $T_1 \rightarrow_{sc} T_3$.

Similarly, we can verify all other entries in table 3 and 4. The interactions among ordering type of dependencies can also be verified using this mechanism.

Example 5

For instance, take a look of the interaction for two ordering type dependencies as shown in table 5.

/* Give facts of begin on commit dependency and a begin dependency */

precede(commit, t1, begin, t2).

precede(begin, t2, begin, t3).

Query:

— *?- findDep(Dependency, t1, t3).*

Result:

Dependency = begin on commit.

This verifies that $T_1 \rightarrow_{bc} T_2$ and $T_2 \rightarrow_{b} T_3$ imply the implicit dependency $T_1 \rightarrow_{bc} T_3$.

We can further verify the tables for non-transitive type of dependency interactions. The results of this kind of dependency interaction is show in tables 6 and 7. We give an example below.

Example 6

We can verify that $T_1 \rightarrow_{fca} T_2$ dependency and $T_1 \rightarrow_{b} T_3$ dependency imply the implicit dependency $T_2 \rightarrow_{bc} T_3$. We can give the facts of:

/* Give facts of a fca dependency and a b dependency */

enforce(abort, t1, commit, t2).

precede(begin, t1, begin, t3).

Query:

— ?- findDep(Dependency, t2, t3).

Result:

Dependency = begin on commit.

This verifies that one enforcement type dependency $T_1 \rightarrow_{fca} T_2$ and one ordering type of dependency $T_1 \rightarrow_b T_3$ can imply the implicit dependency $T_2 \rightarrow_{bc} T_3$.

By far, we have verify all the dependency properties, dependency interactions, and implicit dependencies. These properties and interaction relationships are very useful in detection for dependency conflicts and dependency redundancies, and they will enable us to build correctly specified advanced transaction.

4.7 Analysis of Dependency Conflicts

In the previous section, we outlined how to ensure that the composite dependency is conflict-free. Even though each composite dependency is conflict-free, we can still have conflicts involving multiple transactions. Figure 8 gives examples of conflicts involving multiple subtransactions. Figure 8(a) shows a conflict that occurs when there are the following dependencies: $T_{wi} \rightarrow_{sc} T_{wj}$, $T_{wj} \rightarrow_{sc} T_{wk}$ and $T_{wi} \rightarrow_{bc} T_{wk}$. Since the strong commit dependency has the transitive property, $T_{wi} \rightarrow_{sc} T_{wj}$ and $T_{wj} \rightarrow_{sc} T_{wk}$ implies $T_{wi} \rightarrow_{sc} T_{wk}$. Further, as outlined in Section 4.1, $T_{wi} \rightarrow_{sc} T_{wk}$ requires the existence of $T_{wk} \rightarrow_c T_{wi}$. This basically means that T_{wi} must commit after T_{wk}. However, the dependency $T_{wi} \rightarrow_{bc} T_{wk}$ requires that T_{wk} cannot begin until T_{wi} commits. In other words, this is an impossible situation – T_{wk} cannot begin before T_{wi} commits and at the same time T_{wk} must commit before T_{wi}. Figure 8(b) shows another example.

Before describing the different kinds of conflicts, we define what we mean by conflicts in a set of dependencies.

Definition 23

[Conflict-free Property] A set of dependencies $\mathbf{D} = \{d_1, d_2, \ldots, d_n\}$ is said to have the *conflict-free* property if there are no i dependencies, where $i \leq n$ in the set $\{d_1, d_2, \ldots, d_n\}$ such that these

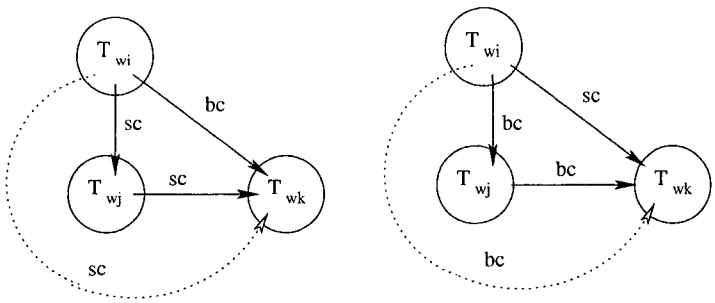

Figure 8: Conflict in Dependencies on Multiple Subtransactions

dependencies conflict with each other.

Note that this definition of conflict-free property subsumes Definition 15 which focused on conflicts in composite dependencies only.

Conflicts can occur between event ordering dependencies – infeasible ordering required by different event ordering dependencies. Conflicts can also occur between event enforcement dependencies – conflicting enforcement requirements placed by event enforcement dependencies. Finally, conflicts can also occur between event ordering dependencies and event enforcement dependencies – this is because some event enforcement dependencies imply an event ordering, and some event ordering dependencies prohibit events.

4.7.1 Conflicts between Event Ordering Dependencies

The event ordering dependencies, as the name implies, imposes an execution order on the events. It is possible for multiple event ordering dependencies to require an execution order that is not feasible in practice. To detect such infeasible requirements, we draw the graph for the generalized advanced transaction model and check for the presence of some cycles.

Theorem 7

A generalized advanced transaction has conflicting dependencies if the graph specifying the generalized transaction has a cycle $T_{wi} \rightarrow_{d_a} T_{wj} \rightarrow_{d_b} T_{wk} \ldots T_{wn} \rightarrow_{d_l} T_{wi}$ where each edge is labeled with some event ordering dependency d_x and for any pair d_m, d_n of event ordering dependencies involved either $d_m \subseteq d_n$ or $d_n \subseteq d_m$.

Proof 7

Let the generalized advanced transaction model have a cycle $T_{wi} \rightarrow_{d_a} T_{wj} \rightarrow_{d_b} T_{wk} \ldots T_{wn} \rightarrow_{d_l}$

T_{wi}. There can be three cases. The first is when all the dependencies are of begin-event ordering type. Recall that a begin-event ordering dependency $T_{wi} \rightarrow_{d_k} T_{wj}$ requires that T_{wj} can begin only after T_{wi} has completed some event (which can be begin, commit or abort). For the given cycle, because of $T_{wi} \rightarrow_{d_a} T_{wj}$, we have T_{wj} can begin only after T_{wi} has completed some event. Similarly, for the edge $T_{wj} \rightarrow_{d_b} T_{wk}$, we have the restriction that T_{wk} can begin only after T_{wj} has completed some event. Since any event of T_{wj} can occur at the same time or after its begin event, we can conclude that T_{wk} can begin only after T_{wi} has completed some event. Proceeding in this way, T_{wi} can begin only after T_{wi} has completed some event. But T_{wi} cannot complete any event before it has begun. We, therefore, arrive at a contradiction, and such a cycle imposes impossible execution ordering. The second case is the one in which all the event ordering dependencies are of the termination ordering type. The termination ordering dependency $T_{wi} \rightarrow_{d_a} T_{wj}$ require that T_{wj} cannot terminate (abort or commit) before T_{wi} does so. Similarly, the dependency $T_{wj} \rightarrow_{d_b} T_{wk}$ require that T_{wk} cannot terminate before T_{wj} terminates. The dependency $T_{wn} \rightarrow_{d_l} T_{wi}$ require T_{wn} to terminate before T_{wi}. By transitivity, we have that T_{wi} cannot terminate before T_{wi} terminates. This is impossible. The third case is that there are some begin event ordering dependencies and some termination ordering dependencies. Since any pair of dependency must either be the same or related by the inclusion relationship, the only kinds of begin event ordering dependencies that can be present in the cycle are s, wbc, bc, and ba. Suppose one or more edges in the cycle $T_{wi} \rightarrow_{d_a} T_{wj}$ have such begin event ordering dependencies. These dependencies imply that T_{wj} cannot begin until T_{wi} terminates. Since begin of T_{wj} precedes termination of T_{wj}, these dependencies imply that T_{wj} cannot terminate before T_{wi}. In other words, we can assume the existence of a termination dependency between T_{wi} and T_{wj}. Thus, for each such begin event ordering dependency, we add a termination dependency. Now all the edges in the cycle have a termination dependency. Using reasoning similar to the second case, we can prove that existence of a cycle implies impossible satisfaction of the dependencies.

4.7.2 Conflicts Between Event Enforcement Dependencies

An event enforcement dependency requires certain event to happen. The events of interest are commit, abort and begin. Since event enforcement dependencies do not prohibit events from happening,

the only conflicts possible are those in which one dependency requires some transaction T_{wi} to commit and another requires T_{wi} to abort. The following theorem formalizes the conditions under which event enforcement dependencies cause conflict.

Theorem 8

A generalized advanced transaction has conflicting dependencies if all the following conditions are satisfied:

1. any node T_{wk} in the graph specifying the generalized transaction has at least two incident edges labeled with event-enforcement dependencies, say, $T_{wi} \rightarrow_{d_x} T_{wk}$ and $T_{wj} \rightarrow_{d_y} T_{wk}$;

2. the dependency $T_{wi} \rightarrow_{d_x} T_{wk}$ imposes the constraint that the occurrence of event e_{wi} necessitates the occurrence of event e_{wk};

3. the dependency $T_{wj} \rightarrow_{d_y} T_{wk}$ puts the constraint that the occurrence of event e_{wj} requires the occurrence of event e'_{wk};

4. e_{wk} and e'_{wk} can never occur at the same time; and

5. e_{wi} and e_{wj} can occur at the same time.

Proof 8

Consider the following scenario. Suppose the events e_{wi} and e_{wj} have both occurred. The dependency $T_{wi} \rightarrow_{d_x} T_{wk}$ requires event e_{wk} to occur. The dependency $T_{wj} \rightarrow_{d_y} T_{wk}$ requires event e'_{wk} to occur. Since e_{wk} and e'_{wk} cannot both occur at the same time, it is impossible to satisfy both the dependencies and there is a conflict.

Theorem 9

The complexity of checking whether two event enforcement dependencies $T_{wi} \rightarrow_{d_x} T_{wk}$ and $T_{wj} \rightarrow_{d_y} T_{wk}$ cause a conflict or not is in the worst case $O(2^n)$ where n is the number of transactions that lie in the path from T_{wi} to T_{wj}.

Proof 9

To prove that the event enforcement dependencies do not cause a conflict, we need to show that e_{wi} and e_{wj} cannot occur simultaneously. This is only possible if there is a path from e_{wi} to e_{wj} or vice-versa and the dependencies existing on this path impose constraints that ensure $e_{wi} \Rightarrow \neg e_{wj}$. To

prove that the two dependencies do not cause conflict, we need to derive that $e_{wi} \Rightarrow \neg e_{wj}$ from the constraints implied by the dependencies of transactions connecting T_{wi} and T_{wj}. This derivation incurs an overhead of $O(2^n)$ where n is the number of transactions connecting T_{wi} and T_{wj}.

Instead of proving $e_{wi} \Rightarrow \neg e_{wj}$, we can detect the presence of certain dependencies between T_{wi} and T_{wj}. But this involves understanding what kinds of conflicts can occur between event enforcement dependencies. In the rest of this section, we discuss all the possible conflicts that can occur because of the presence of multiple event enforcement dependencies. Consider the node T_{wk} that has multiple incoming edges – let two of these edges correspond to the dependencies $T_{wi} \rightarrow_{sc} T_{wk}$ and $T_{wj} \rightarrow_{ex} T_{wk}$. If both T_{wi} and T_{wj} commit, then T_{wk} is required to commit and abort which is not possible. In such a case we have a conflict, unless some other dependencies exist between T_{wi} and T_{wj} which prevent the simultaneous commitment of T_{wi} and T_{wj}. One way to check this is to see if T_{wi} and T_{wj} are connected by edges. If not, then T_{wi} and T_{wj} can commit independently. Otherwise, we need to check whether the constraints imposed by the dependencies that exist between the T_{wi} and T_{wj} imply $c_{wi} \Rightarrow \neg c_{wj}$. For instance, if either $T_{wi} \rightarrow_{ex} T_{wj}$ or $T_i \rightarrow_{ba} T_{wj}$ or $T_{wj} \rightarrow_{ba} T_{wi}$ exists then T_{wi} and T_{wj} do not commit together. If none of these dependencies exist between T_{wi} and T_{wj}, we have a problem.

The next problematic case is when $T_{wi} \rightarrow_{fca} T_{wk}$ and $T_{wj} \rightarrow_{a} T_{wk}$. When T_{wi} and T_{wj} abort, we have a problem because T_{wk} is required to both commit and abort. We do not have a problem if dependencies between T_{wi} and T_{wj} ensure that T_{wi} and T_{wj} do not abort at the same time. This is possible if either $T_{wi} \rightarrow_{fca} T_{wj}$, $T_{wj} \rightarrow_{fca} T_{wi}$, $T_{wi} \rightarrow_{bc} T_{wj}$, or $T_{wj} \rightarrow_{bc} T_{wj}$ exist. Each of these dependencies ensure that $a_{wi} \Rightarrow \neg a_{wj}$ and both a_{wi} and a_{wj} do not occur together.

The last case is when $T_{wi} \rightarrow_{fca} T_{wk}$ and $T_{wj} \rightarrow_{ex} T_{wk}$. Here again, we have a problem if T_{wi} aborts and T_{wj} commits. The presence of any of the following dependencies $T_{wi} \rightarrow_{a} T_{wj}$, $T_{wj} \rightarrow_{ba} T_{wi}$, or $T_{wj} \rightarrow_{sc} T_{wi}$ ensure that a_{wi} and c_{wj} do not occur at the same time. This is because $a_{wi} \Rightarrow c_{wj}$ can be inferred from any of the above dependencies.

One might think that there is a problem with the dependencies $T_{wi} \rightarrow_{sc} T_{wk}$ and $T_{wj} \rightarrow_{a} T_{wk}$. If T_{wi} commits and T_{wj} aborts, then the two dependencies may require T_{wk} to commit and abort. This is clearly not possible. But the very nature of dependencies do not allow T_{wi} to commit and T_{wj} to abort. This is because the dependency $T_{wi} \rightarrow_{sc} T_{wk}$ implies the existence of

the dependency $T_{wk} \rightarrow_a T_{wi}$. Since abort dependency is transitive in nature, there is an implicit dependency $T_{wj} \rightarrow_a T_{wi}$. The possibility of T_{wi} committing when T_{wj} aborts does not arise in this case.

Algorithm for Detecting Conflicts caused by sc and ex dependencies.

Algorithm 5

Detect Conflicts by sc and ex dependencies

Input: the advanced transaction specification $AT_w =< S, D, C >$

Output: a result showing whether there is conflict

Procedure DetectScExConflict(AT_w)

begin

for each node T_{wk} that has incident edges of the form $T_{wi} \rightarrow_{sc} T_{wk}$ and $T_{wj} \rightarrow_{ex} T_{wk}$

if $(T_{wi}, T_{wj}) \in E \vee (T_{wj}, T_i) \in E$

if $L(T_{wi}, T_{wj}) \cap \{ex, ba\} \neq \{\} \vee L(T_{wj}, T_{wi}) \cap \{ex, ba\} \neq \{\}$

return false

if there is a path P from T_{wi} to T_{wj} or from T_{wj} to T_{wi}

for each $(T_{wx} \rightarrow_{d_z} T_{wy}) \in P$

if $ba \notin L(T_{wx}, T_y)$

return true

return true

end

The above algorithm reduces the complexity of evaluating whether the incident edges $T_{wi} \rightarrow_{sc} T_{wk}$ and $T_{wj} \rightarrow_{ex} T_{wk}$ cause a conflict or not. It does this by checking whether certain kinds of dependencies exist between T_{wi} and T_{wj}. If these dependencies are present, then $T_{wi} \rightarrow_{sc} T_{wk}$ and $T_{wj} \rightarrow_{ex} T_{wk}$ do not cause conflict. This algorithm, however, reports false positives. It may report a conflict, when it is actually not present. For instance, consider the following example shown below. $T_{wi} \rightarrow_{ex} T_{wk}$, $T_{wj} \rightarrow_{sc} T_{wk}$, $T_{wi} \rightarrow_{ex} T_{wn}$ and $T_{wn} \rightarrow_{sc} T_{wj}$. Although this is conflict-free, our algorithm will report the existence of a conflict. Such false positives will not be encountered if we

do an analysis of the predicates. However, the exponential time complexity precludes using that approach.

4.7.3 Conflicts between Event Ordering and Event Enforcement Dependencies

There are two reasons why conflicts may occur between event enforcement dependencies and event ordering dependencies. The first reason is because some event enforcement dependencies impose an execution order (please refer Section 4.1). Such execution orders may conflict with other event ordering dependencies. For instance, the strong commit dependency $T_{wi} \rightarrow_{sc} T_{wj}$ implies the dependency $T_{wj} \rightarrow_{c} T_i$. This additional implied dependency might give rise to conflicts. To detect such conflicts, we insert edges for such implicit dependencies. As outlined in Section 4.7.1, we check for the presence of cycles to detect such conflicts.

The second reason why conflicts can occur is because sometimes event ordering dependencies require the prohibition of some events. Specifically, there are some event ordering dependencies $T_{wi} \rightarrow_{x} T_{wj}$ that do not allow T_{wj} to begin until T_{wi} has completed some event e_{wi}. In other words, in the absence of occurrence of e_{wi}, b_{wj} cannot take place.

Theorem 10

A generalized advanced transaction has conflicting dependencies if all the following conditions are satisfied:

1. any node T_{wk} in the graph specifying the generalized transaction has an incident edge labeled with an event ordering dependency, say, $T_{wi} \rightarrow_{d_x} T_{wk}$ and another incident edge labeled with an event enforcement dependency $T_{wj} \rightarrow_{d_y} T_{wk}$;

2. the event ordering dependency $T_{wi} \rightarrow_{d_x} T_{wk}$ imposes the constraint that the absence of event e_{wi} prohibits the occurrence of event e_{wk};

3. the event enforcement dependency $T_{wj} \rightarrow_{d_y} T_{wk}$ imposes the constraint that the occurrence of event e_{wj} requires the occurrence of event e_{wk};

4. the absence of event e_{wi} and the presence of event e_{wj} is possible at the same time.

Proof 10

Since occurrence of event e_{wj} does not guarantee the occurrence of e_{wi}, it is possible that only

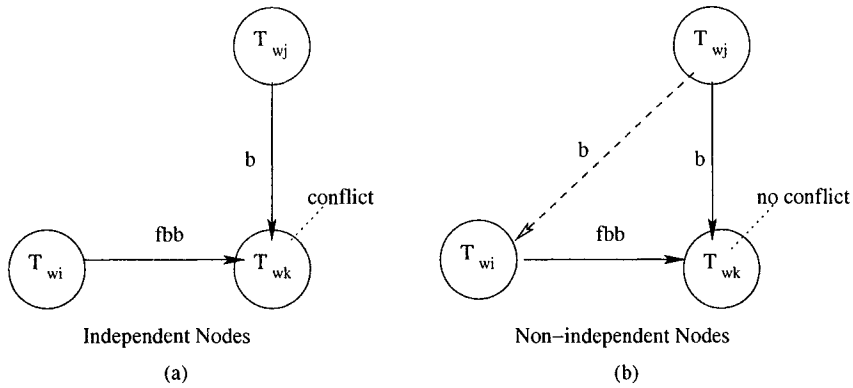

Figure 9: Conflicts with Force-begin-on-begin Dependency and Begin Dependency

e_{wj} will occur and not e_{wi}. Since e_{wj} has occurred, by virtue of the dependency $T_{wj} \rightarrow_{d_y} T_{wk}$, e_{wk} is required to occur. Moreover, since e_{wi} has not occurred, e_{wk} cannot occur because of the dependency $T_{wi} \rightarrow_{d_x} T_{wk}$. Thus, the two dependencies impose conflicting requirements on event e_{wk}.

For instance, the event ordering dependency $T_{wj} \rightarrow_b T_{wk}$ does not allow T_{wk} to begin unless T_{wj} has begun. Suppose there is an event enforcement dependency $T_{wi} \rightarrow_{fbb} T_{wk}$. Please refer to Figure 9. Now if T_{wi} begins and T_{wj} does not begin, we have a problem: T_{wk} is required to begin and not begin. If the begin operations of T_{wi} and T_{wj} occur independently, then the two given dependencies are impossible to satisfy. However, if there exist some dependency between T_{wi} and T_{wj} that ensure that b_{wi} and $\neg b_{wj}$ will never occur together, that is, $b_{wi} \Rightarrow b_{wj}$, then we do not have a problem. The dependency $T_i \rightarrow_{fbb} T_{wj}$ ensure this and so does $T_{wj} \rightarrow_b T_{wi}$. If either of these two dependencies do not exist between T_{wi} and T_{wj}, then the dependencies $T_i \rightarrow_{fbb} T_{wk}$ and $T_{wj} \rightarrow_b T_{wk}$ cause a conflict.

Next, consider the case of dependency $T_{wi} \rightarrow_{fbb} T_{wk}$ and $T_{wj} \rightarrow_{ba} T_{wk}$. Suppose T_{wi} has begun and T_{wj} has not aborted. In such a scenario, T_{wk} is required to begin as well as not begin. This is not possible. The only way this situation can be avoided is if there is some dependency between T_{wi} and T_{wj} that prevents the simultaneous occurrence of b_{wi} and $\neg a_{wj}$. In other words, if $b_{wi} \Rightarrow a_{wj}$ can be inferred from the dependency that exists between T_{wi} and T_{wj}, then we do not have a problem. This happens when $T_{wj} \rightarrow_{ba} T_{wi}$ exists. When this dependency exist, then we may have a situation that is impossible to satisfy. We have a similar problem for the case when

$T_i \rightarrow_{fbb} T_{wj}$ and $T_{wj} \rightarrow_{bc} T_{wk}$. For this case, we need the presence of the dependency $T_{wj} \rightarrow_{bc} T_{wi}$ to avoid the conflict.

Similarly, the dependencies $T_{wi} \rightarrow_{fba} T_{wk}$ and $T_{wj} \rightarrow_{b} T_{wk}$ result in a conflict unless either $T_{wi} \rightarrow_{fba} T_{wj}$ or $T_{wj} \rightarrow_{b} T_{wi}$ exist between T_{wi} and T_{wj}. The dependencies $T_{wi} \rightarrow_{fba} T_{wk}$ and $T_{wj} \rightarrow_{ba} T_{wk}$ are problematic unless either $T_{wi} \rightarrow_{a} T_{wj}$ or $T_{wj} \rightarrow_{ba} T_{wi}$ exist between T_{wi} and T_{wj}. The dependencies $T_{wi} \rightarrow_{fba} T_{wk}$ and $T_{wj} \rightarrow_{bc} T_{wk}$ are problematic unless either $T_{wi} \rightarrow_{fca} T_{wj}$ or $T_{wj} \rightarrow_{bc} T_{wi}$ exist between T_{wi} and T_{wj}.

Similar problems are created by other dependencies as well. $T_{wi} \rightarrow_{fbc} T_{wk}$ and $T_{wj} \rightarrow_{b} T_{wk}$ result in a conflict unless either $T_{wi} \rightarrow_{fbc} T_{wj}$ or $T_{wj} \rightarrow_{b} T_{wi}$ exist between T_{wi} and T_{wj}. $T_{wi} \rightarrow_{fbc} T_{wk}$ and $T_{wj} \rightarrow_{ba} T_{wk}$ result in a conflict unless $T_{wj} \rightarrow_{ba} T_{wi}$ exists. Similarly, $T_{wi} \rightarrow_{fbc} T_{wk}$ and $T_{wj} \rightarrow_{bc} T_{wk}$ result in a conflict unless either $T_{wi} \rightarrow_{sc} T_{wj}$ or $T_{wj} \rightarrow_{bc} T_{wi}$ exist between T_{wi} and T_{wj}. $T_{wi} \rightarrow_{fbt} T_{wk}$ and $T_{wj} \rightarrow_{ba} T_{wk}$ result in a conflict unless $T_{wj} \rightarrow_{ba} T_{wi}$ exists. $T_{wi} \rightarrow_{fbt} T_{wk}$ and $T_{wj} \rightarrow_{bc} T_{wk}$ result in a conflict unless $T_{wj} \rightarrow_{bc} T_{wi}$ exists. $T_{wi} \rightarrow_{fbt} T_{wk}$ and $T_{wj} \rightarrow_{b} T_{wk}$ result in a conflict unless either $T_{wj} \rightarrow_{b} T_{wi}$ or $T_{wi} \rightarrow_{fbt} T_{wj}$ exists.

Sometimes event ordering dependencies, such as, *ba* and *bc* prohibit the occurrence of events. $T_{wi} \rightarrow_{ba} T_{wk}$ and $T_{wj} \rightarrow_{ba} T_{wk}$ result in a problem unless there exists a dependency of the form $T_{wi} \rightarrow_{a} T_{wj}$. Similar situations are created by the dependencies $T_{wi} \rightarrow_{bc} T_{wk}$ and $T_{wj} \rightarrow_{bc} T_{wk}$. There is a problem unless there exists a dependency of the form $T_{wi} \rightarrow_{sc} T_{wj}$. Also, $T_{wi} \rightarrow_{ba} T_{wk}$ and $T_{wj} \rightarrow_{bc} T_{wk}$ conflict unless there exists a dependency of the form $T_{wi} \rightarrow_{ex} T_{wj}$ or $T_{wi} \rightarrow_{ba} T_{wj}$.

Once such relationships between dependencies are identified and tabulated, conflict detection is simplified. Conflict detection involves representing the advanced transaction in the form of a graph, checking for certain cycles in the graph, looking for nodes having multiple incident edges corresponding to dependencies and checking whether the source nodes of these edges are related by certain dependencies.

4.8 *Algorithms for Detecting Conflicts*

Since the behavior of an advanced transaction having dependency conflicts is unpredictable, it is important to analyze the transaction before it can be executed. Such analysis can be done statically.

In this section, we present algorithms for automatically detecting conflicts. Prior to an advanced transactions could be submitted to the scheduler for execution, conflicts with the dependencies must be detected and removed. From the discussion of Section 4.7, we have been aware of how different kinds of conflicts are formed and how to detect them. However, due to the huge amount of implicit dependencies and numerous types of indirect conflicts, manually checking all the dependency conflicts would be a very time consuming and error-prone task. Algorithms that can automatically generate all the implicit edges, and then detect and remove dependency conflicts are greatly helpful. In this section, we focus on the algorithms and implementation details on performing dependency analysis and detection for dependency conflicts.

To detect the conflicts, the structure of *dependency graph* described in Section 3.1 is useful. Recall that a dependency graph is composed of nodes and edges, where nodes are the subtransactions and edges are the dependencies in an advanced transaction. There are two equivalent ways to maintain the dependency graph in memory, either with a real graph data structure, or using a set structure to model the graph. A graph data structure is intuitively the straight forward way, however, frequent implicit dependency insertion operations and edge comparing operations will make the graph implementation time-costing. In our implementation, we choose the dependency set data structure to maintain the dependencies graph. Every dependency in the dependency set is corresponding to an edge in the dependency graph. When connecting all the dependencies in a dependency set up node by node, one can easily rebuild the structure of the dependency graph.

The conflict detection algorithm is organized in three major steps - (i) generate initial dependency set, (ii) generate derived dependency set, and (iii) detect conflicts of dependencies. First, an initial dependency set could be built using the specification of the advanced transaction. Then, the derived dependency set will be generated, based on explicit edges in the initial dependency set. For conflict analysis, we need to insert all the implicit dependencies in the derived dependency set. The procedure of building up the derived dependency set is organized in a loop. In each iteration, new implicit dependencies, which could be derived from the current dependency set, are inserted into the derived dependencies set. The loop of operations will continue until no more new edge can be derived.

The rest is to detect the different kinds of conflicts. All these detection procedures are based

on the derived dependency set. For each type of dependency conflicts discussed in Section 4.7, a corresponding procedure is needed to detect this type of conflict. Our conflicts with all the composite dependencies; conflicts with the event ordering type of dependencies; conflicts with the event enforcement dependencies; and conflicts with pairs of event enforcement and event ordering dependencies. These procedures could be performed in any order, since detecting for each type of conflicts is independent.

The algorithms for dependency conflict detection and the implementation details are described following.

Step 1: Generate the initial dependency set (IDS) for the advanced transaction. This is based on the specifications of the advanced transaction, which are provided by the application developer. The initial dependency set records every dependency $T_{wi} \rightarrow_{d_x} T_{wj}$ in the specification as an edges of the form (T_{wi}, T_{wj}, d_x), where T_{wi} is the source node, T_{wj} is the destination node, and d_x in the type of dependency associated between T_{wi} and T_{wj}.

Algorithm 6

Generate the Initial Dependency Set

Input: the advanced transaction specification $AT_t =< S, D, C >$

Output: an initial dependency set IDS

Procedure GenerateIDS(AT_t)

STEP 1: generate initial dependency set IDS

Begin

//Build the initial dependency set, according to the specification

IDS = {}; // set of edges

NS = {}; // set of nodes

For every dependency $(T_{wi} \rightarrow_{d_x} T_{wj}) \in AT_t(D)$

generate an edge (T_{wi}, T_{wj}, d_x); //dependency of type x from T_{wi} to T_{wj}

$IDS = IDS + (T_{wi}, T_{wj}, d_x)$;

For every subtransaction $(T_{wi}) \in AT_t(S)$

generate a node (T_{wi});

$NS = NS + (T_{wi})$;

End

Step 2: Generate the derived dependency set (DDS). The derived dependency set will be built based on the initial dependency set generated in step 1. This procedure is organized in loops. In every iteration, the algorithm scans all the dependencies currently in the DDS and check whether new edges could be implied by the existing edges. (i) for each symmetric dependency, insert a corresponding implied edge; (ii) for each dependency implying an implicit dependency (like sc, fca, a), insert the corresponding edge; (iii) for each pair of dependencies with interaction relationships, insert the corresponding implied edge. The loops are repeated until no more new edges can be derived.

In each round, newly generated implicit edges will be put into the set and they will be marked as *unchecked*. These edges newly inserted could possibly imply more new implicit edges, and their relationship will be checked in the next round. The algorithm keeps checking new edges round by round, until in certain round there is no new edge can be derived. Now we have the derived dependency set built up.

Algorithm 7

Generate the Derived Dependency Set

Input: the IDS of advanced transaction

Output: a derived dependency set (DDS) of the advanced transaction

Procedure GenerateDDS

STEP 2: generate the derived dependency set DDS

Begin

// initiate the derived dependency set

done == false;

initiate $DDS == IDS$;

mark every edge in DDS as *unchecked*

While (done = false)

Begin

For each edge $(T_{wi}, T_{wj}, d_x) \in DDS$

Begin

// If this dependency implies a relationship, insert the implicit edge

If this edge is *unchecked*

If $d_x = sc$

generate an edge (T_{wj}, T_{wi}, c);

else If $d_x = a$

generate an edge (T_{wi}, T_{wj}, c);

else If $d_x = fca$

generate an edge (T_{wj}, T_{wi}, bc);

else If $d_x = ba$

generate an edge (T_{wj}, T_{wi}, a);

else If $d_x = bc$

generate an edge (T_{wj}, T_{wi}, sc);

else If $d_x = ex$

generate an edge (T_{wj}, T_{wi}, ex);

// insert the implied edges based on interaction of dependencies

For each edge $(T_{wj}, T_{wk}, d_p) \in DDS$ that is *unchecked*

If (T_{wi}, T_{wj}, d_x) and (T_{wj}, T_{wk}, d_p) imply an interaction dependency d_t

generate an edge (T_{wi}, T_{wk}, d_t);

For each edge $(T_{wi}, T_{wk}, d_q) \in DDS$ that is *unchecked*

If (T_{wi}, T_{wj}, d_x) and (T_{wi}, T_{wk}, d_q) imply an interaction dependency d_s

generate an edge (T_{wj}, T_{wk}, d_s);

End

// mark existing edges as *checked*;

For every edge $\in DDS$

set edge as *checked*

// insert newly generated edges, make them as *unchecked*;

For every newly generated edge e_{new} in this round

set edge e_{new} as *unchecked*

$DDS = DDS + e_{new}$

If there is no new edge inserted in this round

done == true;

End

End

Step 3. Detect conflicts with the composite dependencies, using the DDS obtained in step 2. When multiple dependencies are specified between a pair of transactions, we need to ensure that this composite dependency does not impose conflicting requirements. The table of conflicting dependency pairs is shown in Table 2. In this step, we visit all the edges one by one, and find each pair of edges with the same source node and destination node; then, we check whether the two dependencies are conflicting or not. If they are conflicting, report error.

Algorithm 8

Detecting Conflicts in Composite Dependencies

Input: the derived dependency set DDS

Output: message identifying whether there are conflicts

Procedure DetectCompositeDependency

STEP 3: detect conflicts with composite dependencies

Begin

For each edge $(T_{wi}, T_{wj}, d_x) \in DDS$

begin

For each edge (T_{wm}, T_{wn}, d_y) other than $(T_{wi}, T_{wj}, d_x) \in DDS$

begin

// check if they belong to same composite dependency

If $T_{wi} = T_{wm}$ **AND** $T_{wj} = T_{wn}$ // a composite pair

begin

check the conflicting table for d_x

$\{Conflict\_Set\}$ = all conflicting dependencies for d_x

// check whether they are conflicting

If $d_y \in \{Conflict\_Set\}$

return "conflict"

end

end

return "no conflict";

End

Step 4. Detect conflicts with the event ordering type of dependencies, using the DDS set. We detect whether there is a cycle of a common type of event ordering dependencies existing in the DDS graph. If there is such a cycle, report conflicts. For every pair of connected edges (T_{wi}, T_{wj}, d_1) and (T_{wj}, T_{wk}, d_2), if d_1 and d_2 are of same ordering type, we insert an imaginary edge (T_{wi}, T_{wk}, d_1); else, if d_1 and d_2 are ordering type and they are of inclusion relationship, (assuming the included dependency is d_z) we insert an imaginary edge (T_{wi}, T_{wk}, d_1). With these imaginary edges inserted, if there exists a cycle of a common type d_z of ordering dependencies, we will eventually meet a pair of edges (T_{wp}, T_{wq}, d_z) and (T_{wq}, T_{wp}, d_z), and it means a conflict. If there is no such cycles, we report there is no conflict.

Algorithm 9

Detecting Conflicts with Ordering Dependencies

Input: the derived dependency set DDS

Output: message identifying whether there are conflicts

Procedure DetectOrderingConflicts

STEP 4: detect conflicts with event ordering dependencies

Begin

$\{ImaginarySet\} = DDS$;

//Check of there are conflicts (cycles) in event ordering dependencies

For each ordering type edge $(T_{wi}, T_{wj}, d_1) \in \{ImaginarySet\}$, where $d_1 \in \{c, t, s, b, ba, bc\}$

Begin

// search for cycle $T_{wi} \rightarrow_d T_{wj}$ and $T_{wj} \rightarrow_d T_{wi}$

If ($((T_{wj}, T_{wi}, d_2) \in \{ImaginarySet\})$ **AND** ($(d_1 = d_2)$ **OR** $(d_1 \subseteq d_2)$ **OR** $(d_2 \subseteq d_1)$))

return "conflict"

// ordering dependencies are transitive, insert imaginary edges

else If $(T_{wj}, T_{wk}, d_2) \in \{ImaginarySet\}$ **AND** ($(d_1 = d_2)$ **OR** $(d_1 \subseteq d_2)$ **OR** $(d_2 \subseteq d_1)$)

If $(d_1 = d_2)$ OR $(d_1 \subseteq d_2)$

$d_z == d_1$

else If $d_2 \subseteq d_1$

$d_z == d_2$

$\{ImaginarySet\} = \{ImaginarySet\} + (T_{wi}, T_{wk}, d_z)$

End

return "no conflict"

remove $\{ImaginarySet\}$

End

Step 5. Detect conflicts between pairs of event enforcement dependencies who are incident on the same node. If multiple edges are incident on a same node, check whether the edges cause any problem. Report conflicts if there are problems.

The major challenge is how to identify whether the two source nodes, with dependencies incident on the same destination node, are independent or not. By generating all the derived edges, the above algorithm reduces the complexity of this checking. It does this by checking whether certain kinds of desired dependencies (either explicit or implicit) exist between the two source nodes T_{wi} and T_{wj}, if these dependencies are present, then T_{wi} and T_{wj} are not independent, and the

dependencies are not conflicting; otherwise the dependencies are conflicting. For example, for dependencies $T_{wi} \rightarrow_{sc} T_{wk}$ and $T_{wj} \rightarrow_{ex} T_{wk}$, if there exists a derived edge $T_{wk} \rightarrow_{ex} T_{wi}$, this dependency specification does not cause a conflict.

Algorithm 10

Detecting Conflicts with Event Enforcement Dependencies

Input: the derived dependency set DDS

Output: message identifying whether there are conflicts

Procedure DetectEnforcementConflict

STEP 5: detect conflicts between event enforcement dependencies

Begin

//Check conflicts with event enforcement dependencies who are incident on the same node

For every node $T_{wk} \in NS$

Begin

If there exists a pair of edges (T_{wi}, T_{wk}, sc) and $(T_{wj}, T_{wk}, ex) \in DDS$

If there is no $((T_{wi}, T_{wj}, ex)$ **OR** (T_{wi}, T_{wj}, ba) **OR** $(T_{wj}, T_{wi}, ba))$

return "conflict";

If there exists a pair of edges (T_{wi}, T_{wk}, fca) and $(T_{wj}, T_{wk}, ex) \in DDS$

If there is no $((T_{wi}, T_{wj}, a)$ **OR** (T_{wj}, T_{wi}, ba) **OR** $(T_{wj}, T_{wi}, sc))$

return "conflict";

If there exists a pair of edges (T_{wi}, T_{wk}, fca) and $(T_{wj}, T_{wk}, a) \in DDS$

If there is no $((T_{wi}, T_{wj}, fca)$ **OR** (T_{wi}, T_{wj}, bc) **OR** $(T_{wj}, T_{wi}, fca))$ **OR** $(T_{wj}, T_{wi}, bc))$

return "conflict";

End

return "no conflict"

End

Step 6. Detect conflicts between pairs of event enforcement dependency and event ordering dependency, who are incident on the same node. Please refer to Figure 9 for this kind of dependency

conflicts. The structure of this procedure is very similar to step 5.

Algorithm 11

Detecting Conflicts with Event Enforcement and Ordering Dependencies

Input: the derived dependency set DDS

Output: message identifying whether there are conflicts

Procedure DetectEnforcementOrderingConflict

STEP 6: detect conflicts between an event enforcement dependency and an event ordering dependency

Begin

//Check conflicts with enforcement and ordering dependencies

For every node $T_{wk} \in NS$

Begin

If there exists a pair of edges $(T_{wi}, T_{wk}, fbb \mid c \mid a \mid t)$ and $(T_{wj}, T_{wk}, b) \in DDS$

If there is no $((T_{wi}, T_{wj}, b \mid ba \mid bc)$ **OR** $(T_{wj}, T_{wi}, b \mid ba \mid bc))$

return "conflict";

If there exists a pair of edges $(T_{wi}, T_{wk}, fbb \mid c \mid a \mid t)$ and $(T_{wj}, T_{wk}, ba) \in DDS$

If there is no $((T_{wi}, T_{wj}, b \mid ba \mid bc)$ **OR** $(T_{wj}, T_{wi}, b \mid ba \mid bc))$

return "conflict";

If there exists a pair of edges $(T_{wi}, T_{wk}, fbb \mid c \mid a \mid t)$ and $(T_{wj}, T_{wk}, bc) \in DDS$

If there is no $((T_{wi}, T_{wj}, b \mid ba \mid bc)$ **OR** $(T_{wj}, T_{wi}, b \mid ba \mid bc))$

return "conflict";

End

return "no conflict"

End

The generation of the derived dependency set is crucial in this algorithm. Recall that, a number primitive dependencies could imply implicit dependencies, a implies c for instance; and, the interactions between dependencies could impose some implicit dependency relationships, for example,

ex could be derived from sc and fca. We must take into consideration the effects of these derived dependencies in checking for dependency conflicts. If not all the implicit dependencies are generated properly, potential dependency conflicts would remain undetected, and the detection result would be inaccurate.

After combining the dependencies with the same source node and destination node, the conflicts with composite dependencies could be detected – we examine the combined dependencies using the conflicting dependency Table 2. With the help of the derived dependency set, conflicts with event ordering type of dependencies could be detected if there exists a cycle of the same type of ordering dependency. Similarly, other types dependency conflicts can be detected with the derived graph.

4.9 Well-Structured Generalized Advanced Transaction

Definition 24

[Well-Structured Generalized Advanced Transaction] A generalized advanced transaction $T =< S, D, C >$ is well-structured if the following conditions hold:

1. subtransactions appear in D and C must be defined in S;
2. dependencies in D satisfy the conflict-free property;
3. dependencies in D satisfy the minimality property; and
4. completion sets C conforms to the dependencies specified in D.

All the advanced transactions must be well-structured. Any advanced transaction that is not well-structured should not be submitted to the scheduler, since it could lead to problems like deadlock and unavailability during execution. With the proposed algorithm for detection and removal of dependency conflicts and redundancies, we can find out the improper dependency structure in an advanced transaction specification. Further, the algorithms proposed earlier for inspecting completion sets and minimal dependency sets will also help one to detect and remove the incorrect dependency specifications. The criteria in the definition of well-structured generalized advanced transactions will enable us to develop a transaction processing system that is capable of processing various kinds of advanced transactions.

In this chapter, we have shown how dependencies present in advanced transaction models can be analyzed to ensure correct execution. We focused on two important properties of dependencies: conflict-free and minimality. Conflict-free property ensures that the dependencies are physically realizable. It also ensures that the dependencies are free from deadlocks. Minimality ensures that no redundant dependencies are specified. This helps to eliminate the processing of unnecessary dependencies.

CHAPTER V

RELIABLE EXECUTION MODEL AND SCHEDULER

Advanced transactions are characterized by having a number of component subtransactions whose executions are controlled by dependencies. The dependencies pose new challenges which must be addressed to ensure secure and reliable execution of advanced transactions. Violation of dependencies in advanced transactions could lead to unavailability of resources and information integrity problems. Although advanced transactions have received a lot of attention, not much work appears in addressing these issues. For instance, Adam, Atluri and Huang [1] have discussed modeling and analysis of workflows using Petri Nets. But they did not discuss how to reliably schedule an advanced transaction and enforce the dependencies. J. Tang [111] and Attie at el. [13] proposed different way to specify and enforce intertask dependencies in workflows. However, they did not formalize the control-flow dependencies as in the ACTA format. And they failed to discuss how to reliably execute advanced transactions with clearly specified termination states. To the best of our knowledge, none of the existing papers discuss detailed solutions to correctly enforce dependencies and reliably schedule advanced transactions.

Correct execution of advanced transactions is a non-trivial problem due to the constraints imposed by the dependencies. Improper scheduling of subtransactions in an advanced transaction may result in integrity and availability problems. For instance, suppose there is a *begin on commit* dependency between subtransactions T_{w1} and T_{w2}, which requires that T_{w2} cannot begin until T_{w1} commits. If the scheduler fails to enforce this dependency, then the integrity of the application may be compromised. As a second example, consider the existence of a *strong commit* dependency between transactions T_{w3} and T_{w4} that requires T_{w4} to commit if T_{w3} does so. Suppose the scheduler executes and commits T_{w3} before T_{w4}. Later if T_{w4} needs to be aborted for some reason, then we have a complex situation: T_{w4} needs to abort as well as commit. In such a case, allowing T_{w4} to commit or abort will cause integrity problems and keeping it incomplete raises issues pertaining to availability. Reliable scheduling algorithms for advanced transactions are needed to ensure that

dependencies are not violated during the normal execution of advanced transactions.

In this chapter, we propose solutions for reliable scheduling and recovery that overcome the problems mentioned above. The execution model will be based on the generalized advanced transaction as defined in Chapter 3. We assume that the advanced transaction is well-structured, that is, the dependencies in an advanced transaction do not impose conflicting constraints and in any execution it will be possible to commit all subtransactions specified by some completion set. We give an abstract architecture of our proposed system that includes the scheduler and recovery manager and shows the interactions between the various components. We enumerate the scheduling constraints imposed by each dependency and investigate how to ensure its satisfaction during execution. Later we discuss how to handle the constraints imposed by multiple dependencies. We describe the data structures that are needed by our scheduler and provide algorithms that ensure the satisfaction of dependencies. We formalize the notion of *reliable scheduling* and prove that the proposed algorithm ensures reliable scheduling.

5.1 Architecture of our System

In this chapter, we focus on the scheduler and recovery manager for advanced transactions. Figure 10 shows the main components and the interactions between these components to ensure the correct execution of advanced transactions. Our system consists of transaction manager, scheduler, recovery manager, log manager, and buffer manager.

Transaction Manager: The transaction manager accepts advanced transactions submitted by users. These transactions are then forwarded to the scheduler. It interacts with the scheduler to get the execution result which is then forwarded to the users.

Scheduler: The scheduler is responsible for determining the execution order of subtransactions in advanced transactions. The scheduler has a concurrency control mechanism that uses Strict Two-Phase Locking (Strict 2PL) to execute subtransactions of advanced transactions. During different execution stages, the scheduler sends messages to log manager to record the corresponding actions in the log.

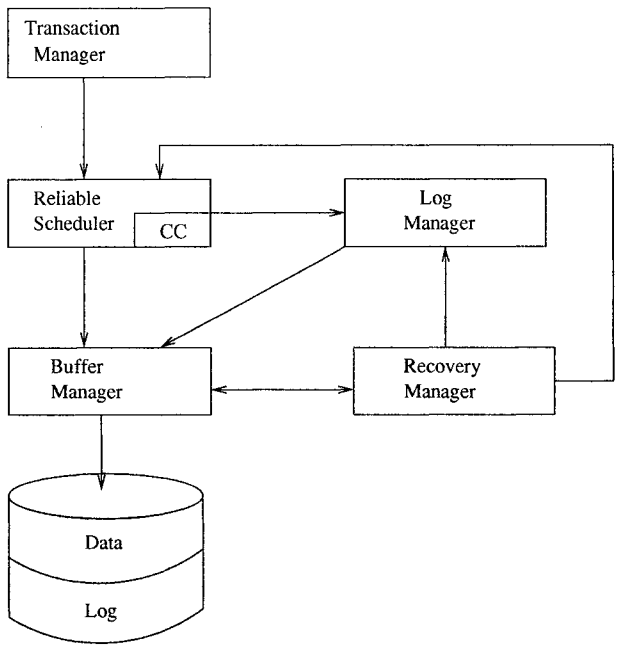

Figure 10: The Architecture of Our Execution Model

Recovery Manager: The recovery manager is activated in the event of a system crash. It communicates with the buffer manager, examines the log and performs the appropriate recovery actions. The recovery manager writes log records for subtransactions whose execution was interrupted because of the crash. The incomplete advanced transactions are forwarded to the scheduler for re-execution.

Log Manager: The log manager maintains the log. It receives updates from the different components. It makes sure that the log is copied to the disk at certain times.

Buffer Manager: Access to the disk takes place through the buffer manager. It captures data and log from disk, or sends data and log to disk. It responds to the requests from the scheduler, recovery manager and log manager.

When an user or application submits an advanced transaction to the transaction manager. The transaction manager issues the start of the advanced transaction and sends it to the scheduler. Our scheduler schedules a subtransaction to execute. The concurrency control will take care of the read and write operations in the subtransaction. At the same time, the corresponding log records should be recorded by log manager. When the advanced transaction completes its execution, the result will

be sent back by the transaction manager.

When a system crash occurs, the recovery manager is activated. The recovery manager requests the log from buffer manager, it then scans the log to find all interrupted subtransactions and restores the consistency. The interrupted subtransactions are identified and submitted to the scheduler for re-execution. The details of reliable recovery will be discussed in next chapter.

5.2 *Execution Model*

One subtransaction can be at different states during its lifetime. The transitions among these states are controlled by primitives.

Definition 25

[State of Subtransaction in Scheduling] A subtransaction T_{ij} can be in any of the following *states*: *unscheduled* (un_{ij}), *initiation* (in_{ij}), *execution* (ex_{ij}), *prepare* (pr_{ij}) (means prepare to commit), *committed* (cm_{ij}) and *aborted* (ab_{ij}). Execution of a primitive causes a subtransaction to change its state.

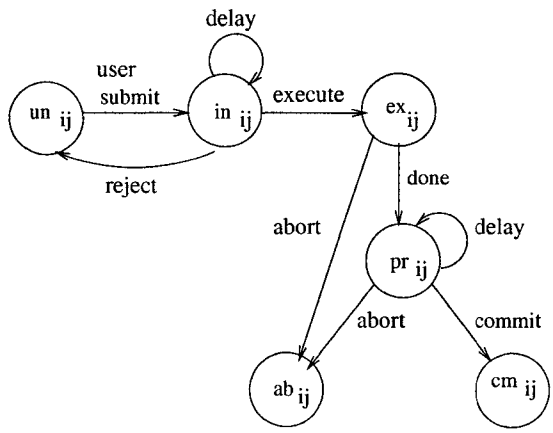

Figure 11: States of Subtransaction T_{ij}

We next discuss what are these states and how the state transitions take place. Detailed state transition diagrams are shown in figure 5.2.

- **unscheduled** (un_{ij}): A subtransaction (T_{ij}) is in an unscheduled state when it has not been sent to the scheduler. At this point, the scheduler can do nothing about it.

- **initiation** (in_{ij}): When a subtransaction (T_{ij}) is sent to the scheduler, its state changes from unscheduled to initiation. In this state, the subtransaction is waiting to be executed. Later the scheduler will execute, delay or reject this subtransaction.

- **execution** (ex_{ij}): a subtransaction (T_{ij}) moves from initiation state to execution state by executing the begin primitive. When a subtransaction is in the execution state, the only way a scheduler can control it is by aborting the subtransaction.

- **prepare** (pr_{ij}): after a subtransaction (T_{ij}) finishes its execution and is ready to commit, it is in the prepare state. At this point, a scheduler can determine whether the subtransaction should commit or abort.

- **committed** (cm_{ij}): a subtransaction (T_{ij}) is in committed state after it has executed the commit primitive.

- **aborted** (ab_{ij}): a subtransaction (T_{ij}) is in the aborted state when it has executed the abort primitive and has been aborted. Note that, there are two ways to enter the aborted state – a subtransaction can be in the execution state or prepare state when it gets aborted.

The aborted state and the committed states are called the *final states*. When a subtransaction has reached the final state, the scheduler can no longer change its state.

Next we formally define the complete execution criteria of an advanced transaction. Here we assume that the specification C in $AT_w =< S, D, C >$ includes all the possible completion sets - both successful completion and unsuccessful completion. The execution outcome (the set of committed subtransactions) is expected to be matched with as least one completion set specified in C. We say an advanced transaction is completed if all subtransactions in any one completion set are committed, and all the other subtransactions are either in final states or unscheduled states. The formal definition appears below.

Definition 26

[Complete Execution] The execution of an advanced transaction is said to be *complete* if the following conditions hold: all subtransactions in any one complete set are committed, and all the other subtransactions of this advanced transaction are either in final states or unscheduled state. Or, in

formal manner, $\exists\, C_k \in C \bullet ((\forall\, T_{ij} \in C_k \bullet cm_{ij} = true) \wedge (\forall\, T_{im} \in S - C_k \bullet cm_{im} = true \vee ab_{im} = true \vee un_{im} = true))$.

A subtransaction that is never terminated but remains active after the completion of an advanced transaction is called an *orphan subtransaction.* A reliable scheduler must be able to schedule the subtransactions in correct order, it should not violate any dependency constraints during execution, and should not leave any orphan subtransactions after execution completion.

Definition 27

[Reliable Scheduling] The scheduling of an advanced transaction is *reliable* if it satisfies the following conditions.

1. all dependency constraints of the advanced transaction are satisfied
2. when execution of an advanced transaction completes, every subtransaction of the advanced transaction must be in a final state (a committed/aborted state) or in an unscheduled state.

The above requirements are necessary for a reliable scheduler to correctly execute an advanced transaction and to avoid availability and integrity problems. We have already discussed how violating the dependency constraints in an advanced transactions result in integrity and availability problems. The last condition further ensures that there are no orphan subtransactions left after execution, since orphan subtransactions hold resources and cause availability problems.

5.3 Data Structures Required by the Scheduler

Before giving the details of the algorithm, we describe the data structures needed by our scheduler.

5.3.1 Scheduling Action Table for Primitive Dependencies

The actions to be taken by the scheduler in order to correctly enforce a dependency of the form $T_{ij} \rightarrow_x T_{ik}$ depends on the type of dependency existing between T_{ij} and T_{ik} and the states of T_{ij} and T_{ik}. This information is stored in the *scheduling action table.* For each dependency of the form $T_{ij} \rightarrow_x T_{ik}$, we construct a scheduling action table TB_x. This table has five rows and five columns corresponding to the initiation, execution, prepare, abort, and commit states of T_{ij} and T_{ik} respectively. There is no entry corresponding to the state unscheduled. This is because the scheduler

cannot take any action until the user has submitted the subtransaction – when the user submits the subtransaction, it's state is changed to initiation. An entry in this table is denoted as $EN_x(i,j)$ where i represents a state of the subtransaction (T_{ij}), and j represents a state of the subtransaction (T_{ik}). The entry $EN_x(i,j)$ can have the following values:

1. *no restriction*, shown as '–' in the table, means that the scheduler need not impose any restriction for the state transitions of the two subtransactions T_{ij} and T_{ik}. The subtransactions T_{ij} or T_{ik} can go into the next state without any restriction.

2. *delay* $T_{ij}(T_{ik})$, means that the subtransaction $T_{ij}(T_{ik})$ cannot make a state transition at this stage. It must wait at the current state until the other subtransaction $T_{ik}(T_{ij})$ has entered another state.

3. *execute* $T_{ij}(T_{ik})$, means that the subtransaction $T_{ij}(T_{ik})$ must be executed.

4. *abort* $T_{ij}(T_{ik})$, means that the subtransaction $T_{ij}(T_{ik})$ must be aborted.

5. *reject* $T_{ij}(T_{ik})$, means that the subtransaction $T_{ij}(T_{ik})$ will be rejected instead of being scheduled for execution. This entry is only possible when subtransaction $T_{ij}(T_{ik})$ is in its initiation state.

6. *prohibited*, shown as '/' in the table, means it is not possible for the subtransactions to be in the corresponding states simultaneously because of this dependency.

7. *final states*, shown as 'final' in the table, means both the subtransactions are in the final state. No further state transitions are possible.

The dependency scheduling tables specify the necessary actions that must be taken by the scheduler to ensure the satisfaction of all the dependencies. In this section, due to space limit, we only list the dependency scheduling action tables for *strong-commit*, *commit*, and *begin-on-commit* dependencies. A whole listing of all scheduling action tables is shown in appendix of this dissertation.

1. $[T_{ij} \rightarrow_{sc} T_{ik}]$, the scheduler needs to ensure the commitment of both subtransactions. The scheduling action table TB_{sc} is shown in Table 21.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	–	–	reject T_{ij}
ex_{ij}	–	–	–	–	abort T_{ij}
pr_{ij}	delay T_{ij}	delay T_{ij}	delay T_{ij}	–	abort T_{ij}
cm_{ij}	/	/	/	final	/
ab_{ij}	–	–	–	final	final

Table 8: Scheduling Action Table for *strong commit* Dependency

The first row of the table specifies the actions to be taken when T_{ij} is in the initiation state. In this row all the entries are marked with '–' indicating that the scheduler does not impose any constraint on T_{ij} or T_{ik} changing states. The entry in the second row, last column (that is, $EN(ex_{ij}, ab_{ik})$) is an 'abort T_{ij}'. This means that when T_{ij} is in the execution state, and T_{ik} is aborted, then T_{ij} must be aborted as well. The entry in the third row, first column (that is, $EN(pr_{ij}, un_{ik})$) is 'delay T_{ij}'. This means that when T_{ij} is in the prepare state and T_{ik} is unscheduled, T_{ij} must wait in the prepare stage. The entry in the fourth row, first column (that is, $EN(cm_{ij}, un_{ik})$) is '/'. This means that the scheduler will not allow this to happen. The entry in the fourth row, fourth column (that is, $EN(cm_{ij}, cm_{ik})$) is 'final'. This means that both the transactions have reached their final states, and the scheduler need not do anything more.

2. $[T_{ij} \rightarrow_c T_{ik}]$, the scheduler needs to ensure the order of two subtransactions' commitment. The scheduling action table TB_c is shown in Table 22.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	delay T_{ik}	/	–
ex_{ij}	–	–	delay T_{ik}	/	–
pr_{ij}	–	–	delay T_{ik}	/	–
cm_{ij}	–	–	–	final	final
ab_{ij}	–	–	–	final	final

Table 9: Scheduling Action Table for *commit* Dependency

3. $[T_{ij} \rightarrow_{bc} T_{ik}]$, the scheduler needs to ensure that the beginning of subtransaction T_{ik} must be after the commitment of subtransaction T_{ij}. The scheduling action table TB_{bc} is shown in Table 29.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	delay T_{ik}	/	/	/	/
ex_{ij}	delay T_{ik}	/	/	/	/
pr_{ij}	delay T_{ik}	/	/	/	/
cm_{ij}	–	–	–	final	final
ab_{ij}	reject T_{ik}	/	/	/	/

Table 10: Scheduling Action Table for *begin-on-commit* Dependency

The following ensures the correctness of our scheduling action tables.

Lemma 1

The scheduler by taking the actions listed in the scheduling action tables for the primitive dependencies can enforce the dependencies correctly.

The proof of this and all other theorems are given in the correctness proof section following.

5.3.2 Scheduling Action Table for Composite Dependencies

Based on the dependency scheduling action tables for all primitive dependencies, we propose an algorithm to create scheduling action tables for composite dependencies. These tables are called the *composite scheduling action tables*. The composite scheduling action table for the composite dependency consisting of primitive dependencies x, y, and z is denoted by $TBC_{x,y,z}$. Note that, this table can be created statically for all the composite dependencies existing in advanced transactions.

In determining the proper actions for a composite dependency, we need to combine the entries from two or more scheduling action tables of the component primitive dependencies. To obtain the correct action from these different entry items, we need to define the *priority* for each type of scheduling actions in the primitive scheduling action table. The different actions are prioritized in the following order: "prohibited" , "reject" , "abort" , "delay" , "execute", and "no restriction", where "prohibited" signifies the highest priority and "no restriction" signifies the least priority. We use the notation $>$ to describe the priority ordering. For instance, "prohibited $>$ reject" means that "prohibited" has a higher priority than "reject". In combining the actions of two or more primitive dependency tables, the scheduler will choose the table entry with the highest priority, and set this entry as the action for the composite dependency. For example, when a scheduler finds "no restriction" in one execution table and a "delay" entry in other scheduling table, it will take the

"delay" entry as the action for the composite dependency.

We next give an algorithm to combine the scheduling tables and determine the correct actions for the composite dependency. To combine the scheduling tables of two (or more) primitive dependencies, we compare the corresponding table entries and choose the action that satisfies the constraints of all component dependencies.

Algorithm 12

Creating Composite Scheduling Action Table

Input: (i) $T_{ij} \rightarrow_{d_1,d_2,...,d_n} T_{ik}$ – the composite dependency composed of the primitive dependencies $d_1, d_2, ..., ... d_n$ and (ii) **TB** = $\{TB_{d_1}, TB_{d_2}, \ldots, TB_{d_n}\}$ – the scheduling action tables for the primitive dependencies

Output: Scheduling action table TBC for this composite dependency

Procedure $CreateCompositeSchedulingTable(T_{ij} \rightarrow_{d_1,d_2,...,d_n} T_{ik}, \mathbf{TB})$

begin

for each state (S_{ij}) of subtransaction (T_{ij}) $\in \{un_{ij}, in_{ij}, ex_{ij}, pr_{ij}, ab_{ij}, cm_{ij}\}$

for each state (S_{ik}) of subtransaction (T_{ik}) $\in \{un_{ik}, in_{ik}, ex_{ik}, pr_{ik}, ab_{ik}, cm_{ik}\}$

begin

/* initialization */

$EN_{TBC}(S_{ij}, S_{ik})$ = "–"

set max_p = "–"

/* get every component dependency's scheduling table entry */

for every primitive dependency d_k in this composite dependency

begin

access the scheduling action table TB_k for this dependency d_k

get the corresponding entry $EN_k(S_{ij}, S_{ik})$

/* finding the highest priority entry in these dependencies */

if $EN_k(S_{ij}, S_{ik}) > max_p$

$max_p = EN_k(S_{ij}, S_{ik})$

end for

$EN_{TBC}(S_{ij}, S_{ik}) = max_p$

end for

end

We next show that, with the priority assignment, the above algorithm could be able to ensure the satisfaction of all primitive dependencies in a composite dependency. This is formalized in the following theorem.

Lemma 2

The scheduler can enforce composite dependencies correctly.

5.3.3 State Table

The scheduler during the execution of advanced transactions maintains some dynamic data structures called *state tables*. A state table is created for each advanced transaction that has been submitted by the user. The state table records the execution states of the subtransactions in an advanced transaction while it is being executed. Whenever a subtransaction of this advanced transaction changes state, the corresponding entry in the state table is updated. When the advanced transaction terminates, the state table is deleted.

5.3.4 Job Queue

The *job queue* is another dynamic data structure that is needed by the scheduler. The job queue holds subtransactions that have been submitted by the user but which are not being currently executed. The jobs submitted by a user is initially placed in the job queue. Also, when a subtransaction needs to wait before being processed further, it is placed in the job queue. In other words, subtransactions in the initiation state or prepare state are placed in this job queue. When the subtransaction in the initiation (prepare) state is ready to execute (commit), it is removed from this queue. When the subtransaction is in the initiation state and it is rejected by the scheduler, it is removed from the job queue and it's state changes to unscheduled.

5.4 *Reliable Scheduling of Advanced Transactions*

Before describing how advanced transactions are scheduled, we state our assumptions. We assume that the specifications of all generalized advanced transactions that will be executed by our system are correct. This translates into two assumptions. First, there are no dependency conflicts in the specification of a generalized advanced transaction. Second, in any normal execution it will be possible to satisfy at least one completion set of the advanced transaction. In a separate work [118], we discuss how to statically analyze the specification of a generalized advanced transaction for correctness.

The execution of an advanced transaction is organized in three stages: (i) Preparation Stage, (ii) Execution Stage, and (iii) Termination Stage. Scheduling algorithms for these stages are described in the following subsections.

5.4.1 Preparation Stage

In this stage, the user submits the advanced transaction for execution. After receiving the input from the user, a state table is created for this advanced transaction. The entries for each subtransaction in this state table is initialized to *initiation*. The subtransactions are placed in the job queue for later execution. When the user has completed submitting subtransactions for the advanced transaction, the advanced transaction moves into the execution stage. The following algorithm summarizes the work done in the preparation stage.

Algorithm 13

InputAdvancedTransaction

Input: (i) $AT_i =< S, D, C >$ – the advanced transaction.

Procedure $InputAdvancedTransaction(AT_i)$

begin

receive the input $AT_i =< S, D, C >$

create $StateTable_i$

for each $T_{ij} \in S$

begin

$StateTable_i[T_{ij}] = initiation$ /* set initial states for subtransactions */

enQueue(JobQueue, T_{ij}) /* insert in the job queue */

end for

end

5.4.2 Execution Stage

In this stage, the subtransactions submitted by the user get executed. When the scheduler gets a subtransaction, it first looks into the advanced transaction specification to find out all dependencies associated with it. For each dependency, the scheduler finds a corresponding dependency table, and the states of the two involved subtransactions. The scheduler then accesses the dependency scheduling action table, and gets a possible action for the subtransaction. The action can be one of the following: allow it to commit/abort, send the transaction to execute, hold the subtransaction in waiting, and reject the transaction at the initiation state. If the action causes the subtransaction to change state, the state table entry corresponding to this subtransaction may need to be modified. The following algorithm formalizes the actions taken in this stage.

Algorithm 14

Execution Stage

Input: (i) $AT_i =< S, D, C >$ – the advanced transaction that must be executed and (ii) **TB** – the set of primitive and composite scheduling action tables associated with the dependencies of the advanced transaction AT_i.

Procedure $ExecuteAdvancedTransaction(AT_i, \mathbf{TB})$

begin

while(TRUE)

begin

T_{ij} = deQueue(JobQueue) /* get the job from the job queue */

$Action = getAction(T_{ij}, AT_i, \mathbf{TB})$

if $Action = wait$

enQueue(JobQueue, T_{ij}) /* insert in queue */

else if $Action = abort$

abort T_{ij}

$StateTable_i[T_{ij}] = aborted$

else if $Action = reject$

$StateTable_i[T_{ij}] = unscheduled$

else if $Action$ = '–' /* no restriction for T_{ij} */

begin

if $StateTable_i[T_{ij}] = initiation$

send (T_{ij}) to execute /* execute the operations for T_{ij} */

$StateTable_i[T_{ij}] = executing$

else if $StateTable_i[T_{ij}] = executing$

get execution results

if execution result is *completed* /* operations completed */

$StateTable_i[T_{ij}] = prepare$

enQueue(JobQueue, T_{ij})

else if execution failed /* operations failed */

$StateTable_i[T_{ij}] = aborted$

else if $StateTable_i[T_{ij}] = prepare$

commit T_{ij}

$StateTable_i[T_{ij}] = committed$

end

end

end

The above algorithm makes a call *getAction* to get the action that must be taken by the scheduler. We next describe the algorithm *getAction* that describes how the scheduler determines an action for scheduling a submitted subtransaction (T_{ij}), focusing on ensuring the dependency constrains associated with T_{ij}. We assume that the primitive and composite dependency tables that will be needed by this advanced transaction have already been created.

Algorithm 15

Get Action From ActionTables

Input: (i) T_{ij} – the subtransaction for which the action must be determined, (ii) $AT_i =< S, D, C >$ – the advanced transaction whose subtransaction is T_{ij}, and (iii) **TB** – the set of primitive and composite scheduling action tables associated with the dependencies of the advanced transaction AT_i.

Output: The action the scheduler should take for subtransaction T_{ij} at the current state

Procedure $getAction(T_{ij}, AT_i, \mathbf{TB})$

begin

$ACTION$ = '–' /* Initialize ACTION */

/* find out all the dependencies associated with T_{ij} */

for every dependency $T_m \rightarrow_d T_n \in D$

begin

if $(T_{ij} \neq T_m)$ **AND** $(T_{ij} \neq T_n)$

skip this round, and continue to next round

else /* this dependency is associated with T_{ij} */

begin

if $T_{ij} = T_n$ /* d is a dependency pointed to T_{ij} */

/* get the state of the subtransactions */

let $S_y = StateTable_i[T_{ij}]$

let $S_x = StateTable_i[T_m]$

else if $T_{ij} = T_m$ /* d is a dependency that T_{ij} lead out */

/* get the state of the subtransactions */

let $S_x = StateTable_i[T_{ij}]$

let $S_y = StateTable_i[T_n]$

access the corresponding dependency scheduling table TB_d

locate the corresponding entry $EN_d(x, y)$ according to the states

if $EN_d(x, y) > ACTION$ /* check the priority */

$ACTION = EN_d(x, y)$;

end

end for

return $ACTION$;

end

The *getAction* algorithm is also with priority assignment to provide the correct action for the scheduler. We next present the proof of the correctness of this algorithm.

Lemma 3

The scheduler can take the correct actions to schedule an advanced transaction.

5.4.3 Termination Stage

For an advanced transaction to complete, the set of committed subtransactions should match at least one of the completion sets specified with this advanced transaction. Therefore, in this stage, we will check whether this condition is satisfied. Using the state tables we find out the set of subtransactions that are in the execution and prepare states. If these sets are not empty, then the advanced transaction is still active and it cannot terminate at this point. Otherwise, the advanced transaction is allowed to terminate. In such a case, we check whether the set of committed transactions correspond to one of the completion sets specified in the advanced transaction. If anyone of the completion set is matched, the advanced transaction is allowed to complete, the subtransactions that are in the initiation state are removed from the job queue, and the scheduler will return a "successful completion" message. Otherwise a "not complete" message is returned.

Once the execution of an advanced transaction is complete, the state table corresponding to the advanced transaction is deleted.

Algorithm 16

Termination Stage

Input: (i) $AT_i =< S, D, C >$ – the advanced transaction whose termination is being determined.

Output: (i) result indicating whether the advanced transaction terminated successfully or not.

Procedure $TerminateAdvancedTransaction(AT_i)$

begin

$executing = prepared = committed = initiation = \{\}$

for each subtransaction $T_{ij} \in S$

begin

if $StateTable_i[T_{ij}] = initiation$

$initiation = initiation \cup \{T_{ij}\}$

if $StateTable_i[T_{ij}] = committed$

$committed = committed \cup \{T_{ij}\}$

else if $StateTable_i[T_{ij}] = executing$

$executing = executing \cup \{T_{ij}\}$

else if $StateTable_i[T_{ij}] = prepared$

$prepared = prepared \cup \{T_{ij}\}$

end for

/* check whether there are active subtransactions for AT_i */

if $prepared \neq \{\}$ OR $executing \neq \{\}$

return 'not complete'

else begin /* no active subtransactions currently */

for each $C_{ij} \in C$

begin /*check if one completion set is matched */

if $C_{ij} \subseteq committed$

begin

for all $T_{ij} \in initiation$

remove(JobQueue, T_{ij})

Delete $StateTable_i$

return 'terminated successfully'

end if

end for

return 'not complete'

end

end

5.5 *Correctness Proof*

The following lemma and theorem prove the correctness of the mechanisms.

Lemma 1

The scheduler by taking the actions listed in the scheduling action tables for the primitive dependencies can enforce the dependencies correctly.

Proof 1

Here we show the correctness proof for just one dependency: the strong commit dependency. The scheduler can enforce the strong commit dependency by following the scheduling table (TB_{sc}). For a strong commit dependency $T_{ij} \rightarrow_{sc} T_{ik}$, all the incorrect execution results are:

1. *Incorrect Case* IR_1: $< abort\ T_{ik} >$ appears before $< commit\ T_{ij} >$ in history;
2. *Incorrect Case* IR_2: $< commit\ T_{ij} >$ appears before $< abort\ T_{ik} >$ in history;
3. *Incorrect Case* IR_3: only $< commit\ T_{ij} >$ appears in history, T_{ik} is not executed.

We show that these three situations above will not be reached if the scheduler follows the scheduling action table TB_{sc}.

For [Incorrect Case IR_1:] $< abort\ T_{ik} >$ before $< commit\ T_{ij} >$ in a history means that when T_{ik} enters the aborted state (ab_{ik}), T_{ij} can be in one of these states: initiation state (in_{ij}), execution state (ex_{ij}), prepare state (pr_{ij}).

- if T_{ij} is in state in_{ij}, then as per Table 21 the scheduler will reject T_{ij}. In such a case the history will have only $< abort\ T_{ik} >$.
- if T_{ij} is in state ex_{ij} or state pr_{ij}, then as per Table 21 the scheduler will abort T_{ij}. The execution result will be $< abort\ T_{ik} >$ followed by $< abort\ T_{ij} >$ in a history.

Observing from analysis above, it is impossible to have $< abort\ T_{ik} >$ before $< commit\ T_{ij} >$ in a history.

For [Incorrect Case IR_2:] From the Table 21 note that T_{ij} needs to wait in the prepare state and cannot commit until T_{ik} enters the committed state (cm_{ij}). Thus, $< commit\ T_{ij} >$ before $< abort\ T_{ik} >$ in history is not possible.

For [Incorrect Case IR_3:] Since T_{ij} should wait on the prepare state (pr_{ij}) until T_{ik} is in the committed state, T_{ij} cannot enter the committed state without the execution of T_{ik}. Thus, only $< commit\ T_{ij} >$ in the history is impossible.

We can see from above that these incorrect cases will not appear with the proposed scheduling action table for strong commit dependency. Therefore, the scheduling table (TB_{sc}) enforces the strong commit dependency. Similarly, we can prove that all other dependencies will be satisfied by the corresponding scheduling action tables.

Lemma 2

The scheduler can enforce composite dependencies correctly.

Proof 2

To prove this, we need to show that the algorithm for generating the composite dependency action table is correct. That is, each entry in this table lists the actions that are needed to enforce the composite dependency. We show the entries of combining two dependencies following this algorithm, and prove that the entry satisfies the composite dependency. The algorithm compares the entries of the two action tables and chooses the action with the higher priority. In the following, we show that this indeed is correct.

Combining "prohibited" action with any action This entry has the highest priority. If any other type of table entry is combined with "prohibited", the result is "prohibited". If there exists a "prohibited" entry EN_x for a dependency $T_x \rightarrow_{d_1} T_y$, it means that the corresponding states of the two subtransactions are not possible by our algorithm. When we combine this dependency with another dependency $T_x \rightarrow d_2\, T_y$, these two subtransactions are still not allowed to enter these states. Therefore, if an entry is "prohibited" for one primitive dependency, any composite dependency having this dependency will also have a "prohibited" entry.

Combining "no restriction" action with any action The lowest priority of the entries is "no restriction". A "no restriction" entry means the two subtransactions can go to next states without any restriction. Thus, if a "no restriction" entry is combined with any other entry, the other entry has precedence.

Combining "abort" and "delay" This situation will never be encountered. If the abort action appears in a scheduling action table, one of the subtransactions must be in its final state. If the delay action appears, then none of the subtransactions can be in the final state. Thus, no situations will require combining "abort" and "delay".

Combining "reject" and "abort" The "reject" and "abort" action will not appear together. This is because the rejection action can only appear in the row or column corresponding to the initiation state. Such a row or column cannot have an "abort" entry.

Combining "execute" and "abort" The "execute" and "abort" action will not appear together. This is because if the execute action appears in one dependency scheduling table, then this subtransaction must be in its initiation state and a subtransaction in initiation state cannot have an "abort" action entry.

Combining "delay" and "reject" The "delay" and "reject" actions will not appear together. The reject action appears when one subtransaction has reached its final stage. The delay entry appears only when none of the subtransactions have reached the final stage.

Combining "execute" and "reject"("delay") The "execute" and "reject" ("delay") action will appear together only if there is a conflict of dependencies. In our definition of an advanced transaction, we require that none of the dependencies in an advanced transaction conflict with each other.

Thus, when two dependencies are combined, the action of the composite dependency satisfies the constraints of the two dependencies. The process can be repeated for the case where more than two dependencies are combined. Thus our composition algorithm always ensures the satisfaction of the dependencies.

Lemma 3

The scheduler can take the correct actions to schedule an advanced transaction.

Proof 3

The action taken by the scheduler is the one that is returned by the procedure *GetAction*. The *GetAction* function gets a subtransaction of an advanced transaction as input. It searches all the scheduling action tables related to the dependencies associated the subtransaction. It finds the highest priority action and returns to the scheduler. So we need to prove the *GetAction* function finds the correct action for a subtransaction. Since a pair of subtransactions will never be in a state that has a *prohibited* entry in the scheduling action table, we show the comparison among all the other entries in the scheduling action tables for the pair of subtransactions.

Between "no restriction" and any other action "no restriction" is the lowest priority of the entries. It means the subtransaction can go to next state without any restriction. Thus, any higher priority action is picked by the GetAction function won't violate the dependencies associated with the subtransaction.

Between "delay" and "abort""delay" means that the subtransaction should wait in the current state, "abort" means that the subtransaction should be aborted. The priority of "abort" is higher than "delay". The "delay" action only occurs at initiation and prepare states; the "abort" action only occurs after the initiation state. So the subtransaction must be in prepare state. This indicates that the subtransaction need to wait to commit/abort based on the other subtransaction execution result. Going through all the scheduling action tables we have, we find out there is no requirement which request a subtransaction to commit after waiting in prepare state. So choosing "abort" action, will not effect the other dependency with requiring "delay" action.

Between "reject" and "delay" "reject" means that the subtransaction should not be executed. "delay" means that the subtransaction should wait in the current state. The priority of "reject" is higher than "delay". the "reject" action only occurs in initiation state; The "delay" action only occurs at initiation and prepare states. So the subtransaction must be in initiation state. This indicates that the subtransaction need to wait to execute/be reject based on the other sub-transaction. After we check all the scheduling action tables, we find out there is only "reject" action requirement after the subtransaction wait in initiation state. So choosing "reject" action will not effect the other dependency with requiring "delay" action.

Between "reject" and "execute" "reject" means that the subtransaction should not be executed. "execute" means that the subtransaction must begin execution. Theses indicate the conflict dependencies. We won't have such situation.

Between "reject" and "abort" Based on our scheduling action tables, "reject" only occurs in the initiation state; "abort" only occurs after the initiation state. So we won't have such situation.

Between "execute" and "delay" "execute" means that the subtransaction must begin execution.

"delay" means that the subtransaction should wait in the current state. These indicate the conflict dependencies. We won't have such situation.

Between "execute" and "abort" Based on our scheduling action tables, "execute" only occurs in the initiation state; "abort" only occurs after the initiation state. SO we won't have such situation.

Thus, the highest priority action the GetAction function finds is the correct action to be returned to the scheduler. So we prove that the scheduler take the correct action to schedule a subtransaction.

Theorem 11

The proposed scheduling algorithm ensures reliable scheduling as per Definition 27.

Proof 11

To prove this, we need to show that (1) all dependency constraints of the advanced transaction are satisfied, and (2) when a subtransaction completes, there are no orphan transactions. Lemmas 1, 2, and 3 ensure that the actions taken by the scheduler do not violate the dependency constraints. We need to prove that at the completion of the advanced transaction all subtransactions are either in the unscheduled or final states. Note that in the termination stage, the advanced transaction is allowed to complete only if no subtransactions of this advanced transaction are executing or are in the prepare state. However, one or more subtransactions may be in the initiation state. These subtransactions are removed from the JobQueue before the transaction is allowed to terminate. This ensures that there are no orphan subtransactions.

CHAPTER VI

FAILURE RECOVERY FOR ADVANCED TRANSACTION

In order to coordinate the execution of the various subtransactions in an advanced transaction, dependencies are specified among them. These dependencies are enforced during the normal execution. However, in face of a system crash or failure in execution, the system may be left in an inconsistent state – some subtransactions in one advanced transaction may be committed, some may be partially executed and others unscheduled, and the dependencies among these subtransactions may not be enforced correctly due to interruption of execution. In such a situation, proper recovery actions must be taken to react to the failure. Although researchers have worked on the problem of advanced transaction recovery [20, 26, 33, 38, 60, 62, 63], most of these work focus on restoring consistency by removing the effects of partially executed subtransactions, they fail to address how to ensure dependencies during advanced transaction recovery.

System crash or failures might occur during the execution of an advanced transaction. The recovery manager needs to take appropriate actions to react to such failures – restore the system back to consistent states and ensure the satisfaction of all dependencies. Let us illustrate the problems that will occur if dependencies are not enforced during recovery. Consider an advanced transaction of two subtransactions T_{w5} (reserving a room in a resort) and T_{w6} (reserving a rental car), and there is a *force-begin-on-begin* dependency which requires that if T_{w5} begins then T_{w6} must also begin. The scheduler enforces this dependency by starting T_{w6} after T_{w5} begins. Suppose T_{w6} gets completed before T_{w5} and a crash occurs before T_{w5} completes. Since T_{w6} is complete and T_{w5} is incomplete, the traditional recovery mechanism undoes T_{w5} and redoes T_{w6} to restore data consistency. This results in an undesirable situation. Subsequently, if the scheduler re-executes T_{w5}, then T_{w6} will also be re-executed resulting in multiple rental car reservations. On the other hand, if T_{w5} is not re-executed, then T_{w6} remains committed, which is also undesirable – we do not want the partial outcome of T_{w6} alone. Thus, restoring consistency by removing the effects of interrupted subtransactions is not enough for advanced transaction. The recovery algorithm must also take into

consideration of the effect of dependencies and ensure their satisfaction during recovery.

In this chapter, we investigate the issues pertaining to recovery. When a system crash occurs, subtransactions may be at different stages of execution and dependencies may not be preserved. To restore system consistency, the subtransactions whose execution was interrupted by the crash must be undone. Undoing a subtransaction may impact other subtransactions that are related by dependencies. In such cases, we identify the recovery actions needed for these other subtransactions. Thus, our recovery mechanism not only preserves database consistency but also satisfies the given dependencies. Finally, we formalize what we mean by *reliable recovery* and prove that our mechanism does indeed satisfy this definition. Our recovery manager is very general. It is based on the generalized advanced transactions as defined in Chapter 3. The mechanism can be used for processing many advanced transactions who can be decomposed into subtransactions that are co-ordinated through dependencies.

6.1 Reliable Recovery

During normal execution of advanced transaction, the scheduler will guarantee that all dependencies are enforced. However, in the event of a system crash, advanced transactions may be terminated in an abnormal way. Our recovery mechanism must not only restore database consistency but also ensure the satisfaction of all dependencies. The following definition formalizes our idea of reliable recovery.

Definition 28

[Reliable Recovery] The recovery algorithm is *reliable* if it satisfies the following conditions on its completion:

1. the database should be in a consistent state,
2. dependencies associated with every advanced transaction are satisfied, and
3. all interrupted advanced transactions have completed execution.

When a system crash occurs, subtransactions may be at different states in execution (please refer to the execution states described in section 5.1). A subtransaction can be in any one of the following

states: committed, aborted, execution, prepared, and initiation. Here we do not have unscheduled, because all subtransactions that the user submits are initialized to initiation state in the state table.

Definition 29

[Interrupted Subtransaction] *Interrupted subtransactions* are those subtransactions who were in *execution* or *prepare* state when the system crash occurs. Subtransactions in final states (committed or aborted) and in *initiation* states are not considered as interrupted since they are not active in execution at the point of system crash.

6.1.1 Log Records used by Recovery Manager

In a traditional transaction processing system, the log records that are stored in stable storage contain all the information that is needed to perform the recovery. Advanced transaction recovery is much more complex than the recovery in traditional transaction processing system. In order to recover from a system failure, the state information of advanced transactions and subtransactions need to be logged onto some stable storage. We propose that such information should be stored in the system log. Execution of an advanced transaction primitive, a subtransaction primitive, or a subtransaction operation results in the insertion of log records. Moreover, the logs also contain records associated with check-pointing. These are explained below.

Execution of a begin primitive in an advanced transaction results in the insertion of the following log records. $< START\ AT_t\ S >$ where AT_t indicates the advanced transaction id, and S indicates the set of all submitted subtransactions with its inputs in the advanced transaction. For instance, S could be like $(T_{ti}, \{v\}),(T_{tj}, \{\}),(T_{tk}, \{w,y\})$. This information is used by the recovery manager to rebuild the state table for an incomplete advanced transaction. The input of the advanced transaction is used by scheduler to re-execute an incomplete advanced transaction. The completion of the execution of an advanced transaction is indicated by a log record $< COMPLETE\ AT_t >$.

Execution of the begin primitive for a subtransaction T_{ti} results in the following record being inserted in the log: $< START\ T_{ti} >$ where T_{ti} is the subtransaction id. Note that the subtransaction id includes information about the advanced transaction id of which the subtransaction is a part. Similarly, execution of the primitives commit or abort results in the following log records for subtransaction T_{ti}: $< COMMIT\ T_{ti} >$ or $< ABORT\ T_{ti} >$. Same as the undo redo log in traditional

transaction, the log should be flushed immediately to disk after these two records. Execution of a write operation by transaction T_{ti} on data item X which causes the value of X to change from v to w results in the insertion of the following log record: $< T_{ti}\ X, v, w >$. Note that according to the rules of undo-redo algorithm, this log record must be written on the disk prior to the value of X being changed on the disk.

Due to recovery purpose, we also propose a new log record $< RECOVERY\ ABORT\ T_{ti} >$. This record is only written to the log during recovery process. It indicates that the subtransaction T_{ti} is an interrupted subtransaction, which is undone during recovery process. After this record is written, it must be flushed to disk. Because when a system failure occurs again during the recovery processing, this record is used to distinguish the interrupted subtransactions with the subtransactions aborted during normal execution. This is because special actions need to be taken for interrupted subtransactions.

In order to minimize the amount of log that is scanned during recovery, we use checkpointing. Two log records are associated with checkpointing operation. One is the beginning of checkpointing denoted as $< START\ CKPT\ S >$, where the set S is the set of the active subtransactions when the checkpointing begins. This record is flushed to the disk after it is written. This is followed by writing to disk all buffers that contain changed value of data items modified by subtransactions. When all such dirty buffers have been copied to the disk, the record $< END\ CKPT >$ is written which denotes the end of checkpointing. It indicates that all operations before $< START\ CKPT\ S >$ are stored in the stable storage.

6.1.2 Recovery Algorithms for Advanced Transactions

We now propose a detailed recovery mechanism which works with our advanced transaction scheduler. The first phase finds out all the interrupted subtransactions, restores the before image to undo them. Then, redo all committed subtransactions to make sure the committed result has been stored in the system in the second phase. The last phase, we need to adjust the states of all subtransactions in incomplete advanced transactions, and submit the subtransactions that need to be re-executed to the scheduler. Based on the advanced transaction starting log, we can create a state table for each incomplete advanced transaction and fill in the corresponding states for all subtransactions

that are submitted for this advanced transaction. The last stage is to submit the state table and the subtransactions to the scheduler for re-execution.

Algorithm 17

Advanced Transaction Recovery Algorithm

Input: the log

Output: a consistent advanced transaction state in which all the dependencies are enforced

Procedure $AdvancedTransactionRecovery(log)$

Initialization:

/* lists keeping track of incomplete advanced transactions and subtransactions in them */

$completeAT = \{\}$ /* set of all complete advanced transactions */

$incompleteAT = \{\}$ /* set of all incomplete advanced transactions */

$globalCommitted = \{\}$ /* set of all committed subtransactions */

$globalAborted = \{\}$ /* set of all aborted subtransactions */

$globalInterrupted = \{\}$ /* set of all interrupted subtransactions */

$endCkptFlag = false$ /* the endCkptFlag set to false */

Phase 1 Undo Phase

begin

/* Scan backwards until the $< START\ CKPT\ M >$ that corresponds to the latest $< END\ CKPT >$ log record is found */

do /* Scan backwards until the $< START\ CKPT >$ */

get last un-scanned log record

case the log record is $< COMMIT\ T_{ti} >$

if $T_{ti} \notin globalAborted$

$globalCommitted = globalCommitted \cup \{T_{ti}\}$

case the log record is $< ABORT\ T_{ti} >$

$globalAborted = globalAborted \cup \{T_{ti}\}$

case the log record is $< RECOVERY\ ABORT\ T_{ti} >$

if $T_{ti} \notin (globalCommitted \cup globalAborted)$

$globalInterrupted = globalInterrupted \cup \{T_{ti}\}$

case the log record is update record $< T_{ti}, x, v, w >$

if $T_{ti} \notin globalCommitted$

change the value of x to v /* restores before image of x */

case the log record is $< START\ T_{ti} >$

if $T_{ti} \notin (globalCommitted \cup globalAborted \cup globalInterrupted)$

$globalInterrupted = globalInterrupted \cup \{T_{ti}\}$

write a $< RECOVERY\ ABORT\ T_{ti} >$ record and flush to the disk

case the log record is $< COMPLETE\ AT_k >$

$completeAT = completeAT \cup \{AT_k\}$

case the log record is $< START\ AT_k\ S >$

if $AT_k \notin completeAT$

$incompleteAT = incompleteAT \cup \{AT_k\}$

case the log record is $< END\ CKPT >$

$endCkptFlag = true$ /* we have found an end ckpt record */

until log record is $< START\ CKPT\ M >$ AND $endCkptFlag = true$

/* Find the interrupted subtransactions and transactions in the set M */

for each subtransaction $T_{ij} \in$ M

begin

if $T_{ij} \notin (globalCommitted \cup globalAborted)$

$globalInterrupted = globalInterrupted \cup T_{ij}$

if $AT_i \notin completeAT \cup incompleteAT$

$incompleteAT = incompleteAT \cup \{AT_k\}$

end for

/* Scan backward to undo the aborted and interrupted subtransactions,

until start of all subtransactions in set M is processed. */

do

case the log record is update record $< T_{kj}, x, v, w >$

if $T_{kj} \notin globalCommitted$ AND $T_{kj} \in M$

change the value of x to v /* restores before image of x */

case the log record is $< START\ T_{kj} >$

if $T_{kj} \in globalInterrupted$

write $< RECOVERY\ ABORT\ T_{kj} >$ record and flush to the disk

until all $< START\ T_{kj} >$ is processed where $T_{kj} \notin globalCommitted$ AND $T_{kj} \in M$

end

Phase 2 Redo Phase

$submitted = \{\}$ /* set of all subtransactions for each incomplete advanced transaction */

$input = \{\}$ /* set of input records for each incomplete advanced transaction */

begin

/* Scan forward from the <START CKPT M> found in Undo Phase to the end of log */

do

case the log record is update record $< T_{ti}, x, v, w >$

if $T_{ti} \in globalCommitted$

change the value of x to w /* restores after image of x */

case the log record is $< START\ AT_t\ S >$

if $AT_t \in incompleteAT$

for each subtransaction T_{ti} in S

$submitted = submitted \cup \{T_{ti}\}$

if there is input with T_{ti}

$input = input \cup (T_{ti},\ v)$

until end of log is reached

end

Phase 3 create state table, submit incomplete advanced transactions to scheduler

begin

for each advanced transaction $AT_t \in incompleteAT$

begin

$toschedule = \{\}$

Get the complete set of the advanced transaction C

create $StateTable_t$ and $Committed_t = \{\}$

for each subtransaction $T_{tx} \in submitted$ set of AT_t

begin

if subtransaction $T_{tx} \in globalInterrupted$

$StateTable_t[T_{tx}] = initiation$

$toschedule = toschedule \cup \{T_{tx}\}$

else if subtransaction $T_{tx} \in globalCommitted$

$StateTable_t[T_{tx}] = committed$

$Committed_t = Committed_t \cup \{T_{tx}\}$

else if subtransaction $T_{tx} \notin globalAborted$

$StateTable_t[T_{tx}] = aborted$

else

$StateTable_t[T_{tx}] = initiation$

$toschedule = toschedule \cup \{T_{tx}\}$

end for

send $StateTable_t$, toschedule set and input set to the scheduler for execution

end for

end

The recovery stage completes after all the incomplete advanced transactions sent to the scheduler have completed execution. When recovery processing stops, all the subtransactions should be in the final state. The interrupted subtransactions also have been aborted, and the incomplete advanced transactions have been submitted to scheduler to execute with other normal advanced transactions. Here, we don't abort the incomplete advanced transactions, we just abort the interrupted subtransactions, and re-build the state tables. An important point should be mentioned here. Our recovery mechanism can be used multiple times. This means that the recovery mechanism can take care the

crash again during the recovery processing. The record $< RECOVERY\ ABORT\ T_{ij} >$ can give us what's the interrupted subtransactions when a system crash happen again.

The correctness of the recovery mechanism is formalized by the lemmas and theorems given below.

6.2 Correctness Proof for Failure Recovery Algorithm

Lemma 4

The advanced transaction system will be restored back to consistent state after the recovery process.

Proof 4

We consider the advanced transaction systems contain a database, where the data items are stored. In the event of a failure, the database state may become inconsistent. This is because some subtransactions may have partially executed when a crash occurs. Our recovery process uses an undo-redo approach. The undo phase finds out all the aborted and interrupted subtransactions and restores the before image of the data items modified by these subtransactions. Thus, the effects of all aborted and interrupted subtransactions are removed in the undo phase. Next, the redo phase finds out all the committed subtransactions and restores the after image of the data modified by these subtransactions. Thus, the effects of all committed transactions are written on the disk in this phase. The state of the database at the completion of the undo and redo phases will contain all the changes made by committed transactions and will not have any changes made by aborted or interrupted subtransactions. This ensures that the database is in a consistent state after the completion of the undo and redo phases. The last phase is the re-execution phase. In this phase all incomplete advanced transactions will be completed. Since the scheduler uses strict 2 phase lock for executing subtransactions, database consistency will be preserved after the execution of the subtransactions making up incomplete advanced transaction.

Lemma 5

All interrupted advanced transactions will have completed execution after the recovery process.

Proof 5

In the undo phase, we identify all the advanced transactions that were interrupted because of a crash. We find the $< START\ CKPT\ M >$ record corresponding to the last $< END\ CKPT >$

Dependency with T_{ij} Interrupted	T_{ik} *Committed*	T_{ik} *Aborted*	T_{ik} *Interrupted*	T_{ik} *Unscheduled* or Initiation
$T_{ij} \rightarrow_{sc} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{c} T_{ik}$	NE	possible	possible	possible
$T_{ij} \rightarrow_{a} T_{ik}$	NE	possible	possible	possible
$T_{ij} \rightarrow_{s} T_{ik}$	NE	NE	NE	possible
$T_{ij} \rightarrow_{t} T_{ik}$	NE	NE	NE	possible
$T_{ij} \rightarrow_{b} T_{ik}$	possible	possible	possible	po ssible
$T_{ij} \rightarrow_{bc} T_{ik}$	NE	NE	NE	possible
$T_{ij} \rightarrow_{ba} T_{ik}$	NE	NE	NE	possible
$T_{ij} \rightarrow_{fbc} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{fbb} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{fba} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{fbt} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{ex} T_{ik}$	NE	possible	possible	possi ble
$T_{ij} \rightarrow_{fca} T_{ik}$	possible	NE	NE	NE

Table 11: Possible States of Subtransaction T_{ik} if Subtransaction T_{ij} is Interrupted

record in the log. M contains all subtransactions that were active when the checkpointing was done. From the advanced transactions that correspond to subtransactions in M and those that were started later, we identify those that did not have a record of the form $< COMPLETE\ AT_k >$. These are identified as incomplete advanced transactions. The incomplete advanced transactions may have one or more subtransactions that are in the interrupted state. Our recovery algorithm will undo all interrupted and aborted subtransactions and redo all the committed subtransactions. The final phase is to continue re-execution of the interrupted subtransaction. This includes creating the state table for the interrupted transaction which is given to the scheduler. The scheduler also receives the inputs of this advanced transaction. Since we assume that the inputs provided by the user will be able to satisfy at least one completion set, this advanced transaction will be able to complete.

Lemma 6

Dependencies associated with every advanced transaction will be satisfied after the recovery process.

Proof 6

Our recovery algorithm has three phases. We prove that dependencies are satisfied in each phase. First, we prove that dependencies are satisfied in the undo phase. Two kinds of subtransactions are undone – those that have been interrupted and those that have been aborted. We begin by considering

Dependency with T_{ik} Interrupted	T_{ij} *Committed*	T_{ij} *Aborted*	T_{ij} *Interrupted*	T_{ij} *Unscheduled* or Initiation
$T_{ij} \rightarrow_{sc} T_{ik}$	NE	possible	possible	possi ble
$T_{ij} \rightarrow_{c} T_{ik}$	possible	possible	possible	po ssible
$T_{ij} \rightarrow_{a} T_{ik}$	possible	possible	possible	po ssible
$T_{ij} \rightarrow_{s} T_{ik}$	possible	possible	NE	NE
$T_{ij} \rightarrow_{t} T_{ik}$	possible	possible	NE	NE
$T_{ij} \rightarrow_{b} T_{ik}$	possible	possible	possible	NE
$T_{ij} \rightarrow_{bc} T_{ik}$	possible	NE	NE	NE
$T_{ij} \rightarrow_{ba} T_{ik}$	NE	possible	NE	NE
$T_{ij} \rightarrow_{fbc} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{fbb} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{fba} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{fbt} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{ex} T_{ik}$	possible	possible	possible	possible
$T_{ij} \rightarrow_{fca} T_{ik}$	NE	NE	NE	possible

Table 12: Possible States of Subtransaction T_{ij} if Subtransaction T_{ik} is Interrupted

the case of interrupted subtransactions. For this, we use two tables that show possible states of a subtransaction T_{ik} (T_{ij}) when subtransaction T_{ij} (T_{ik}) is interrupted. Each entry can take either the value *possible* or *NE*. *NE* means that the subtransaction cannot be in that state. *possible* means that the subtransaction can be in that state. For instance, consider Table 11. Consider the last row. The first column in this row has the entry *possible*. It means that when T_{ij} is interrupted, T_{ik} may be in a committed state. The second column in this row has the entry *NE*. This means that when T_{ij} is interrupted, as per our scheduling rules, it is not possible for T_{ik} to be in the aborted state. Thus, for dependencies of the form $T_{ij} \rightarrow_x T_{ik}$, if T_{ij} is interrupted, T_{ik} may be in an aborted, committed, interrupted, or initiation state. From the table we observe that if T_{ik} is already in aborted or committed states, undoing T_{ij} will not violate the dependency. We also observe that if T_{ik} is in the interrupted or initiation state and T_{ij} is aborted, no dependency is violated – this is because T_{ij} and T_{ik} will be treated as if they have not been executed. In short, we see that aborting the interrupted subtransaction T_{ij} does not affect the dependencies of the form $T_{ij} \rightarrow_x T_{ik}$. Similarly, from Table 12 we can infer that aborting the interrupted subtransaction T_{ik} does not affect dependencies of the form $T_{ij} \rightarrow_x T_{ik}$. Thus by undoing interrupted subtransactions, the dependencies are not violated.

Next, we consider the case of dependencies associated with subtransactions in the final state.

Since our scheduler preserves dependencies, it will allow a subtransaction to move to the final state only when the dependencies associated with it are satisfied. Thus, for subtransactions that aborted or committed before the crash occurred, the dependencies will be satisfied. Undoing these aborted subtransactions or redoing committed ones will not violate the dependencies.

The last part that remains to be proved is that the re-execution phase preserves dependencies. Since re-execution of incomplete advanced transactions is performed by the scheduler, the dependencies will be preserved in this phase as per Theorem 11.

Theorem 12

The recovery algorithm described above ensures reliable recovery as per Definition 28.

Proof 12

The proof follows from Lemmas 4, 5 and 6.

CHAPTER VII

DAMAGE ASSESSMENT AND REPAIR FOR ADVANCED TRANSACTIONS

7.1 Repair from Malicious Attacks

Advanced transaction systems may be subject to malicious attacks, since vulnerabilities cannot be completely removed from systems and preventive measures sometimes fail, Malicious attackers may intrude into the system by some stolen or illegal identifications, and they can launch some unwanted illegal subtransactions to gain some personal benefits. We call these illegal subtransactions as *malicious subtransactions* in this chapter.

Repair from malicious attacks have been investigated in the context of database systems [3, 4, 5, 65, 66, 69, 70, 83, 110, 122]. In such systems, a transaction executed by a malicious user might corrupt some data item. Other transactions reading from this data item and writing on other data items help spread the damage. Techniques for damage assessment and repair in database systems are not adequate for advanced transactions. This is because transactions in a database are independent entities. The only way in which one transaction depends on another is through read-write dependencies, where one transaction writes a data item that is read by another. An advanced transaction consists of subtransactions that have control-flow dependencies specified among them, in addition to the read-write dependencies. These control-flow dependencies co-ordinate the execution of subtransactions. The existence of dependencies can complicate the repair process for malicious attacking, since damage could be spread faster and wider due to the constraints of these dependencies. The presence of these dependencies poses new challenges and requires new techniques for damage assessment and repair.

In this chapter, we propose repair algorithms for advanced transactions. We assume that the attackers successfully authenticated into the system through illegal forms, launched some unwanted

subtransactions and achieved some damages. A malicious attacker can launch one or more malicious subtransactions in an advanced transaction. One malicious subtransaction could corrupt data items accessed by some benevolent subtransaction; dependencies could also help spread the damage caused by malicious subtransactions. Consequently, a number of good advanced transactions could get affected.

As an illustration of the context we focus on, consider one advanced transaction AT_i containing two subtransactions T_{ij} (decrement an account balance) and T_{ik} (check and charge maintenance fee). Suppose there is a *force-begin-on-commit* dependency specified between T_{ij} and T_{ik}, which requires if T_{ij} commits then T_{ik} must begin execution. Assuming a malicious attacker triggers the execution of subtransaction T_{ij} that decrements a bank account balance $acct_x$ from 2000 to 100. On successful completion of subtransaction T_{ij}, subtransaction T_{ik} is also executed automatically due to the constraints of the force-begin-on-commit dependency, and T_{ik} charges a maintenance fee 15 since balance of $acct_x$ has fallen below the designated limit. Following, subtransaction T_{mn} of a different advanced transaction AT_m reading the value in $acct_x$ and using this to compute $interest_x$, $total\_bal$, further corrupts other data items as well. We can see that, damage can be spread due to the read-write relationship among subtransactions and the control-flow dependencies as well.

One simple repair solution is the rollback strategy – undo all the subtransactions of the malicious ones and as well as the good ones that have executed after the malicious attacks, and then re-execute the good advanced transactions. This strategy is very inefficient for advanced transactions because the execution of advanced transaction tends to have long duration, and re-execution of unaffected benign advanced transactions could take long time and this is unwanted. Moreover, the repair algorithm for advanced transactions must take into consideration of the effects of dependencies in repair. If the repair algorithm undoes a subtransaction in repairing process, it must ensure that the dependencies on this subtransaction are satisfied. Reconsider the above example again. Upon detection of the malicious attacks, the recovery strategy in traditional database system will undo the malicious subtransaction T_{ij}, it also identifies T_{mn} as affected such that it reads from T_{ij}. However, it will not be able to remove the effects of the affected subtransaction T_{ik} whose execution was triggered by T_{ij} due to control-flow dependencies. The consequence is that the account customer will be charged 15 maintenance fee incorrectly. Further, since the repair algorithm undoes T_{mn},

some other subtransaction T_{mq} in the same advanced transaction AT_m may also need to be undone due to certain dependency constraints between T_{mn} and T_{mq}. The repair algorithm should be able to ensure satisfactions of all dependencies.

In this chapter, we propose a repair algorithm that assesses and removes all effects of malicious subtransactions. Our approach tries to improve upon the traditional rollback strategy by minimizing the number of benign advanced transactions that need to be undone and re-executed. We also take into account the nature of the dependencies that exist between the subtransactions of an advanced transaction when we are doing damage assessment. The analysis on properties of dependencies enables us to find out the specific subtransactions in the advanced transaction that are affected and to take proper action to remove the malicious effects. We discriminate malicious subtransactions, affected subtransactions by read-write dependencies, affected subtransactions by control-flow dependencies in repairing process, and unaffected good subtransactions. The repair algorithm will undo the malicious and affected subtransactions, and then re-execute the affected ones; while keep the unaffected benign advanced transaction untouched. Minimizing the number of subtransactions that are undone and re-executed speeds up the damage repair process.

7.2 Strategy for Damage Assessment and Repair

As stated earlier, advanced transaction systems may be subjected to malicious attacks. Survivability techniques in traditional transaction processing systems do not work for advanced transaction models due to the existence of control-flow dependencies in advanced transactions. The failure recovery mechanisms for advanced transactions [92], working towards failure and system crashes, do not address how to repair from malicious attacks either. In this section, we will discuss the mechanisms for damage assessment and repair from malicious attacks for the advanced transactions.

We assume that the attackers successfully authenticated into the system through some illegal forms, and then they can launch some unwanted subtransactions, and achieved some damages to the advanced transaction system. We focus on those intrusions that inject malicious subtransactions into the advanced transaction instead of the attacks that crash or cause denial-of-service of the system. These intrusions happen when attackers authenticate into the system with a stolen login account or intercepted password, or when some defense mechanisms (such as access control) are broken by

the attacker. Under these intrusions, the attackers can forge or alter some subtransactions to corrupt the system and gain some personal benefit. Malicious subtransactions can spread damage to more subtransactions in the systems.

We assume that there is an Intrusion Detection Manager (IDM) in the system. The IDM can identify the malicious attacks periodically. The IDM reports the malicious subtransactions that were either created or launched by attackers. The IDM monitors the system or network activity to discover attempts to disrupt or gain illicit access to system, by means of statistical profiles or known pattern of attacks. The IDM can identify the direct damage caused by malicious attacks, but it cannot discover all the damage done to the system. The damage directly caused by the attacker can spread into the whole advanced transaction processing system by the execution of normal subtransactions without being detected by the IDM.

7.2.1 Information Needed by the Repair Algorithm

Damage assessment and repair in advanced transaction systems are much more complex than the traditional transaction processing system due to the existence of subtransactions and control-flow dependencies. The repair process will need to know the execution state of each advanced transaction and every subtransaction as well. After some malicious attacks take place, proper damage assessment must be performed and correct actions must be taken for every subtransaction, and the actions adopted must ensure the satisfaction of dependencies specified with these subtransactions.

In our algorithm, the log records store the information about the state of the advanced transactions and the subtransactions. It also provides information about read-write dependencies. The advanced transaction specification, as discussed in Chapter 3, is also required, since it stores information about the details of the control-flow dependency specifications.

In order to repair from a malicious attack, the state information of an advanced transaction needs to be logged onto some stable storage. We propose that such information be stored in the system log. Execution of a primitive for an advanced transaction (like start and complete), a subtransaction primitive (like begin, commit and abort), or a subtransaction operation (like read and write) results in the insertion of log a record.

Execution of a start primitive on an advanced transaction results in the insertion of the following

log record. $< START\ AT_i,\ S_i >$ where AT_i indicates the advanced transaction id and S_i indicates the set of all submitted subtransactions with its inputs in the advanced transaction. The completion of the advanced transaction is indicated by a log record $< COMPLETE\ AT_i >$.

Execution of the primitive begin for subtransaction T_{ij} results in the following records being inserted in the log: $< START\ T_{ij} >$ where T_{ij} is the subtransaction id. Note that the subtransaction id includes information about the advanced transaction id of which this subtransaction is a part. Similarly, execution of the primitives commit or abort results in inserting the following log records for subtransaction T_{ij}: $< COMMIT\ T_{ij} >$ or $< ABORT\ T_{ij} >$.

Execution of operations also cause log records to be inserted. A read operation causes the log record $< T_{ij}\ X, value >$ to be inserted where *value* is the value of X that was read by the subtransaction. Similarly, a write operation causes the log record $< T_{ij}\ X, v, w >$ where X is the data item written by the subtransaction and v and w indicate the old value and the new value of the data item written by subtransaction T_{ij}.

7.2.2 Impacts of Dependencies on Damage Assessment

We first formally define the malicious advanced transaction, malicious subtransaction and benign advanced transaction.

Definition 30

[Malicious Advanced Transaction] An advanced transaction is considered a *malicious advanced transaction*, if there is one or more committed subtransactions in this advanced transaction that have been executed by a malicious user.

Definition 31

[Malicious Subtransaction] A subtransaction T_{ij} is considered a *malicious subtransaction* if T_{ij} is a committed subtransaction in a malicious advanced transaction.

Definition 32

[Benign Advanced Transaction] An advanced transaction is considered a *benign advanced transaction*, if there are no subtransactions executed by a malicious user in this advanced transaction.

The malicious advanced transactions are relatively easy to repair. In repairing process, we undo the whole malicious advanced transaction. This is because we consider the malicious advanced

transactions as unwanted and we need to remove their effects. However, damage assessment and repair for benign advanced transactions are far more complicated. To better understand the effects of attacks and the spread of damage, we need to examine the impacts of different dependencies (the read-write dependencies and control-flow dependencies) during damage assessment.

7.2.2.1 Impacts of Read-write Dependencies

In this chapter we consider only one kind of external dependency (dependencies that are inter-advanced transactions), that is the *read-write dependency*. Such dependencies can also be presented between subtransaction among the same advanced transaction. Concurrency control mechanisms [20], such as two-phase locking and time-stamping, ensure correct execution in the presence of such dependencies. We assume these concurrency control mechanisms exist to take care of such dependencies in execution of advanced transactions.

As mentioned earlier, damage can spread across different advanced transactions through read-write dependencies. A subtransaction in a benign advanced transaction can read a corrupted data item from a malicious subtransaction, and thus it is affected. In turn, the affected subtransaction can spread the damage further, which implies, subtransactions in many benign advanced transactions can be *read-write dependency affected.*

Definition 33

[Conflicting Operation] Two data operations $o_i[d]$ and $o_j[d]$ conflict with each other if they operate on the same data object d and at least one of them is a write.

Definition 34

[Read-write dependent upon] Subtransaction T_{ij} is *read-write dependent upon* another subtransaction T_{kl}, if there exists a read-write dependency between T_{kl} and T_{ij} in history on data item x such that:

1. T_{ij} reads x after T_{kl} has updated x, T_{kl} does not abort after T_{ij} reads x, and,

2. if any T_{pq} updates x after T_{kl} has updated x but before T_{ij} reads it, then T_{pq} is aborted.

Definition 35

[Read-write Affected Subtransaction] A subtransaction T_{qj} is considered as a read-write affected subtransaction if T_{qj} reads data written by a malicious or affected subtransaction T_{pi}.

Definition 36

[Dirty Data Item] A data item x is *dirty* if x is in the write set of any malicious subtransaction or affected subtransaction.

In other word, subtransaction T_{ij} is a read-write affected subtransaction if it read some dirty data items. All read-write affected subtransactions must be undone.

7.2.2.2 Impacts of Control-flow Dependencies

A malicious or affected subtransaction could trigger or change the execution of some other subtransactions within the same advanced transaction because of the presence of control-flow dependencies. These new affected subtransactions can further spread the damage.

Damage can spread across advanced transactions through read-write dependencies. Within one affected benign advanced transaction who contains read-write affected subtransactions, there could possibly exist more subtransactions also got affected due to the presence of *control-flow dependencies*. In repair, we need to *undo* the read-write affected subtransactions. The effects of undoing one subtransaction is similar to the *abort event* for this subtransaction in normal execution – all data items changed by this subtransaction are restored to their before images and the effects of this subtransaction are totally removed from the system. Due to the constraints of some control-flow dependencies, the undo action on one subtransaction may require to adopt proper actions (like undo or abort) on the other subtransactions in the same advanced transaction. In other words, if we undo some subtransactions, there may exist some other subtransactions that must also be undone, in order to enforce certain type of dependencies specified among these subtransactions. We refer to this property as the *force-undo-on-undo property*.

Definition 37

[Force-Undo-on-Undo Property] A control-flow dependency $T_{ij} \rightarrow_d T_{ik}$ (or $T_{ik} \rightarrow_d T_{ij}$) is said to hold the *force-undo-on-undo property* if performing an undoing action on T_{ij} enforces the undoing action on T_{ik} due to the dependency constraints. In simple, force-undo-on-undo property holds if the dependency requires that if T_{ij} is undone then T_{ik} must also be undone. Formally, $undo_{ij} \Rightarrow undo_{ik}$.

Dependency	$T_{ij} \rightarrow_d T_{ik}$ is force-undo-on-undo?	Action in repair
$T_{ij} \rightarrow_{c/sc/t} T_{ik}$	No	No
$T_{ij} \rightarrow_a T_{ik}$	Yes	if T_{ik} committed, undo T_{ik}
$T_{ij} \rightarrow_{bc} T_{ik}$	Yes	if T_{ik} committed, undo T_{ik}
$T_{ij} \rightarrow_{b/ba} T_{ik}$	No	No
$T_{ij} \rightarrow_{fbb/fbc/fba/fbt} T_{ik}$	No	No
$T_{ij} \rightarrow_{fca/ex} T_{ik}$	No	No

Table 13: $T_{ij} \rightarrow_d T_{ik}$ is a Force-undo-on-undo Dependency?

Dependency	$T_{ik} \rightarrow_d T_{ij}$ is force-undo-on-undo?	Action in repair
$T_{ik} \rightarrow_{c/t/a} T_{ij}$	No	No
$T_{ik} \rightarrow_{sc} T_{ij}$	Yes	if T_{ik} committed, undo T_{ik}
$T_{ik} \rightarrow_{b/bc/ba} T_{ij}$	No	No
$T_{ik} \rightarrow_{fbb/fbc/fba/fbt} T_{ij}$	No	No
$T_{ik} \rightarrow_{fca/ex} T_{ij}$	No	No

Table 14: $T_{ik} \rightarrow_d T_{ij}$ is a Force-undo-on-undo Dependency?

Undoing action removes the effects of subtransactions made on the system. If, undoing a subtransaction T_{ij} necessitates the undoing action on T_{ik} due to certain dependency constraint, we say this dependency has the force-undo-on-undo property. For instance, the *abort* dependency is one dependency having the force-undo-on-undo property. Note that not all control-flow dependencies have the force-undo-on-undo property. Only a subset of the control-flow dependencies exhibits this property. The sets of control-flow dependencies holding the force-undo-on-undo property are shown in Table 13 and Table 14.

Three different forms of control-flow dependencies have the force-undo-on-undo property. With these dependencies, if T_{ij} is undone in repair, T_{ik} is also required to be undone. These dependencies are:

1. *abort dependency* from T_{ij} to T_{ik} – $T_{ij} \rightarrow_a T_{ik}$,
2. *begin-on-commit dependency* from T_{ij} to T_{ik} – $T_{ij} \rightarrow_{bc} T_{ik}$, and
3. *strong commit dependency* in reversed direction T_{ik} to T_{ij} – $T_{ik} \rightarrow_{sc} T_{ij}$

More discussions on the force-undo-on-undo property for these dependencies are provided in Observation 1 in section 7.4.

The following example illustrates the repair actions needed for force-undo-on-undo type of dependencies. Suppose there is an abort dependency $T_{w1} \rightarrow_a T_{w2}$ specified between two subtransactions T_{w1} (book an air ticket to resort B) and T_{w2} (book the hotel room in resort B), assuming T_{w1} is identified as read-write affected on a corrupted data item p_t, and T_{w1} needs to be undone. During repair process, if we undo T_{w1} and keep T_{w2} committed in the advanced transaction system, the result is erroneous – since all effects of T_{w1} is removed in the advanced transaction system, and no air ticket is reserved. In this case, the subtransaction T_{w2} must be undone as well (cancel the reservation of hotel rooms), here we call T_{w2} is *undo-action dependent upon* T_{w1} and T_{w2} is an *undo-action affected subtransaction.*

Definition 38

[Undo-action Dependent Upon] A subtransaction T_{ik} is *undo-action dependent upon* another subtransaction T_{ij} of the same advanced transaction, if, there is a control-flow dependency d_u ($T_{ij} \rightarrow_{d_u} T_{ik}$ or $T_{ik} \rightarrow_{d_u} T_{ij}$) specified between T_{ik} and T_{ij}, and the dependency d_u has the force-undo-on-undo property.

Definition 39

[Undo-action Affected Subtransaction] In an advanced transaction AT_i, subtransaction T_{ik} is said to be an *undo-action affected subtransaction* if –

1. there is a control-flow dependency d_u specified in AT_i with T_{ik} in the form of $T_{ij} \rightarrow_{d_u} T_{ik}$ or $T_{ik} \rightarrow_{d_u} T_{ij}$,
2. d_u is a control-flow dependency having the force-undo-on-undo property,
3. T_{ij} is identified to be an affected subtransaction – either it is a read-write affected subtransaction or it is an undo-action affected subtransaction.

We do not consider data-flow dependencies while performing repair. As mentioned earlier, whenever there is a data-flow dependency between two subtransactions T_{ij} and T_{ik}, there exists a *begin-on-commit* dependency between those subtransactions. Corresponding repair action will be taken for this control-flow dependency. No separate action needs to be taken for data-flow dependencies.

Definition 40

[Affected Subtransaction] Subtransaction T_{wp} is considered an *affected subtransaction* if either it is a read-write affected subtransaction or it is an undo-action affected subtransaction.

Therefore, during damage assessment and repair, there are two possibilities that one subtransaction can be identified as *affected subtransaction*:

(I) If a subtransaction T_{mp} read some dirty data items that have been written by a malicious or affected subtransaction T_{nq};

(II) If a subtransaction T_{ms} is undo-action dependent upon some affected subtransaction T_{mr}. Undoing the affected subtransaction T_{mr} could necessitate the undoing action on subtransaction T_{ms} due to the constraints of the force-undo-on-undo type of control-flow dependency specified among them.

7.2.3 Strategy and Actions in Repair

We next present the repair strategy for advanced transactions to repair from malicious attacks. We discriminate the malicious advanced transactions, affected benign advanced transactions (could contain either read-write affected or undo-action affected subtransactions), and unaffected benign advanced transactions.

Definition 41

[Repair Actions] The *repair actions* for advanced transaction are listed following:

- For **malicious advanced transactions**, undo all the subtransactions;

- For **affected subtransactions** in benign advanced transactions, need proper repair actions –

    1. read-write affected subtransactions, undo them first, then re-execute them;
    2. undo-action affected subtransactions, undo them, they may be re-scheduled by the scheduler later;

- For **unaffected subtransactions** in benign advanced transactions, keep them unchanged.

Let's use an example to illustrate how the repair mechanism works.

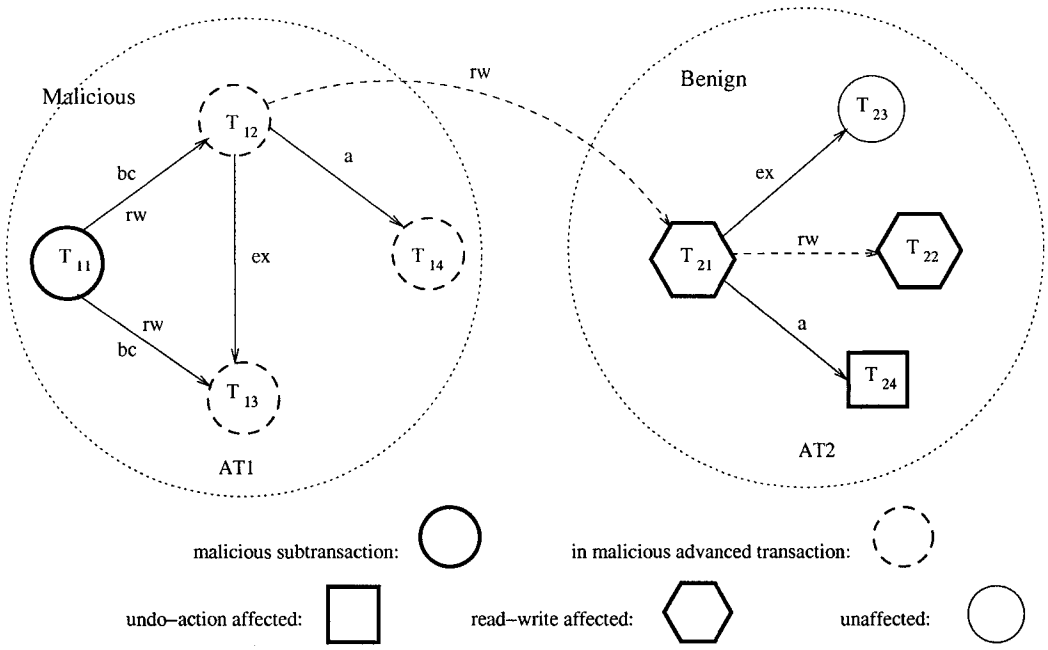

Figure 12: Example of Damage Assessment and Repair

Example 7

In the example shown in figure 12, we have two advanced transactions, AT_1 and AT_2, and the subtransactions T_{11} is a malicious subtransaction injected by some intruder to the advanced transaction system. The dependencies in AT_1, shown in the diagram, are: $\{T_{11} \rightarrow_{bc} T_{12}$, $T_{12} \rightarrow_{bc} T_{13}$, $T_{12} \rightarrow_{ex} T_{13}$, and $T_{13} \rightarrow_{a} T_{14}.\}$ The dependencies in AT_2 are: $\{\ T_{21} \rightarrow_{ex} T_{23}$ and $T_{21} \rightarrow_{a} T_{24}.\ \}$

In addition, there exists some read-write dependencies between the subtransactions across the two advanced transactions in execution: $\{\ T_{12} \rightarrow_{rw} T_{21}$, and $T_{21} \rightarrow_{rw} T_{22}.\ \}$

We know that advanced transactions AT_1 is malicious advanced transactions since it contains T_{11} as a malicious subtransaction; and AT_2 is a benign advanced transaction since it contains no malicious subtransaction. The malicious advanced transaction AT_1 must be undone and removed its effects. For all benign advanced transactions, if there exist some subtransactions affected by any subtransaction in AT_1, they must be identified and repaired. We can detect that subtransaction T_{21} is read-write affected since there is a read-write dependency between T_{12} and T_{21}. Moreover, we detect that subtransaction T_{24} is undo-action dependent upon T_{21}, since there is a $T_{21} \rightarrow_a T_{24}$ dependency and abort dependency is force-undo-on-undo type. T_{24} is identified as an undo-action affected subtransaction. We can further identify that T_{22} is read-write affected by T_{21}. The

subtransactions that are affected (T_{21}, T_{22}, T_{24}) need to be undone. Subtransaction T_{23} do not need to be undone, since it is not affected by any malicious subtransaction or affected subtransactions.

7.3 *Repair Algorithms for Advanced Transactions*

First we state the assumptions that we make:

1. We consider the effect of committed malicious subtransactions in repair.
2. We assume that the execution of different advanced transactions can interleave with each other.
3. We use the strict two phase-locking mechanism [20] to execute each subtransaction of an advanced transaction.

We denote the set of committed malicious subtransactions, in the history of the advanced transaction system, by the set B. We assume these malicious subtransactions are detected by the IDM periodically. Based on these malicious subtransactions, we can identify all the malicious advanced transactions – if there is one or more malicious subtransaction in this advanced transaction, it is considered to be malicious. We represent the set of malicious advanced transactions as the set BT. The basic idea is that all malicious advanced transactions must be undone. Among the benign advanced transactions, some subtransactions might be read-write dependency affected by the subtransactions in malicious advanced transactions, and these affected subtransactions might in turn affect other more subtransactions, which can be either undo-action affected or read-write affected. We must identify all affected subtransactions, and remove the effects of all malicious and affected subtransactions in repair.

We next present the details of our algorithm. Our algorithm proceeds in four phases. The first phase is to undo all malicious advanced transactions and identify the sets of committed subtransaction in benign advanced transactions. The second phase is the damage assessment phase, we find out all the undo-action affected subtransactions and read-write affected subtransactions. In this phase, the $getUndoActionAffected(T_{ij}, committed[i])$ function will find out all the undo-action affected subtransactions in this advanced transaction with an identified affected subtransaction T_{ij}, based on

Table 13 and Table 14. The third phase undoes all affected subtransactions. The fourth phase is responsible for re-execution and continuation of incomplete advanced transactions.

Algorithm 18

Advanced Transaction Repair Algorithm

Input: (i) the system log, (ii) advanced transaction specifications, (iii) **BT** – set of malicious advanced transactions

Output: a consistent execution state in which the effects of all malicious and affected subtransactions are removed

Procedure AdvancedTransactionRepair

Phase 1: Undo malicious advanced transactions,

and identify the sets of committed subtransactions in benign advanced transactions

/*set holding aborted subtransactions of all advanced transactions*/

$globalAborted = \{\}$

/*sets holding the committed subtransactions of benign advanced transactions AT_w*/

$committed[w] = \{\}$

/*set holding all committed subtransactions of all benign advanced transactions*/

$committedBenign = \{\}$

begin

/* Scan backwards until we reach $< START\ AT_p\ S >$

where AT_p is the earliest malicious advanced transaction*/

do

get the last unscanned log record

switch the log record

case the log record is $< ABORT\ T_{wi} >$

$globalAborted = globalAborted \cup \{T_{wi}\}$

case the log record is $< COMMIT\ T_{wi} >$

if $AT_w \notin$ **BT** $\wedge\ T_{wi} \notin globalAborted$

for AT_w, $committed[w] = committed[w] \cup \{T_{wi}\}$

case the log record is update record $< T_{wi}, x, v, w >$

/*undo the subtransaction in the malicious advanced transaction*/

if $AT_w \in \mathbf{BT} \wedge T_{wi} \notin globalAborted$

change the value of x to v /* restores before image of x */

case the log record is $< START\ T_{wi} >$

if $AT_w \in \mathbf{BT} \wedge T_{wi} \notin globalAborted$

write $< ABORT\ T_{wi} >$ log record

case the log record is $< START\ AT_w\ S >$

if $AT_w \in \mathbf{BT}$

write $< ABORT\ AT_w >$ log record

else

$committedBenign = committedBenign \cup committed[w]$

default continue;

until the log record is $< START\ AT_p\ S >$,

and AT_p is the earliest malicious advanced transaction.

end /* phase 1 */

/* after phase 1, we have undone all malicious advanced transaction, and get the set of committed subtransaction for each completed benign advanced transaction in the history.*/

Phase 2: find all affected subtransactions

$corrupted = \{\}$ /* a set holds all corrupted data items */

$undoList = \{\}$ /* a set holds all affected subtransactions */

$finishedAT = \{\}$ /* a set holds all completed advanced transactions */

begin

/* Scan forward from $< START\ AT_p\ S >$ where AT_p is the earliest malicious advanced transaction*/

while not end of log

begin

switch next log record

case log record = $< COMPLETE\ AT_i >$

if $C_j \subseteq committed[i]$, where $C_j \in C$ *of* AT_i

/* one completion set C_j is matched */

$finishedAT = finishedAT \cup \{AT_i\}$

else

if $AT_i \in finishedAT$

$finishedAT = finishedAT - AT_i$

case log record = $< T_{ij}, X, v, w >$ /* a write record */

/* mark this corrupted data item if it was

written by malicious or affected subtransaction */

if $(AT_i \in \mathbf{BT} \vee T_{ij} \in undoList) \wedge T_{ij} \notin globalAborted$

$corrupted = corrupted \cup \{X\}$

case log record = $< T_{ij}, X, value >$ /* a read record */

/* check whether it reads from a corrupted data item*/

begin

if $X \in corrupted \wedge AT_i \notin \mathbf{BT} \wedge T_{ij} \notin globalAborted$

$undoList = undoList \cup \{T_{ij}\}$

/* get the set of undo-action affected subtransactions */

$undoAffectedSet = getUndoActionAffected(T_{ij}, committed[i])$

if $undoAffectedSet \neq NULL$

$undoList = undoList \cup undoAffectedSet$

scan back to the earliest affected subtransaction in $undoAffectedSet$

end

case log record = $< COMMIT\ T_{ij} >$

if $T_{ij} \in undoList$

$committed[i] = committed[i] - T_{ij}$

end /* while loop */

end /* phase 2 */

Phase 3: undo all the subtransactions in the undoList set.

$submit = \{\}$ /* a set holds all subtransactions needed to submit to scheduler */

$input = \{\}$ /* a set holds all the input of subtransactions needed to submit */

begin

/* scan backwards from the end of the log and undo all subtransactions in $undoList$ until reach the start of the earliest malicious advanced transaction */

do

get the last unscanned log record

switch the log record

case log record = $< START\ AT_i\ S >$

if $AT_i \notin finishedAT$

for each subtransaction T_{ij} in S

$submit = submit \cup \{T_{ij}\}$

if there is input with T_{ij}

$input = input \cup (T_{ij},\ v)$

case log record = $< T_{ij}, X, u, v >$

if $T_{ij} \in undoList \vee X \in corrupted$

restore the value of X to u /* restoring before image */

case log record = $< START\ T_{ij} >$

if $T_{ij} \in undoList$

write $< ABORT\ T_{ij} >$ log record

until reach $< START\ AT_p\ S >$ of the earliest malicious advanced transaction

end /* phase 3 */

Phase 4: resubmit the incomplete advanced transactions to scheduler and continue execution.

begin

for each $committed[i] \in committedBenign$

if $i \notin finishedAT$

create $StateTable_t$

$toschedule = \{\ \}$

for each subtransaction $T_{ix} \in$ *submit set of* AT_i

Begin

if subtransaction $T_{ix} \in undoLsit$

$StateTable_t[T_{ix}] =' initiation'$

$toschedule = toschedule \cup \{T_{ix}\}$

else if subtransaction $T_{ix} \in globalAborted$

$StateTable_t[T_{ix}] =' aborted'$

else if subtransaction $T_{ix} \in committed[i]$

$StateTable_t[T_{ix}] =' committed'$

else /* for the subtransactions has been submitted, but did not execute */

$StateTable_t[T_{ix}] =' initiation'$

$toschedule = toschedule \cup \{T_{ix}\}$

end

submit $StateTable_t$, *toschedule and input set* to the scheduler

end /* phase 4 */

After presenting the advanced transaction repair algorithm, we give the detail of the function, which finds all undo-action affected subtransactions based on an affected subtransaction and the dependencies specified with it. This function is recursive and finds out all it undo-action affected neighbors. Based on the previous analysis for each dependency in Table 13 and Table 14, we know only three control-flow dependencies can cause more subtransactions to be undo-action affected. The advanced transaction specification $AT_i =< S, D, C >$ as proposed in Chapter 3 will be used to determine the affected subtransaction's neighbors (which are those connected by control-flow dependencies).

Algorithm 19

Find undo-action affected Subtransactions

Input: (i) the affected subtransaction T_{ij} (ii) the committed subtransactions set committed[i] for the advanced transaction AT_i

Output: a set of all affected subtransactions in this advanced transaction which are undo-action

affected by subtransaction T_{ij}

Procedure *getUndoActionAffected*

$affectedList = \{\}$ /* set holds the affected subtransactions and will be returned */

$anewList = \{\}$ /* temporary set that holds the affected subtransaction */

begin

for each connected neighbor T_{ik} to T_{ij} with a control-flow dependency

if $T_{ik} \in committed[i]$

begin

if there is $T_{ik} \rightarrow_{sc} T_{ij}$

$affectedList = affectedList \cup \{T_{ik}\}$

if there is $T_{ij} \rightarrow_{bc} T_{ik}$

$affectedList = affectedList \cup \{T_{ik}\}$

if there is $T_{ij} \rightarrow_{a} T_{ik}$

$affectedList = affectedList \cup \{T_{ik}\}$

if $T_{ik} \cup affectedList$

$committed[i] = committed[i] - T_{ik}$

end

for each subtransaction $T_{ik} \in affectedList$

$anewList = anewList \cup getUndoActionAffected(T_{ik}, committed[i])$

$affectedList = affectedList \cup anewList$

return *affectedList*

end

When repair processing finishes, all malicious advanced transactions are undone and their effects are removed. All affected subtransactions are also undone. They shall be submitted to the scheduler for re-execution.

7.4 Correctness Proof for Repair Algorithm

We first provide the details on the investigation of the *force-undo-on-undo* properties for all control-flow dependencies.

Observation 1

The following set of control-flow dependencies have the force-undo-on-undo property –

1. *abort dependency* from T_{ij} to T_{ik} – $T_{ij} \rightarrow_a T_{ik}$,
2. *begin-on-commit dependency* from T_{ij} to T_{ik} – $T_{ij} \rightarrow_{bc} T_{ik}$, and
3. *strong commit dependency* in reversed direction T_{ik} to T_{ij} – $T_{ik} \rightarrow_{sc} T_{ij}$

No other control-flow dependency has this property, as shown in Table 13 and Table 14.

We first examine the control-flow dependencies in the form of $T_{ij} \rightarrow_d T_{ik}$, checking if undoing of T_{ij} necessitates the undoing action of T_{ik}.

1. Among the *event ordering type* of control-flow dependencies, certain dependency can impose the execution order on the commit event and begin event of two different subtransactions – one subtransaction cannot begin until the other subtransaction is committed. In other words, T_{ik} is not permitted to start execution if the other subtransaction T_{ij} is aborted. The *begin-on-commit* dependency $T_{ij} \rightarrow_{bc} T_{ik}$ is the only dependency imposing this constraint, and it holds the force-undo-on-undo property. Other event ordering type of dependencies do not impose this constraint, and they do not have the force-undo-on-undo property.
2. Among the *event enforcement type* of control-flow dependencies, certain dependency can impose the constraints that if one subtransaction is aborted, then the other one must also abort. The *abort* dependency $T_{ij} \rightarrow_a T_{ik}$ is the only dependency imposing this constraint. In repair, if T_{ij} is undoing and its effects are removed, the the effects of T_{ik} must also be removed and thus T_{ik} must also be undone. Therefore, the abort dependency hold the force-undo-on-undo property. Other event enforcement type of dependencies do not have this property.

Therefore, only two control-flow dependencies in form $T_{ij} \rightarrow_d T_{ik}$ have the force-undo-on-undo property: the *abort* dependency and the *begin-on-commit* dependency.

We next examine the control-flow dependencies in the form of $T_{ik} \rightarrow_d T_{ij}$. Among the *event enforcement type* of control-flow dependencies, only the strong commit dependency $T_{ik} \rightarrow_{sc} T_{ij}$ has the force-undo-on-undo property, since the commit of T_{ik} requires the commit of T_{ij}, if the effect of commit event of T_{ij} is removed, T_{ik} must be undone and its effects must be removed as well. In such a case, if T_{ik} has committed, it must be undone. This dependency will be violated if we undo T_{ij} without undoing T_{ik}. No other dependency in this form has the force-undo-on-undo property. Table 13 and Table 14 list all the force-undo-on-undo dependencies correctly and completely.

Following, we formally define what is an *effective repair* and provide correctness proof for our repair algorithm.

Definition 42

[Effective Repair] The repair for advanced transactions to recover from malicious attacks is said to be an *effective repair* if the following three conditions are satisfied –

1. the repair is **complete**: the effects of every malicious or affected subtransaction are repaired.
2. the repair is **sound**: the effects of only malicious or affected subtransactions are repaired.
3. the repair **respects dependencies**: dependency constraints are satisfied during repair.

The repair is required to be complete. All effects of the malicious subtransactions must be completely removed, including the malicious advanced transactions themselves and all affected subtransactions. The repair should also be sound. An exact repair should avoid unnecessary undoing and redoing of good subtransactions – minimize the time spent on undoing/redoing the unaffected subtransactions. Only subtransactions that were affected needs to be undone and re-executed. The repair must also enforce the dependencies in advanced transactions. Dependencies should not be violated at any time during the life cycle of an advanced transaction. All dependency constraints must be satisfied during repair.

Here we give the proof of correctness for the proposed repair mechanism.

Theorem 13

The repair mechanism described in Section 7.3 ensures effective repair as per Definition 42.

Proof 13

To prove this, we need to show that all the three conditions in Definition 42 are satisfied.

Lemma 7 ensures that the effects of every malicious and affected subtransactions are repaired.

Lemma 8 ensures that the effects of only malicious and affected subtransactions are repaired.

Lemma 9 ensures that all dependency constraints are satisfied during repair.

Lemma 7

The proposed repair algorithm is complete, the effects of every malicious or affected subtransactions are repaired.

Proof 7

Given the set of malicious subtransactions, their effects are completely removed if all the direct and indirect effects caused by control-flow dependencies and read-write dependencies have been assessed and removed. The following four claims are sufficient to show this.

Claim 1. The repair algorithm removes the direct effects of malicious subtransactions. Every malicious subtransaction has been undone in phase 1. We scan the log backwards until the start of the earliest malicious advanced transaction. If the log record is a write operation, we check whether this subtransaction is belonging to any malicious advanced transaction. If it is, we undo this malicious subtransaction by restoring the before image of the of the data item. Every malicious advanced transaction is processed, and every write operation of all malicious subtransactions is undone. It is clear that the direct effects of malicious subtransactions are removed.

Claim 2. All read-write affected subtransactions are detected and their effects are removed. Every read-write affected subtransaction is detected and added to the *undoList* in phase 2. In phase 2, we scan the log forwards from the start of the earliest malicious advanced transaction to the end of the log. Whenever there is a write operation and if it is performed by a committed malicious subtransaction or a committed affected subtransaction, we mark this data item as dirty and add it to the *corrupted* set. Whenever there is a read operation from a dirty data item, we mark this subtransaction as a read-write affected subtransaction and put it into the

undoList. Assume that there are some suspect subtransactions which have not been added to the *undoList* and T_{wm} is the first one. Then according to the algorithm, it means that, when T_{wm} reads a dirty data item x_t, x_t is still not in the *corrupted* set. Since the execution is under strict two-phase lock, when T_{wm} reads data x_t, every malicious or affected subtransaction that writes x_t before the read operation of T_{wm} has already committed and therefore, x_t should have already been added to the *corrupted* set. This contradicts with the assumption. Therefore, all suspect subtransactions reading from dirty data have been detected in the proposed algorithm. These read-write affected subtransactions are undone and their effects are removed.

Claim 3. All undo-action affected subtransactions are detected and their effects are removed. A subtransaction T_{wk} could be undo-action affected by a malicious or affected subtransaction T_{wi}, either in the form $T_{wi} \rightarrow_{dx} T_{wk}$ or $T_{wk} \rightarrow_{dy} T_{wi}$, if we undo T_{wi} then T_{wk} must also be undone due to dependency constraints. In other words, the effects made by the commit event of T_{wk} is contingent upon the existence of the commit effects of T_{wi}. There are several of these cases: (I) Some control-flow dependencies impose an execution order on the different subtransactions – subtransaction T_{wk} cannot begin execution without the successful commit of T_{wi}. The begin-on-commit dependency $T_{wi} \rightarrow_{bc} T_{wk}$ is this type. (II) Some control-flow dependencies enforce one subtransaction to abort if the other subtransaction is aborted. The abort dependency $T_{wi} \rightarrow_{a} T_{wk}$ is this type. In these two cases, if T_{wi} is detected to be an affected subtransaction and needs to be undone, then T_{wk} must also be undone and its effects should be removed, since it should not exist alone without the commit effects of T_{wi}. No other control-flow dependencies require this kind of constraint. Therefore, we only have these two entries in the Table 13. (III) The strong commit dependency $T_{wk} \rightarrow_{sc} T_{wi}$ implies a reversed implicit dependency $T_{wi} \rightarrow_{a} T_{wk}$, thus we have the entry for strong commit dependency in Table 14. Based on the analysis in the two tables, there are no other dependencies that could affect T_{wk}, given T_{wi} is a malicious or affected subtransactions. Therefore, we know that, all undo-action affected subtransactions can be detected in phase 2 of the algorithm; we then undo them in phase 3 and their effects will be completely removed.

Claim 4. Every dirty data item is restored to its earliest clean value. In phase 3 of the algorithm, we perform the undoing actions and restore data items based on the history by *latest write first restore* order. This guarantees that the data item is correctly restored to its original clean value even after multiple updates. Suppose x_t is a dirty data item, the first subtransaction making x_t dirty could be either the malicious subtransaction T_{wm} or an affected subtransaction T_{xp} who writes it earliest. In phase 3, we restore the before images of x_t by reversed timely order - write earliest restore latest. This ensures that the write operation of T_{wm} (or T_{xp}) is restored to before image latest, and that x_t is changed back to its original clean value. Therefore, restoring the before images in backwards order ensures that all dirty data items will be restored to the correct value.

Lemma 8

The proposed repair algorithm is sound, the effects of only malicious or affected subtransactions are repaired.

Proof 8

We need to ensure that unaffected good subtransactions are not changed in repair. We can prove that only the undo-action affected subtransactions and read-write affected subtransactions are undone.

(I) In phase 2, we identify all the undo-action affected subtransactions. Note that not all subtransactions who are neighbors to affected subtransactions are undo-action affected. Only a subset of control-flow dependencies are force-undo-on-undo type, other control-flow dependencies do not require undoing of a subtransaction on the undoing event of another subtransaction. From Observation 1, we know that only the abort, begin-on-commit, and strong commit dependencies have the force-undo-on-undo property. No other dependencies have this property. The proposed repair algorithm undoes only the subtransactions that are undo-action affected. Unaffected subtransactions are not changed.

(II) In phase 2, we find out all read-write affected subtransactions. We scan the log forwards from the start of the earliest malicious advanced transaction to the end of the log. Only the write operations performed by a committed malicious subtransaction or a committed affected subtransaction will be marked as dirty and added to the *corrupted* set. Other read and write operations will not be affected.

Thus, only the effects of made by affected subtransactions will be repaired while the unaffected subtransactions are kept unchanged in repair.

Lemma 9

The proposed repair algorithm respects dependencies, dependency constraints are satisfied during repair.

Proof 9

We assume that the dependency constraints are enforced during normal execution by the scheduler. Here we need to show that the repair algorithm ensures all dependencies if it makes changes on the states of subtransactions during repairing process.

(I) For malicious advanced transactions, the repair algorithm undoes them as a whole. All changes to data items made by them are removed, and the malicious advanced transaction appears like it has not been executed. The dependencies in the malicious advanced transactions are preserved after repair.

(II) For unaffected benign advanced transactions, where no subtransactions is affected, we keep it unchanged and therefore their dependencies are still preserved.

(III) For affected benign advanced transactions, where there are some subtransactions affected, the repair algorithm can ensure the satisfaction of dependencies. In repair, we undo the affected subtransactions, and change the state of these affected subtransactions from *committed* to *initiation* in phase 4. For certain kinds of control-flow dependencies, the undoing action on one affected subtransaction T_{wi} may require to adopt proper actions on other subtransactions T_{wj} in the same advanced transaction.

First we consider the dependencies in form of $T_{wi} \rightarrow_d T_{wj}$. For *event enforcement* type of dependencies, the undoing of T_{wi} will affect T_{wj} if the dependency is an *abort* dependency or a *force-begin-on-abort* dependency. For *event ordering* type of dependencies, the undoing of T_{wi} will affect T_{wj} if the dependencies is a *begin-on-commit* dependency. No other dependencies will affect T_{wj} on undoing actions. For abort dependency and begin-on-commit dependency, according to the actions in Table 13, the $T_{wi} \rightarrow_a T_{wj}$ and $T_{wi} \rightarrow_{bc} T_{wj}$ dependencies require T_{wj} to be undone, if T_{wi} is detected to be an affected subtransaction. For

force-begin-on-abort dependency, if T_{wj} has not yet started in execution, we set its states as 'initiation' and put it in the *toschedule* set, and send it to the scheduler for execution. These actions respect the dependencies. We can see that all these dependencies are ensured during repair.

Next, for dependencies in form of $T_{wj} \rightarrow_d T_{wi}$, similarly, according to Table 14, the *strong commit* dependency is the only dependency that can affect T_{wj} is T_{wi} is undone. The algorithm undoes T_{wj} in this case and it preserves the $T_{wj} \rightarrow_{sc} T_{wi}$ dependency. Therefore, dependency constraints are satisfied during repairing process in this situation also.

From the above arguments, we can conclude that the proposed algorithm can ensure satisfaction of dependencies in repair.

Therefore, we have proved that the repair mechanism described in Section 7.3 ensures effective repair as per Definition 42. The repair is complete, sound, and respects the dependencies.

CHAPTER VIII

REAL-TIME UPDATING OF ACCESS CONTROL POLICIES

Access control policies are security policies that govern access to resources. *Real-time update* of access control policies, that is, updating policies while they are in effect and enforcing the changes immediately, is necessary for many security-critical applications. Although a lot of work appears in the area of security policies, policy updates have received relatively little attention. For existing work with security access control policies, please refer to Damianou's thesis [36] for a survey. There exists work on identifying interesting adaptive policies and formalization of these policies [40, 104]. However, update access control policy and enforce the change immediately is not discussed previously. In this chapter, I consider real-time update of access control policies for database and advanced transaction systems.

8.1 Update of Access Control Policies in Database Systems

To understand the problem better, we first examine real-time updating of access control policies in the traditional database systems. We will then extend the algorithms to the context of advanced transactions.

Access control policies protect information resources from unauthorized access. Since security policies are extremely critical for an enterprise, it is important to control the manner in which policies are updated. Updating policy in an ad-hoc manner may result in inconsistencies and problems with the policy specification; this, in turn, may create other problems, such as, security breaches, unavailability of resources, etc. In other words, policy updates should not be through ad-hoc operations but done through well-defined *transactions* that have been previously analyzed. An important issue that must be kept in mind about policy update transactions is that some policies may require *real-time updates*. We use the term real-time update of a policy to mean that the policy is changed while it is in effect and this change needs to be enforced immediately.

Such real-time updates of access control policies are needed by dynamic environments and

commercial applications. Often times in such scenarios, system resources need reconfiguration or operational modes require change; this, in turn, necessitates policy updates. The updated policies should be automatically enforced. Suppose the user *John*, by virtue of some policy P, has the privilege to execute a long-duration transaction that prints a large volume of sensitive financial information kept in file I. While *John* is executing this transaction, an insider threat is suspected and the policy P is changed such that *John* no longer has the privilege of executing this transaction. Since existing access control mechanisms check *John*'s privileges *before John* initiates the transaction and not *during* the execution of the transaction, the updated policy P will not be correctly enforced causing financial loss to the company.

In this section, we consider real-time policy updates in the context of a traditional database system. A database consists of a set of objects that are accessed and modified through transactions. Transactions performing operations on database objects must have the privilege to execute those operations. Such privileges are specified by access control policies; access control policies are stored in the form of *policy objects*. Transactions executing by virtue of the privileges given by a policy object is said to *deploy* the policy object. In addition to being deployed, a policy object can also be accessed and modified by transactions. We are considering an environment in which different kinds of transactions execute concurrently some of which are policy update transactions. In other words, a policy may be updated while transactions are executing by virtue of this policy. We propose different algorithms that allow for concurrent and real-time updates of policies. The algorithms differ with respect to the degree of concurrency achieved and the kinds of policies that can be updated.

With the algorithms for policy update in database systems, we can easily extend them into advanced transaction systems. In the advanced transaction systems, we need to pay extra attentions to ensure the satisfaction of all dependencies that exist among the subtransaction.

8.1.1 Traditional Database System and Policy Model

A *database* is specified as a collection of objects together with a set of *integrity constraints* on these objects. At any given time, the *state* of the database is determined by the values of the objects in the database. A change in the value of a database object changes the state. Integrity constraints are

predicates defined over the state. A database state is said to be *consistent* if the values of the objects satisfy the given integrity constraints.

A *transaction* is an operation that transforms the database from one consistent state to another. To prevent the database from becoming inconsistent, transactions are the only means by which data objects are accessed and modified. A transaction can be initiated by a user, a group, or another process. A transaction inherits the access privileges of the entity initiating it. A transaction can execute an operation on a database object only if it has the privilege to perform it. Such privileges are specified by access control policies.

In this dissertation, we consider only one kind of access control policies: authorization policies[1]. An authorization policy specifies what operations an entity can perform on another entity. We focus our attention to systems that support positive authorization policies only. This means that the policies only specify what operations an entity is *allowed* to perform on another entity. There is no explicit policy that specifies what operations an entity is *not allowed* to perform on another entity. The absence of an explicit authorization policy authorizing an entity A to perform some operation O on another entity B is interpreted as A not being allowed to perform operation O on entity B.

We begin by considering simple kinds of authorization policies that are specified by *subject*, *object*, and *rights*. A subject can be a user, a group of users or a process. An object, in our model, is a data object, a group of data objects, or an object class. A subject can perform only those operations on the object that are specified in the rights.

Definition 43

[Policy] A *policy* is a function that maps a subject and a object to a set of access rights. We formally denote this as follows: $P : S \times O \rightarrow \mathrm{P}(R)$ where P represents the policy function, S, represents the set of subjects, O represents the set of objects, $\mathrm{P}(R)$ represents the power set of access rights.

In a database, policies are stored in the form of policy objects.

Definition 44

[Policy Object] A policy object P_i consists of the triple $< S_i, O_i, R_i >$ where S_i, O_i, R_i denote the subject, the object, and the access rights of the policy respectively. Subject S_i can perform only those operations on the object O_i that are specified in R_i.

[1] Henceforth, we use the term policy or access control policy to mean authorization policy.

Example 8

Let $P =< John, FileF, \{r, w, x\} >$ be a policy object. This policy object gives subject John the privilege to Read, Write, and Execute $FileF$.

Before proceeding further, we discuss how to represent the access rights. The motivation for this representation will be clear in Section 8.1.3.

Definition 45

[Representing the Access Right of an Object] Let $\mathbf{O_i} = \{o_1, o_2, \ldots, o_n\}$ be the set of all the possible operations that are specified on Object O_i. The set of operations in $\mathbf{O_i}$ are ordered in the form of a sequence $< o_1, o_2, \ldots, o_n >$. We represent any access right on the object O_i as an n-element vector $[i_1 i_2 \ldots i_n]$. In some access right R_j, if the k-th element of this vector is 0, (that is, $i_k = 0$) then R_j does not allow the operation o_k to be performed on the object O_i. When the k-th element equals 1 (that is, $i_k = 1$) in some access right R_m, it signifies that the access right R_m allows operation o_k to be performed on the object O_i. The total number of access rights that can be associated with object O_i equals 2^n.

Example 9

Let $< r,w,x >$ be the operations allowed on a file F. The access right $R_1 = [001]$ signifies that r, w operations are not allowed on the file F but the operation x is permitted on File F. The access right $R_2 = [101]$ allows r and x operations on the file F but does not allow the w operation.

For an object O_i having n operations, the total number of access rights that can be associated with the object equals 2^n.

Definition 46

[Partial Order of Access Rights of an Object] The set of all access rights associated with an object O_i having n operations forms a *partial order* with the ordering relation $\geq_{O_i}$. The ordering relation is defined as follows: Let $R_j[i_k]$ denote the i_k-th element of access right R_j. $R_p \geq_{O_i} R_q$ holds if and only if $R_p[i_k] = R_q[i_k]$ or $R_p[i_k] > R_q[i_k]$, for all $k = 1 \ldots n$.

Definition 47

[Least Upper Bound Operation] Given two access rights R_p and R_q associated with an object O_i having n operations, the *least upper bound* of R_p and R_q, denoted as $lub(R_p, R_q)$ is computed

as follows. For $k = 1 \ldots n$, we compute the i_k-th element of the least upper bound of R_p and R_q: $lub(R_p, R_q)[i_k] = R_p[i_k] \vee R_q[i_k]$. The n-bit vector obtained from the above computation will give us the least upper bound of R_p and R_q.

Definition 48

[Greatest Lower Bound Operation] Given two access rights R_p and R_q associated with an object O_i having n operations, the *greatest lower bound* of R_p and R_q, denoted as $glb(R_p, R_q)$ is computed as follows. For $k = 1 \ldots n$, we compute the i_k-th element of the greatest lower bound of R_p and R_q: $glb(R_p, R_q)[i_k] = R_p[i_k] \wedge R_q[i_k]$. The n-bit vector obtained from the above computation will give the greatest lower bound of R_p and R_q.

Since each pair of access rights associated with an object have a unique least upper bound and a unique greatest lower bound, the access rights of an object can be represented as a lattice.

Example 10

Let $< r,w >$ be the operations allowed on a file F. The set of all access rights on file F can be represented as $[00], [01], [10], [11]$. The least upper bound of [01] and [10] equals [11]. The greatest lower bound of [01] and [10] equals [00].

Definition 49

[Access Rights Lattice of an Object] The set of all possible access rights on an object O_i can be represented as a lattice which we term the *access rights lattice of object* O_i. The notation $ARL(O_i)$ denotes the set of all nodes in the access rights lattice of object O_i.

All possible access control privileges pertaining to an object can be represented as the nodes on the access rights lattice of the object. Each node in the lattice represents a specific access control privilege. The lower bound on this lattice (labeled as Node 0) denotes the absence of any access rights on this object. The upper bound denotes the presence of all the rights; any subject having these rights can perform all the operations on the object. The other points in the lattice denote the intermediate states.

Figure 13(a) shows the possible access rights associated with a file having only two operations: Read and Write. For example, this might indicate the possible access control privileges associated with a file that has only two operations associated with it: read and write. The most significant

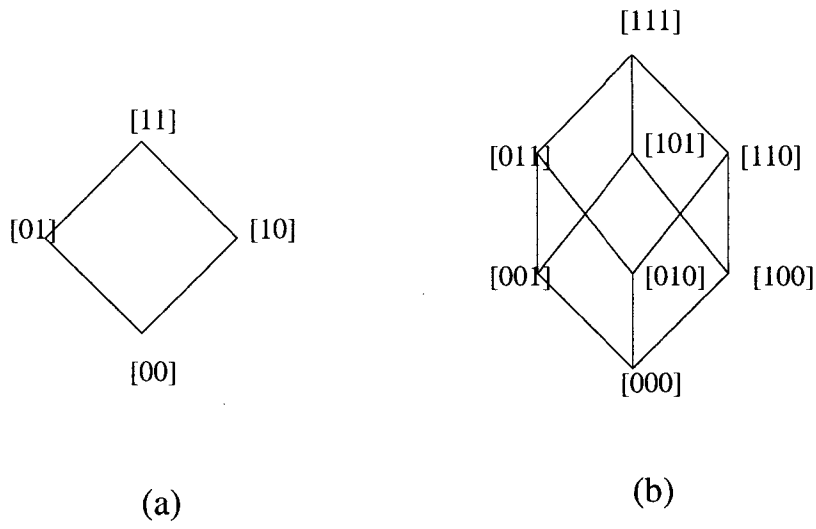

Figure 13: Representing Possible Access Control Rights of Objects

bit denotes the Read operation and the least significant bit denotes the Write operation. The lower bound labeled as Node 00 signifies the absence of Read and Write privilege. The Node 01 signifies that the subject has Write privilege but does not have Read privileges. The Node 10 signifies that the subject has Read privilege but no Write privilege. The Node 11 indicates that the subject has both Read and Write privileges. Figure 13(b) shows the possible access rights associated with an object having three operations.

Next we define a policy in terms of the access rights lattice.

Definition 50

[Policy] A *policy* P_i maps a subject S_i's access privilege to some Node j in the access rights lattice of the object O_i. This is formally stated as follows: $P : S \rightarrow (ARL(O))$.

Definition 51

[Policy Update] A *policy update* is an operation that changes some policy object $P_i =< S_i, O_i, R_i >$ to $P_i' =< S_i, O_i, R_i' >$ where P_i' is obtained by transforming R_i to R_i'. Let R_i, R_i' be mapped to Node j, Node k of $ARL(O_i)$ respectively. The update of policy object P_i changes the mapping of the subject S_i's access privilege from Node j to Node k in the access rights lattice of object O_i.

Having given some background on the policies, we are now in a position to discuss policy objects. Recall from Definition 44 that policies are stored in the database in the form of policy objects. Next we describe the operations associated with the policy objects. Policy objects, like

data objects, can be read and written. However, unlike ordinary data objects, policy objects can also be *deployed.*

Definition 52

[Deploy] A policy object P_j is said to be *deployed* if there exists a subject that is currently accessing an object by virtue of the privileges given by the policy object P_j.

Example 11

Suppose the policy object P_i allows subject S_j to read object O_k. Subject S_j initiates a transaction T_l that reads O_k. While the transaction T_l reads O_k, we say that the policy object P_i is deployed.

The environment we are considering is one in which multiple users will be accessing and modifying data and policy objects, while the policy objects are deployed. To deal with this scenario, we need some concurrency control mechanism. The objectives of our concurrency control mechanism are the following.

- Allow concurrent access to data objects and policy objects.
- Prevent security violations arising due to policy updates.

We now present a simple concurrency control algorithm for updating policies.

8.1.2 A Simple Algorithm for Policy Updates

We assume that each data object is associated with two operations: Read and Write. A policy object is associated with three operations: Read, Write and Deploy. We begin by giving some definitions.

Definition 53

[Conflicting Operations] Two operations are said to *conflict* if both operate on the same data object and one of them is a Write operation.

The Write operation conflicts with a Read or a Deploy operation on the same object.

Definition 54

[Transaction] A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data or policy object }\} \cup \{d_i[x] \mid x \text{ is a policy object }\} \cup \{a_i, c_i\}$;
2. $a_i \in T_i$ iff $c_i \notin T_i$;

3. if t is c_i or a_i, for any other operation $p \in T_i$, $p_i <_i t$; and

4. if $r_i[x]$, $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$.

5. if $d_i[x]$, $w_i[x] \in T_i$, then either $d_i[x] <_i w_i[x]$ or $w_i[x] <_i d_i[x]$.

Condition 1 defines the different kinds of operations in the transactions ($r_i[x]$, $w_i[x]$, $d_i[x]$, a_i, c_i denote Read operation on object x, Write operation on x, Deploy operation on x, Abort or Commit operation respectively). Condition 2 states that this set contains an Abort or a Commit operation but not both. Condition 3 states that Abort or Commit operation must follow every other operation of the transaction. Condition 4 requires that the partial order $<_i$ specify the order of execution of Read and Write operations on a common data or policy object. Condition 5 requires that the partial order $<_i$ specify the order of execution of Deploy and Write operations on a common policy object.

The algorithm that we propose is an extension of the two phase locking protocol [20]. Each data object O_i in our model is associated with two locks: read lock (denoted by $RL(O_i)$) and write lock (denoted by $WL(O_i)$). The locking rules for data objects are the same as the standard two-phase locking protocol [20]. A policy object P_j is associated with three locks: read lock (denoted by $RL(P_j)$), write lock (denoted by $WL(P_j)$) and deploy lock (denoted by $DL(P_j)$).

The locking rules for data objects are the same as the standard two-phase locking protocol [20]. A policy object P_j is associated with three locks: read lock (denoted by $RL(P_j)$), write lock (denoted by $WL(P_j)$) and deploy lock (denoted by $DL(P_j)$).

	Wants		
Has	RL	WL	DL
RL	Yes	No	Yes
WL	No	No	No
DL	Yes	Signal	Yes

Table 15: Locking Rules for Policy Objects

The locking rules for the policy objects are given in Table 15. *Yes* entry in the lock table indicates that the lock request is granted. *No* entry indicates that the lock request is denied. *Signal* entry in the lock table indicates that the lock request is granted, but only after the transaction currently holding the lock is aborted and the lock is released. The first row corresponds to the case where

some transaction has a read lock on a policy object. The first column in the first row is a *Yes* – another transaction requesting a read lock on the same object is given the lock. The second column in the first row is a *No* – another transaction requesting a write lock on the same object is not given the lock. The entry in the first row third column is a *Yes* – this means, that if a transaction has a read lock on a policy object, and another transaction requests a deploy lock on the same object, then this lock request is granted. The second row corresponds to the case where a transaction has a write lock on a policy object. This row has all *No* entries; this means that no other transaction will be given any other locks to the object. Now consider the third row; this row corresponds to a transaction holding a deploy lock on a policy object. The entry in the first and third columns of the third row is a *Yes* signifying that if another transaction wants a read lock or a deploy lock on the object, it is granted. The entry in the second column of the third row is *Signal.* Signal means that the lock request is granted after the transaction currently holding the lock is aborted. For example, suppose some transaction T_i holds a deploy lock DL on a policy object, and another transaction T_j wishes to get the write lock WL and update the policy object. In such a case a signal is generated to abort T_i, after which T_i releases the DL lock and T_j is granted the WL lock. Note that, a transaction updating a policy (T_j) has a higher priority than a transaction (T_i) that deploys this policy; hence, we generate a signal to abort the transaction deploying the policy (T_i).

Next we define what it means for a transaction in our model to be well-formed.

Definition 55

[Well-formed Transaction] A transaction is *well-formed* if it satisfies the following conditions.

1. A transaction before reading or writing a data or policy object must deploy the policy object that authorizes the transaction to perform the operation.

2. A transaction before deploying, reading, or writing a policy object must acquire the appropriate lock.

3. A transaction before reading or writing a data object must acquire the appropriate lock.

4. A transaction cannot acquire a lock on a policy or data object if another transaction has locked the object in a conflicting mode.

5. All locks acquired by the transaction are eventually released.

Definition 56

[Well-formed Two-phase Transaction] A well-formed transaction T_i is *two-phase* if all its lock operations precede any of its unlock operations.

Example 12

Consider a transaction T_i that reads object O_j (denoted by $r_i(O_j)$) and then writes object O_k (denoted by $w_i(O_k)$). Policies P_m and P_n authorize the subject initiating transaction T_i, the privilege to read object O_j and the privilege to write object O_k respectively. An example of a well-formed and two-phase execution of T_i consists of the following sequence of operations: $< DL_i(P_m)$, $RL_i(O_j)$, $d_i(P_m)$, $r_i(O_j)$, $DL_i(P_n)$, $WL_i(O_k)$, $d_i(P_n)$, $w_i(O_k)$, $UL_i(P_m)$, $UL_i(P_n)$, $UL_i(O_j)$, $UL_i(O_k) >$, where DL_i, RL_i, WL_i, d_i, r_i, w_i, UL_i denote the operations of acquiring deploy lock, acquiring read lock, acquiring write lock, deploy, read, write, lock release, respectively, performed by transaction T_i.

Definition 57

[Policy-Compliant Transaction] A transaction is *policy-compliant* if for every operation that a transaction performs, there exists a policy that authorizes the transaction to perform the operation for the entire duration of the operation.

Note that, all transactions may not be policy-compliant. For instance, suppose entity A can execute a long-duration transaction T_i by virtue of policy P_x. While A is executing T_i, P_x changes and no longer allows A to execute T_i. In such a case, if transaction T_i is allowed to continue after P_x has changed, then T_i will not be a policy-compliant transaction.

We borrow the definitions of *history*, *conflict equivalence*, *committed projection of a history*, *serial history*, *serialization graph*, and *conflict serializable history* from Bernstein et al. [20].

Theorem 14

A history is serializable iff its serialization graph is acyclic.

Proof 14

This proof is identical to the proof of the same theorem that is given in Bernstein et al. [20].

Next we define what we mean by a *policy-compliant* history.

Definition 58

[Policy-compliant] A history is *policy-compliant* if all the transactions in the history are policy-compliant transactions.

Theorem 15

A history H in which all the transactions in the committed projection of the history H are well-formed and two-phase is conflict serializable.

Proof 15

We prove this by contradiction. Assume that the history H produced by transactions $\{T_1, T_2, \ldots, T_n\}$ is not serializable. The graph produced from this history contains a cycle. Without loss of generality, assume that this cycle is $T_1 \rightarrow T_2 \rightarrow T_3 \ldots T_n \rightarrow T_1$. The presence of the arrow $T_1 \rightarrow T_2$, signifies that there is an operation in T_1 that conflicts with and precedes another operation in T_2. The unlock operation in T_1 must precede the lock operation in T_2 (this is because the data object involved in a conflicting operation can be locked by only one transaction at any time). That is, $U_1(O_a) < L_2(O_a)$. Using similar arguments, we can argue that for the edge $T_2 \rightarrow T_3$, there is an unlock operation in T_2 that precedes a lock operation in T_3. That is, $U_2(O_b) < L_3(O_b)$. Since transaction T_2 is two-phase, $L_2(O_a) < U_2(O_b)$. Therefore, we can conclude that $U_1(O_a) < L_3(O_b)$. This argument can be extended and we can arrive at the conclusion that $U_1(O_a) < L_1(O_k)$. This is not possible because T_1 is two-phase. Thus, we arrive at a contradiction. Hence, our initial assumption that the history is not serializable is wrong. Therefore, the history H is serializable.

Theorem 16

A history H in which all the transactions in the committed projection are well-formed and two-phase is policy-compliant.

Proof 16

Assume that the history H is not policy-compliant. This means that one or more transactions in the history H are not policy-compliant. Suppose T_i is one such transaction. Without loss of generality, assume that the transaction T_i does not have write access to an object O_r but nevertheless updates object O_r. We show that this cannot happen. Since T_i is well-formed, it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place, T_i has to obtain the deploy lock for the policy object. In other words, before T_i can access O_r, it

has to obtain the deployment lock for a policy P_m that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy P_m that allowed T_i to update O_r. So, the only possibility is that while T_i was updating O_r, the policy P_m got deleted or modified. But according to well-formed rules this is not possible. Any transaction T_j modifying the policy P_m has to obtain a write lock (WL) on P_m. Before the write lock on P_m can be granted, the transaction T_i holding the deploy lock (DL) has to be aborted and the deploy lock released (Table 15). Thus, the above scenario of T_i updating O_r without any policy authorizing T_i to do so, does not arise in our case. Therefore, T_i is policy-compliant. Our assumption, that the history H is not policy-compliant, is wrong.

Although the lock based concurrency control approach provides policy-compliant and serializable schedule, it is overly restrictive. A change of policy may not change the specific subject's access privileges or may result in increased access privileges; in such cases terminating valid access will result in poor performance. This motivates us to propose the following approach.

8.1.3 Concurrency Control using Semantics of Policy Update

In this section we show how we can use semantics of the policy update operation to increase concurrency. The basic idea is to classify a policy update operation either as a *policy relaxation* or as a *policy restriction* operation. Policy relaxation causes increase in subject's access rights; transactions executing by virtue of a policy need not be aborted when the policy is being relaxed. On the other hand, a policy restriction does not increase the access rights of the subject. To ensure policy-compliant transactions, we must abort the transactions that are executing by virtue of the policy that is being restricted. Before going into the details, we first give a definition of policy relaxation and policy restriction.

Recall from Definition 43 that a policy P_i is expressed as a tuple $P_i = < S_i, T_i, R_i >$ where S_i, T_i, and R_i denote the subject, the object, and the the set of access rights. Next, we discuss in more details about the nature of access rights.

Definition 59

[Policy Relaxation Operation] A *policy relaxation operation* is a policy update that increases the access rights of the subject. Let the policy object $P_i =< S_i, O_i, R_i >$ be changed to $P_i' =<$

$S_i, O_i, R_i' >$. Let Let R_i, R_i' be mapped to the nodes k, j respectively in $ARL(O_i)$. A policy update operation is a policy relaxation operation if $lub(k,j) = j$.

Example 13

Let the operations allowed on *FileF* be $< r, w, x >$. Suppose policy $P_i = <$ *John*,*FileF*,[001] $>$ is changed to $P_i' =<$ *John*, *FileF*, [101] $>$. This is an example of policy relaxation because the access rights of subject John has increased. Note that $lub([001],[101]) = [101]$. Thus, this is a policy relaxation.

Definition 60

[Policy Restriction Operation] A *policy restriction operation* is a policy update operation that is not a policy relaxation operation. Let the policy object $P_i =< S_i, O_i, R_i >$ be changed to $P_i' =<$ $S_i, O_i, R_i' >$. Let R_i, R_i' be mapped to the nodes k, j respectively in $ARL(O_i)$. A policy update operation is a policy restriction operation if $lub(k,j) \neq j$.

Example 14

Let the operations allowed on *FileF* be $< r, w, x >$. Suppose policy $P_i = <$ *John*,*FileF*,[001] $>$ is changed to $P_i' =<$ *John*, *FileF*, [110] $>$. This is an example of policy restriction because the access rights of subject John has not increased. Note that, $lub([001],[110]) = [111]$. Since $lub([001],[110]) \neq$ [110], this is an example of policy restriction.

In other words, moving up the lattice along the edges indicate that policy is being relaxed. Moving down the lattice along the edges indicates that policy is being restricted. Moving from one node to another not connected by edges is also considered to be a policy restriction operation.

8.1.3.1 Concurrency Control Based on Knowledge of Policy Change

We now give a concurrency control algorithm that uses the knowledge of the kind of policy change. The operations specified on data objects are Read and Write. A policy object is now associated with four operations: Read, Deploy, WriteRelax, WriteRestrict. The Read and Deploy operations are similar to those specified in Section 8.1.2. The Write operations on policy object are classified as WriteRelax or WriteRestrict. A WriteRelax operation is one in which the policy gets relaxed. All other write operations on the policy object are treated as WriteRestrict. Since the operations

are different than those discussed in Section 8.1.2, we modify the definition of transaction given in Definition 54 to the following:

Definition 61

[Transaction] A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data object } \} \cup \{d_i[x], r_i[x], ws_i[x], wx_i[x] \mid x \text{ is a policy object } \} \cup \{a_i, c_i\}$;

2. $a_i \in T_i$ iff $c_i \notin T_i$;

3. if t is c_i or a_i, for any other operation $p \in T_i$, $p_i <_i t$; and

4. if $r_i[x]$, $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$.

5. if $d_i[x]$, $ws_i[x] \in T_i$, then either $d_i[x] <_i ws_i[x]$ or $ws_i[x] <_i d_i[x]$.

6. if $d_i[x]$, $wx_i[x] \in T_i$, then either $d_i[x] <_i wx_i[x]$ or $wx_i[x] <_i d_i[x]$.

Condition 1 is changed from that in Definition 54 to reflect that the operations allowed on data objects are Read and Write and the operations allowed on policy objects are Read, Deploy, WriteRelax (denoted by wx), and WriteRestrict (denoted by ws). Conditions 2,3, and 4 are the same as given in Definition 54. Condition 5 given in Definition 54 is no longer applicable as there is no simple Write operation on policy objects; this condition is replaced by two conditions (Conditions 5 and 6 in Definition 61). Conditions 5 specifies that if there is a Deploy operation on a policy object and a WriteRestrict operation on the same object, then the ordering relation $<_i$ must specify the order of the operations. Condition 6 specifies a similar condition for Deploy and WriteRelax operation.

The definition of well-formed transaction is changed as follows:

Definition 62

[Well-formed Transaction] A transaction is *well-formed* if it satisfies the following conditions.

1. A transaction before reading or writing a data object must deploy the policy object that authorizes the transaction to perform the operation.

2. A transaction before reading, write relaxing or write restricting a policy object must deploy the policy object that authorizes the transaction to perform the operation.

3. A transaction before reading or writing a data object must acquire the appropriate lock.

4. A transaction before deploying, reading, write relaxing, or write restricting a policy object must acquire the appropriate lock.

5. A transaction cannot acquire a lock on a policy or data object if another transaction has locked the object in a conflicting mode.

6. All locks acquired by the transaction are eventually released.

To ensure serializable and policy-compliant histories, we require each transaction to be well-formed (Def. 62) and two-phase (Def. 56).

We next present our locking mechanism. Each data object O_i is associated with two locks: read lock (denoted by $RL(O_i)$) and write lock (denoted by $WL(O_i)$). The locking rules for data objects are the same as in the two-phase locking protocol [20]. Corresponding to the four operations on the policy object, we have four kinds of locks associated with policy objects: read locks (RL), deploy locks (DL), relax locks (WXL) and restrict locks (WSL). The locking rules are given in the table 16. Let us consider the fourth row; this row corresponds to a transaction holding a deploy lock on a policy object. The entry in the first and fourth columns of the fourth row is a *Yes* signifying that if another transaction wants a read lock or a deploy lock on the object, it is granted. The entry in the second column of the fourth row is a *Yes* – if a transaction has a deploy lock on a policy object and another transaction wants a relax lock on the same object, then the lock request is granted. Thus, if a transaction T_m holds a deploy lock DL on a policy object P_j, and another transaction T_n wants to perform a policy relaxation on P_j, then T_n is granted the WXL lock. Note that T_m does not have to be aborted in this case because the policy P_j that is deployed by T_m is being relaxed. The entry in the third column of the fourth row is *Signal*. This is the case of some transaction T_i holding a deploy lock DL on a policy object, and another transaction T_j wanting to perform an update causing policy restriction. In this scenario, a signal is generated to abort T_i, after which T_i releases the DL lock and T_j is granted the WSL lock.

	Wants			
Has	RL	WXL	WSL	DL
RL	Yes	No	No	Yes
WXL	No	No	No	No
WSL	No	No	No	No
DL	Yes	Yes	Signal	Yes

Table 16: Locking Rules for Policy Objects

Note that our algorithm does not generate well-formed transactions. Specifically, it violates Condition 5 given in Definition 62. Thus, it does not generate conflict-serializable histories. We define an alternate correctness criterion, namely, serializability, and show that our histories do satisfy this new criterion.

Definition 63

[Equivalence of Histories] Two histories H and H' are equivalent if they satisfy the following conditions:

1. they are defined over the same set of transactions,
2. the execution of H on some initial state S results in the same final state S' as the execution of H' on the same initial state S.

Definition 64

[Serializable History] A history H that is equivalent to a serial history H_s is a serializable history.

Theorem 17

Histories generated by the locking rules of Table 16 are serializable.

Proof 17

Our mechanism generates histories that may not be conflict-serializable. Non conflict-serializable execution occurs because the locking rules allow two transactions to hold conflicting locks on a policy object at the same time. Specifically, the rules allow some transaction T_i to hold a deploy lock on a policy object P_q, while another transaction, say, T_j acquires a relax lock on P_q to increase the privileges. This situation is not problematic and the effect of the execution of these two transactions is equivalent to the serial execution of T_i followed by T_j. Any history H containing such transactions T_i and T_j can therefore be converted to an equivalent serial history.

Theorem 18

The locking rules provided in Table 16 provide more concurrency than those given in Table 15.

Proof 18

Let **H1** and **H2** be the set of all possible histories generated by the locking rules given in Table 15 and Table 16 respectively. We need to prove that **H1** is a proper subset of **H2**, that is, **H1** $\subset$ **H2**. The proof will proceed in two parts: (i) First, we will prove that for any history H_1, if $H_1 \in$ **H1**, then $H_1 \in$ **H2**. (ii) Next, we will prove that for some history H_2 where $H_2 \in$ **H2**, $H_2 \notin$ **H1**. In the following paragraphs we outline the two proofs.

Proof of (i): Let $H_1 \in$ **H1**. We need to prove that $H_1 \in$ **H2**. Assume that $H_1 \notin$ **H2**. This is possible only if there is some operation in H_1 that cannot be scheduled by locking rules of Table 16. In other words, the locking rules of Table 16 prohibit obtaining locks necessary for this operation. For ordinary data objects, the locking rules are the same for both the approaches. So the only possibility is that this operation is a read, write or deploy operation on a policy object. Suppose this is a read operation. The locking rule prevents read locks only when some other prior transaction has already acquired some kind of write lock on the object. But in this case the locking rule of Table 15 would have also disallowed the read lock and hence the read operation in H_1. Thus, the operation is not a read operation. Similar arguments can be made for deploy or write operations. Hence any operation in H_1 will also be permitted by the locking rules of Table 16. In other words $H_1 \in$ **H2**.

Proof of (ii): We need to show that for some $H_2 \in$ **H2**, $H_2 \notin$ **H1**. Let $H_2 = < d_i(P_x), r_i(O_a), d_j(P_y), wx_j(P_x), w_i(O_b), c_i, c_j >$. H_2 is a history generated by interleaving the operations of two transactions T_i and T_j, where $T_i =< d_i(P_x), r_i(O_u), w_i(O_v), c_i >$ and $T_j =< d_j(P_y), wx_j(P_x), c_j >$. T_i reads and writes the data objects O_u and O_v respectively. T_i executes by virtue of policy P_x. T_j updates policy P_x by relaxing it. It performs a policy relaxation operation (indicated by wx). T_j executes due to the privileges given by policy P_y. Since the history H_2 can be generated by the locking rules given in Table 16, $H_2 \in$ **H2**. However, H_2 cannot be generated by the locking rules given in Table 15. This is because before the operation wx_j can take place, the locking protocol must obtain the WL on P_x. This will necessitate generating a Signal lock that will abort transaction T_i. Therefore, $H_2 \notin$ **H1**.

Theorem 19

Any history H generated by this concurrency control mechanism is policy-compliant.

Proof 19

Assume that the history H is not policy-compliant. This means that one or more transaction in the history H is not policy-compliant. Suppose transaction T_i is not policy-compliant. Without loss of generality, assume that the transaction T_i does not have write access to an object O_r but nevertheless updates object O_r. We show that this cannot happen. Since T_i satisfies all but Condition 4 of the Definition 55 , it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place, T_i has to obtain the deploy lock for the policy object. In other words, before T_i can access O_r, it has to obtain the deploy lock for a policy P_m that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy P_m that allowed T_i to update O_r. So, the only possibility is that while T_i was updating O_r, the policy P_m got modified such that T_i no longer has the privilege to update O_r. But this scenario cannot occur according to our algorithm. Any transaction T_j modifying the policy P_m has to obtain a write lock (WL) on P_m. Before the write lock on P_m can be granted, the transaction T_i holding the deploy lock (DL) has to be aborted and the deploy lock released (because T_j restricts the policy). Thus, the above scenario of T_i updating O_r without any policy authorizing T_i to do so, does not arise in our case. Therefore, T_i is policy-compliant. Our assumption, that the history H is not policy-compliant, is wrong.

8.1.3.2 Creation and Deletion of Policies

We consider a system having positive authorization policies only. The absence of any policy on a specified subject S_i and object O_j indicates that S_i cannot perform any operation on O_j.

Theorem 20

Creating a new policy $P_n =< S_i, O_j, R_i >$ corresponds to a policy relaxation operation.

Proof 20

The absence of any policy on subject S_i and object O_j can be represented as the existence of a dummy policy P_d that maps the subject S_i to the minimal element in the access rights lattice of the object O_j (indicated by Node 0). The introduction of a new policy P_n over this subject and object can be viewed as an update of the policy P_d to the new policy P_n. Let the access rights specified

by policy object P_n correspond to Node n in $ARL(O_j)$. In this case, $lub(0,n) = n$. Hence, this is an example of policy relaxation.

Thus, when a transaction creates a policy object, it acquires a write relax lock on the policy object. The transaction after completing relaxes this lock. Since no other operations can be performed on the policy object before it is created, no other transactions will have any lock on this policy object when the policy creation transaction executes. Thus, when a new policy is being added, existing transactions are not affected.

Theorem 21

Deleting an existing policy $P_e =< S_i, O_j, R_i >$ can be thought of as a policy restriction operation.

Proof 21

Deletion of an existing policy P_e specified over the subject S_i and object O_j can be thought of as modifying the existing policy P_e to the dummy policy P_d which maps S_i to Node 0 in $ARL(O_i)$. In this case, $lub(e,0) \neq 0$; hence, this is policy restriction.

The transaction, say T_i, that deletes the policy object P_e must acquire a write restrict lock on this policy object. Other transactions may be executing by virtue of this policy object P_e. Such transactions are aborted when T_i acquires the write restrict lock on P_e. In short, when a policy is deleted, transactions executing by virtue of this policy is aborted by the concurrency control mechanism.

8.1.4 Multiple Policies Specified over a Subject and an Object

In many situations multiple policies may be specified over a subject and an object. The access rights given by these policies may differ. Specification of multiple policies complicates the problem of policy updates. This is because an update of a policy not only impacts transactions executing by virtue of that policy, but may also impact transactions that are executing by virtue of other policies defined over the same subject and object.

When different policies are specified over a given subject and an object, how do we evaluate what rights the subject has on the object. These different policies are combined to obtain the *resultant policy*. There may be different strategies for combining the policies. In this paper, we assume

that the rights pertaining to the resultant policy is the sum total of access rights given by the individual policies. Note that, there is no policy object corresponding to the resultant policy. To get all the privileges of the resultant policy, all the individual policy objects used to compute the resultant policy must be deployed. To get the privilege of an individual policy, only that individual policy needs to be deployed.

Definition 65

[Resultant Policy] Let $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$ be the set of policies defined over the same subject S and object O where $P_i =< S, O, R_i >$ for $1 \leq i \leq n$. Let $P_r =< S, O, R_r >$ be the resultant policy. The access right of the resultant policy is computed as follows: $R_r = lub\{R_1, R_2, \ldots, R_n\}$ where *lub* is the least upper bound operator applied on the rights of the individual policies represented in the bit vector form.

8.1.4.1 Specifying Priorities with Policies

When different policies are specified over a given subject and an object, we may not want all the policies to be enforced all the times. In such cases we may associate priorities with policies. The understanding is that the policy with the highest priority is the one that must be deployed; this policy determines the rights a subject has on the object.

Priorities can be specified in two ways: relative or absolute priorities. In relative specification, the policies are ordered according to their priorities. In absolute specification, the set of policies is fixed and any policy can have a priority value from this set. We assume the latter and each policy has a priority value taken from the set of priorities.

Definition 66

[Policy Object with Priority] A policy object with priority is defined as a quadruple $P_i =< S_i, O_i, R_i, pr_i >$ where S_i, O_i, R_i, pr_i denote the subject, object, access rights, and priority of policy P_i respectively. pr_i can take any value from the set of priorities defined below.

Definition 67

[Priority Set] Each application has a priority set PR that defines the domain of priorities. That is, it contains all the possible values that can be taken by the priority of any policy in the application. The elements in the set are totally ordered and the ordering relation is denoted by $<$. Consider two

policies P_i and P_j having priorities pr_i and pr_j respectively. $pr_i < pr_j$ signifies that policy P_i has a lesser priority than policy P_j or P_j has a higher priority than P_i. Since the elements of this set can be totally ordered, this set can be represented as a lattice which we term *priority lattice.*

Example 15

Suppose in an organization the set of priorities PR is defined as follows $PR = \{Low, High\}$. Thus, the priority for a policy can take on either of the two values *Low* or *High*.

Let us once again consider the access rights associated with each object. For an object O_i with n possible operations, there are 2^n possible access rights where each access right is represented as per Definition 45. If there are l elements in the priority set PR, then there are $l * 2^n$ possible priority and access rights pairings. These $l * 2^n$ pairs form a partially ordered set as defined below.

Definition 68

[Partial Order of (Access Right, Priority) Pairings of an Object] The set of all ordered pairs of access rights and priorities associated with an object O_i forms a partial order with the ordering relation $\preceq$. Each element e_i in this set is represented as an ordered pair (l_i, r_i) where l_i denotes the priority level and r_i denotes the access right represented according to Definition 45. For any two elements, (l_i, r_i) and (l_j, r_j), if $(l_i, r_i) \preceq (l_j, r_j)$, then the following condition holds: $l_i \leq l_j$ and $r_i \wedge r_j = r_i$ where $l_i \leq l_j$ signifies that either $l_i < l_j$ or $l_i = l_j$.

We next define least upper bound and greatest lower bound operations on this set.

Definition 69

[Least Upper Bound Operation] Given two elements (l_i, r_i) and (l_j, r_j) where each element describes the priority, access right of some object O, the *least upper bound* of (l_i, r_i) and (l_j, r_j) is computed as follows:

$lub((l_i, r_i), (l_j, r_j)) = (l_k, r_i \vee r_j)$ where $l_k = max(l_i, l_j)$ where $max(l_i, l_j)$ indicates the priority that is higher in the set $\{l_i, l_j\}$.

Definition 70

[Greatest Lower Bound Operation] Given two elements (l_i, r_i) and (l_j, r_j) where each element describes the priority, access right of some object O, the *greatest lower bound* of (l_i, r_i) and (l_j, r_j) is computed as follows:

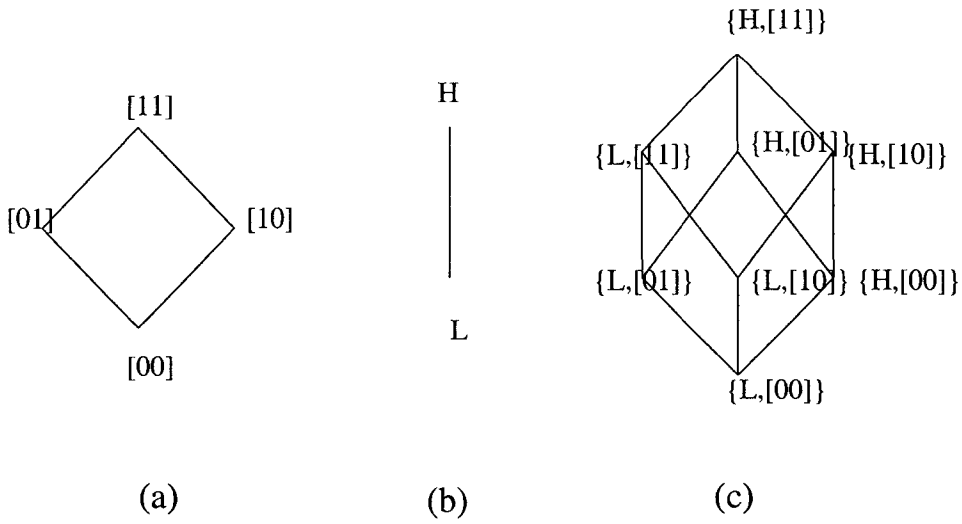

Figure 14: Access Control Rights Lattice of O_i, Priority Lattice PR, and $PARL(O_i, PR)$

$glb((l_i, r_i), (l_j, r_j)) = (l_k, r_i \wedge r_j)$ where $l_k = min(l_i, l_j)$ where $min(l_i, l_j)$ indicates the priority that is lower in the set $\{l_i, l_j\}$.

Since each pair of elements (where each element is an ordered pair of priority and access right) associated with an object has a unique least upper bound and a unique greatest lower bound, the priority- access right pairs of an object can be represented as a lattice.

Definition 71

[Priority Access Right Lattice of an Object] The set of all possible priority-access right pairs of an object O_i and priority set PR can be represented as a lattice which we term *priority access right lattice of object* O_i. The set of all elements in the priority access rights lattice of an object O_i is denoted as $PARL(O_i, PR)$.

Example 16

Consider the example shown in Figure 14. Figure 14(a) shows the access rights lattice of an object O_i having two operations. Figure 14(b) shows the priority lattice PR for the case where there are only two priorities L and H. Figure 14(c) shows the priority access right lattice $PARL(O_i, PR)$ obtained from combining these two lattices.

Definition 72

[Policy with Priority] A policy $P_i =< S_i, O_i, R_i, pr_i >$ maps a subject S_i to a node in the priority access right lattice of object O_i. This is formally stated as follows: $P : S \mapsto (PARL(O, PR))$ where

P is the policy function, S is the set of subjects, $PARL(O, PR)$ denotes the set of all priority-access right pairs for all objects.

When priorities are specified on policies, not all policies can be deployed at a given instant of time. We next define the notion of deployable policies. Stated informally, the set of deployable policies are those having the highest priority.

Definition 73

[Deployable Policies] Let $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$ be the set of policies defined over the same subject S and object O where $P_i =< S, O, R_i, pr_i >$ for $1 \leq i \leq n$. Let $pr_{max} = max(pr_1, pr_2, \ldots, pr_n)$ where max equals maximum of the set of priorities of policies in **P**. The set of deployable policies **D** satisfies the following conditions:

1. $\mathbf{D} \subseteq \mathbf{P}$

2. $\forall P_i \in \mathbf{D}$, $pr_i = pr_{max}$

Recall from Condition 1 of Definition 62, that each transaction before reading or writing a data object must deploy the policy object before performing the authorization. When priorities are specified on policies, a transaction can only deploy deployable policies. That is, a transaction can obtain a deploy lock on a policy object only if it is deployable.

The maximum privilege that a transaction can have depends on the resultant policy. Recall that the resultant policy specifies the total privileges that a subject has on an object. When priorities are specified on policies, the computation of the resultant policy changes. This is because the resultant policy is computed on the set of deployable policies.

Definition 74

[Resultant Policy] Let $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$ be the set of policies defined over the same subject S and object O where $P_i =< S, O, R_i, pr_i >$ for $1 \leq i \leq n$. Let **D** be the set of policies in **P** which can be deployed where $\mathbf{D} = \{P_{d1}, P_{d2}, \ldots, P_{dm}\}$. Let $P_r =< S, O, R_r, pr_r >$ be the resultant policy. $pr_r = pr_{d1}$ where pr_{d1} is the priority of some deployable policy P_{d1}. $R_r = lub(R_{d1}, R_{d2}, \ldots, R_{dm})$.

Example 17

Let $P_i =< S, O, 01, H >$ and $P_j =< S, O, 11, L >$. The resultant policy $P_r =< S, O, 01, H >$. In this case only P_i is the deployable policy.

Definition 75

[Policy Update] A policy update is an operation that changes some policy $P_i =< S_i, O_i, R_i, pr_i >$ to P_i' where P_i' is obtained by changing R_i to R_i' and/or by changing pr_i to pr_i'. Let (pr_i, R_i), and (pr_i', R_i') be mapped to Node i, Node i' of $PARL(O_i, PR)$ respectively. The update of policy object P_i changes the mapping of the subject S_i's access privilege from Node i to Node i' in the priority access rights lattice of object O_i.

Definition 76

[Policy Relaxation Operation] A *policy relaxation operation* is a policy update that increases the access rights of the subject and/or increases the priority of the policy. Let the policy object $P_i =< S_i, O_i, R_i, pr_i >$ be changed to $P_i' =< S_i, O_i, R_i', pr_i' >$. Let (pr_i, R_i), (pr_i', R_i') be mapped to the nodes k, j respectively in $PARL(O_i, PR)$. A policy update operation is a policy relaxation operation if $lub(k,j) = j$.

Since only policies with high priorities can be deployed, increasing the priority of the policy may cause the policy to become deployable which results in increased access privilege of some subject.

Definition 77

[Policy Restriction Operation] A *policy restriction operation* is a policy update operation that is not a policy relaxation operation. Let the policy object $P_i =< S_i, O_i, R_i, pr_i >$ be changed to $P_i' =< S_i, O_i, R_i', pr_i' >$. Let (pr_i, R_i), (pr_i', R_i') be mapped to the nodes k, j respectively in $PARL(O_i, PR)$. A policy update operation is a policy restriction operation if $lub(k,j) \neq j$.

Note that, when a policy's priority is decreased, it may no longer be enforced if this is not the highest priority policy in effect. Thus, the privilege that a subject enjoys by virtue of some policy may be diminished when the policy's priority is decreased. Hence, decreasing the priority of a policy corresponds to policy restriction.

8.1.4.2 Concurrency Control Mechanism with Multiple Policies

When multiple policies are specified over a subject and an object, updating one policy has certain side effects. The modification of a policy not only impacts transactions executing due to that policy but also transactions that are executing by the privileges given by other policies specified over

the same subject and object. An example will help illustrate this point. Suppose a transaction T_p is executing by virtue of policy $P_i =< S, O, 010, Low >$. Consider another policy $P_j =< S, O, 001, Low >$ where the priority of P_j is changed from *Low* to *High*. In such a situation since P_i is no longer a deployable policy, T_p must be aborted.

In short, a policy restriction operation causes transactions executing by virtue of that policy to be aborted. A policy relaxation does not require transactions executing by virtue of it to be aborted. However, if the policy relaxation operation changes the priority of the policy to be greater than the priorities of the deployed policies defined over the same subject and object, then the transactions executing by virtue of these other policies must be aborted.

The above situation is complex. Whenever the priority of a deployed policy is increased, the transactions executing by virtue of the unchanged policies on the same subject and object get affected. Our previous locking mechanisms will no longer work. In our previous mechanisms, when a policy was relaxed no transactions executing by virtue of this policy was affected. Now a policy relaxation also affects transactions executing by virtue of other policies.

For our mechanism, we use the same kinds of locks as specified in Section 8.1.3.1. The locking rules are given in Table 16. The major change is when a deployed policy's priority is increased, we must obtain a write restrict lock on all the other deployed policies specified over the same subject and object. Obtaining a write restrict lock causes all transactions holding the deploy lock to abort. The algorithm which we use to update policy is formalized below.

Algorithm 20

Update policy

Input: (i) Policy $P_i =< S_i, O_i, R_i, pr_i >$ that is to be updated, (ii) R_i' – the new rights of P_i, (iii) pr_i' – the new priority of P_i, (iv) **P** – the set of policies defined over S_i and O_i, and (v) **D** – the set of deployable policies defined over S_i and O_i

Output: (i) Updated policy P_i', (ii) **P'** – the new set of policies defined over S_i and O_i, and (iii) **D'** – the new set of deployable policies

Procedure UpdatePolicy(P_i, R_i', pr_i', **P**, **D**)

begin

$\mathbf{D'} = \mathbf{D}$

if $lub\{(pr_i, R_i), (pr_i', R_i')\} = (pr_i', R_i')$ /* policy relaxation */

begin

acquire $WXL(P_i)$

$P_i' =< S_i, O_i, R_i', pr_i' >$ /* modify P_i */

release $WXL(P_i)$

$\mathbf{P}' = \mathbf{P} \cup \{P_i'\} - \{P_i\}$

if $P_d \in \mathbf{D} \wedge P_d \neq P_i$

if $pr_i' > pr_d$

/* new priority greater than that of existing deployable ones */

for each $P_j \in \mathbf{D} \wedge P_j \neq P_i$

acquire $WSL(P_j)$ /* will abort transactions executing by virtue of P_j*/

release $WSL(P_j)$

$\mathbf{D}' = \{P_i'\}$

else if $pr_i' = pr_d \wedge pr_i < pr_d$

/* new priority lesser than that of existing deployable ones */

$\mathbf{D}' = \mathbf{D} \cup \{P_i'\}$

end

else /* policy restriction */

begin

acquire $WSL(P_i)$

$P_i' =< S_i, O_i, R_i', pr_i' >$ /* modify P_i */

release $WSL(P_i)$

$\mathbf{P}' = \mathbf{P} \cup \{P_i'\} - \{P_i\}$

if $P_d \in \mathbf{D} \wedge P_d \neq P_i \wedge pr_i' < pr_d \wedge pr_i = pr_d$

/* new priority lesser than that of deployable ones */

$\mathbf{D}' = \mathbf{D} - \{P_i\}$

if $\mathbf{D} = \{P_i\}$ /* if P_i was the only deployable policy */

/* Calculate the new set of deployable policies */

$\mathbf{D}' = \{\}$

pr_{max} = max_priority of policies in $\mathbf{P'}$

for each $P_m \in \mathbf{P}$

if $pr_m = pr_{max}$

$\mathbf{D'} = \mathbf{D'} \cup P_m$

end

end

The algorithm begins with initializing the new set of deployable policies $\mathbf{D'}$ with the existing set of deployable policies $\mathbf{D}$. This is the default case. Some operations may require a change in the set of deployable policies. We will explain these as and when the cases arise. We then check whether the update of policy P_i is a relaxation or not. Based on this test, we divide the main body of the algorithm into two parts: one for policy relaxation and the other for policy restriction.

For policy relaxation, we obtain the write relax lock on the policy P_i and update it. The lock is released when the policy update transaction terminates. The new set of policies defined over the subject S_i and O_i, denoted by $\mathbf{P'}$, is computed by removing the old policy P_i and including the new one P_i'. Policy relaxation may increase the priority of P_i. In such cases, we need to decide the fate of existing deployed policies. The actions to be taken depend on the priority pr_i of the original policy P_i as well as the priority pr_i' of the modified policy P_i'. The first case is when the priority pr_i is increased to a value that is greater than that of the existing deployable policies. In this case, the new set of deployable policies $\mathbf{D'}$ consists of P_i only. The original set of deployable policies $\mathbf{D}$ is no longer deployable. Transactions executing on behalf of the policies in $\mathbf{D}$ need to be aborted. To achieve this, we obtain a write restrict lock on all the policies in $\mathbf{D}$. This will cause transactions holding deploy locks on these policies to be aborted. The second case is when the old priority pr_i of P_i is less than that of the deployable policies and the new priority pr_i' is equal to that of the deployable policies. In this case P_i' needs to be included in the new set of deployable policies $\mathbf{D'}$.

Next, we consider the case when the operation is a policy restriction. For policy restriction, a write restrict lock is acquired on P_i. This causes transactions executing by virtue of P_i to abort. After the transaction terminates, this lock is released. The new set of policies $\mathbf{P'}$ defined over

subject S_i and object O_i is computed by removing P_i from the set **P** and including P_i'. Here again, the change of priority may cause a change in the set of deployable policies. The first case is when the original priority pr_i is the same as that of the existing deployable policies and the new one pr_i' is less than that. In this case P_i is removed from the new set of deployable policies **D'**. There may also be a case when P_i is the only deployable policy. In such a case reducing the priority of P_i requires re-evaluating the new set of deployable policies **D'**. To find this set **D'**, we first compute pr_{max} which is the maximum priority of the policies in the set **P'** and then insert in the set **D'** the policies in **P'** that have the same priority as pr_{max}.

We now prove some desirable properties of our mechanism.

Theorem 22

Any history H generated by this mechanism is serializable.

Proof 22

This can be proved in a manner that is similar to the proof of Theorem 17.

Theorem 23

Any history H generated by this mechanism is policy-compliant.

Proof 23

Assume that the history H is not policy-compliant. This means that one or more transactions in the history H is not policy-compliant. Suppose transaction T_i is one such transaction that is not policy-compliant. Without loss of generality, assume that transaction T_i does not have write access to an object O_r but nevertheless updates object O_r. We show that this cannot happen. Since T_i satisfies condition 1 of Definition 62, it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place, T_i has to obtain the deploy lock for a policy P_m that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy P_m that allowed T_i to update O_r. One possibility is that while T_i was updating O_r, the policy P_m got deleted or modified such that T_i no longer has the privilege to update O_r. But this scenario cannot occur according to our algorithm. Any transaction T_j restricting the policy P_m has to obtain a write restrict lock (WSL) on P_m. Before the write restrict lock on P_m can be granted, the transaction T_i holding the deploy lock (DL) has to be aborted and the deploy lock released (because T_j restricts the policy). Thus, the above scenario of T_i updating O_r without any

policy authorizing T_i to do so, does not arise in our case. The other possibility is that while T_i was executing, the policy P_m was not changed, but it became an undeployable policy because some other deployable policy P_n increased its priority. This is also not possible by our algorithm because whenever P_n's priority is increased to be higher than P_m's, the policy update algorithm acquires a write-restrict lock on P_m which causes transactions deploying P_m to be aborted. Thus, the situation of T_i updating O_r without the authorization of any policy object, does not arise in our case. In other words, T_i is policy-compliant. We can make a similar case for all other transactions. Thus, H is policy-compliant.

8.1.4.3 Addition and Deletion of Policies

We assume that the absence of any policy on a specified subject S_i and object O_j indicates that S_i cannot perform any operation on Object O_j. When no policies are specified for a given subject and object, and a new policy is being added, then this corresponds to policy relaxation. The rationale is similar to that specified in in Section 8.1.3.2. When there is only one policy specified over a given subject and object and this sole policy is deleted, it corresponds to policy restriction.

However, if there are multiple policies specified on a subject and object, then addition of a new policy may be a restriction or relaxation depending on the interpretation of the policy. Similarly, deletion of a policy can be relaxation or restriction.

Theorem 24

Adding an additional new policy $P_i =< S_i, O_i, R_i, pr_i >$ defined over subject S_i and object O_i corresponds to a policy relaxation operation if $pr_i = pr_d$ where pr_d is the priority of deployed policies defined over S_i and O_i.

Proof 24

Let the access rights given by the existing policies on subject S_i to object O_i be R_k. When the new policy is combined, the access rights = $R_k \vee R_i$. Let the access right R_k and $R_k \vee R_i$ be mapped to node s and t on $ARL(O_i)$. Since $lub(s,t) = t$, the addition of a new policy corresponds to policy relaxation.

Theorem 25

Deleting an existing policy $P_i =< S_i, O_i, R_i, I_i >$ defined over subject S_i and object O_i with $I_i = L$ corresponds to a policy restriction operation.

Proof 25

Let the access rights given by the existing policies on subject S_i to object O_i be $R_k \vee R_i$. When a policy giving the access privilege of R_i is deleted, the access rights = R_k. Let the access right $R_k \vee R_i$ and R_k be mapped to node s and t on $ARL(O_i)$. Since $lub(s,t) \neq t$, the deletion of policy P_i corresponds to policy restriction.

Theorem 26

Deleting an existing policy $P_i =< S_i, O_i, R_i, I_i >$ defined over subject S_i and object O_i with interpretation = 'M', which is not the sole policy specified on S_i and O_i corresponds to a policy relaxation operation.

Proof 26

Let the access rights given by the existing policies on subject S_i to object O_i be $R_k \wedge R_i$. When a policy giving the access privilege of R_i is deleted, the access rights = R_k. Let the access right $R_k \wedge R_i$ and R_k be mapped to node s and t on $ARL(O_i)$. Since $lub(s,t) = t$, the deletion of policy P_i corresponds to policy relaxation.

Theorem 27

Adding an additional new policy $P_i =< S_i, O_i, R_i, I_i >$ defined over subject S_i and object O_i with interpretation = 'M' corresponds to a policy restriction operation.

Proof 27

Let the access rights given by the existing policies on subject S_i to object O_i be R_k. When the new policy is combined, the access rights = $R_k \vee R_i$. Let the access right R_k and $R_k \wedge R_i$ be mapped to node s and t on $ARL(O_i)$. Since $lub(s,t) \neq t$, the addition of a new policy corresponds to policy restriction. This causes transactions executing by virtue of the original policies to be aborted.

8.2 Implementation of the Policy Update Algorithm

In this section, we will discuss how to implement the algorithms proposed in previous section for real-time access control policy update.

8.2.1 Data Structures for Implementing Policy Updates

A diagram showing the data structures used in implementing the policy update algorithm is shown in Figure 15.

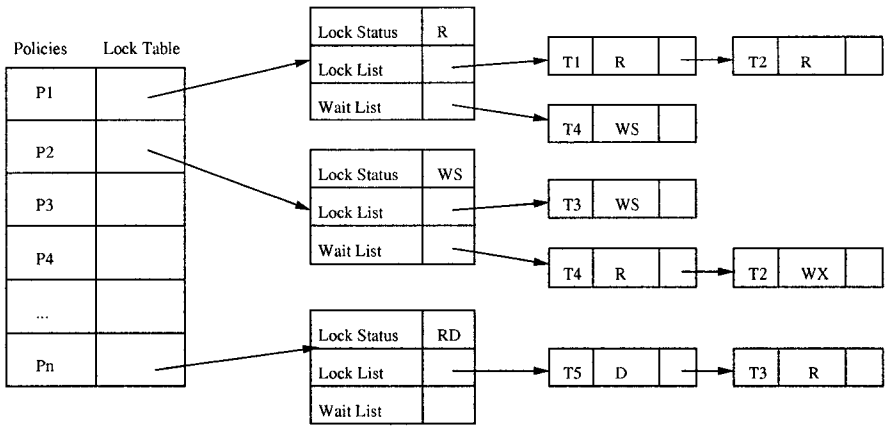

Figure 15: Storing Lock Information for Policy Objects

Policy Object Table This is a hash table containing entries corresponding to each locked policy object and a pointer to the address where all the lock information of this policy object is stored. In other words, each entry in this hash table consists of two fields: the first one stores the policy object id and the second one points to the Lock Information Table of the corresponding policy object. The size of this hash table corresponds to the number of policy objects that are locked.

Lock Information Table This table contains all the lock information pertaining to a policy object. It contains an entry for lock status, a pointer to lock list, and a wait list. The lock status specifies what modes of locks are currently held on the policy object. The lock list specifies the list of transactions that are currently holding locks on the policy object, and the modes of lock they have on the policy object. The lock list can be implemented as a linked list or hash table. Note that, the lock list also contains the lock status information. The lock status information is used frequently. Traversing the lock list and summarizing the lock status information is time-consuming; hence, we keep this information in a separate field. The wait list specifies the list of transactions that are waiting to lock the policy object and the type of lock they want on the policy object. The wait list can be implemented as a priority queue.

In our algorithm, a policy object is associated with four kinds of locks, namely, read, deploy, relax and restrict, that correspond to Read, Deploy, WriteRelax and WriteRestrict operation performed on policy objects. Some of these locks may be held concurrently and others must be held

exclusively. The lock status field in the lock information table specifies what locks are currently held on a policy object. The lock status field can take on any of the following values: **D**: all the locks held on that policy object are deploy locks; **R**: all the locks held on that policy object are read locks; **RD**: all the locks held on that policy object are read and deploy locks; **WX**: the lock held on that policy object is a relax lock; **WS**: the lock held on that policy object is a restrict lock; and, **DWX**: the locks held on that policy object are relax and deploy locks.

When multiple transactions are waiting to lock a policy object, these transactions are inserted into a wait list. Note that, different transactions have different kinds of priority in our system. For critical applications such as the military, policy updates may have a higher priority than transactions deploying the policy. For this reason, when transactions are inserted into a waiting list we arrange them in the order of their priorities. The next transaction that is given the lock from this wait list is the one with the highest priority. In short, we implement this wait list using a priority queue.

8.2.2 Algorithms for Implementing Policy Updates

Algorithm 21

Request a lock on policy object

Input: (i) t: the transaction requesting a lock, (ii) tp: the priority of the transaction requesting the lock, (iii) p: the policy object for which lock is requested, (iv) lm: the lock mode that requested by the transaction, and (v) ht: hash table storing information about policy object.

Output: TRUE/FALSE: indicates whether the request was granted or denied.

Procedure RequestPolicyLock(t, tp, p, lm, ht)

begin

```
if there is no entry for p in ht
   lt = create(p, t, lm);            /* create a lock table for p */
   insert(ht, p, &lt);               /* add entry for p in hash table */
   return TRUE;
lt = getLockTable(ht, p);
if (lt.lockStatus == 'R' or lt.lockStatus == 'RD') // p has only read or read and deploy locks
   if (lm == 'D') and (lt.lockStatus ≠ lm)
      lt.lockStatus = 'RD';
```

```
    if (lm == 'R' or lm == 'D')
      insert(lt.LockList, t);        /* insert t in the LockList */
      return TRUE;
    if (lm == 'WX' or lm == 'WS')
      insert(lt.WaitList, t, tp);    /* insert t in the WaitList */
      return FALSE;
  else (lt.lockStatus == 'D' )       /* p has deploy locks */
    if (lm == 'R')
      lt.lockStatus == 'RD'
    if (lm == 'WX')
      lt.lockStatus == 'DWX'
    if (lm == 'WS' )
      for each transaction T_i in lt.LockList
          abort T_i;                 /* release all deploy locks */
      delete(lt.lockList);
      lt.lockStatus = 'WS';
    insert(lt.LockList, t);          /* insert t in LockList */
    return TRUE;
  else                               /* policy p has a restrict or relax lock */
      insert(lt.Waitlist, t, tp);    /* insert t in WaitList */
      return FALSE;
end
```

Below is the algorithm for releasing a lock for policy object:

Algorithm 22

Release a lock on policy object

Input: (i) t - the transaction that want to release a lock and (ii) p - the policy object for which lock is being released.

Output: TRUE/FALSE - indicates whether the release was successful or not.

```
Procedure ReleasePolicyLock(t, p)
begin
    if (p is not in ht)            /* indicates p is not locked */
      return(FALSE);
    lt = getLockTable(ht, p);
    remove(lt.LockList, t);              /* remove t from LockList */
    if (lt.LockList == NULL)       /* if no other transactions have a lock on p */
      if (lt.WaitList == NULL)     /* if no other transactions are waiting to lock p */
        remove(ht, p);
      else
          request lock for next transaction in wait list
    else                           /* lock list is not empty */
                  /* update lock status for the policy object */
        read = false; deploy = false; relax = false
        for each transaction T_i in lt.LockList do
            if T_i.mode == 'R'
              read = true;
            else if T_i.mode == 'D'
                  deploy = true;
            else if T_i.mode == 'WX'
                  relax = true;
        if (read == true) and (deploy == false) and (relax == false)
          lt.LockStatus = 'R';
        else if (read == true) and (deploy == true) and (relax == false)
              lt.LockStatus = 'RD';
        else if (read == false) and (deploy == true) and (relax == false)
              lt.LockStatus = 'D';
        else if (read == false) and (deploy == true) and (relax == true)
```

lt.LockStatus = 'DWX';

else if (read == false) **and** (deploy == false) **and** (relax == true)

lt.LockStatus = 'WX';

return(TRUE);

end

8.3 Updates of Access Control Policies in Advanced Transactions

In this section, we consider real-time update of access control policies in advanced transaction systems. With the algorithms for policy update in database systems, we can easily extend them into advanced transaction systems. Policy updating in advanced transaction systems is more complicated than in traditional database systems. For example, during a policy restriction update, the execution of this subtransaction must terminate immediately and this subtransaction must abort. This strategy works fine with traditional database systems. However, the advanced transactions are different – subtransactions in advanced transactions are co-operative and their executions are coordinated by dependencies. We need to consider the effects of control-flow dependencies in policy update for advanced transactions. In face of a policy restriction update, aborting the one subtransaction is not enough. After the abort event on one subtransaction, proper actions may be needed on more subtransactions in the same advanced transaction. In this section, we propose algorithms for advanced transactions that not only update policies in real-time and also ensure satisfaction of all dependencies.

8.3.1 Policy Update Algorithm using Semantics of Policy Update

We now give an algorithm for policy change in advanced transactions. Similar to database systems, the operations specified on data objects are Read and Write. A policy object is associated with four operations: *Read, Deploy, WriteRelax, WriteRestrict.*

Definition 78

[Operations of Subtransaction] A *subtransaction* performs a logical unit of work, and it is a low-level coordinated component in an advanced transaction. A subtransaction T_{wi} is a partial order

with ordering relation $<_{wi}$ where

1. $T_{wi} \subseteq \{r_{wi}[x], w_{wi}[x] \mid x$ is a data object $\} \cup \{d_{wi}[x], r_{wi}[x], ws_{wi}[x], wx_{wi}[x] \mid x$ is a policy object $\} \cup \{a_{wi}, c_{wi}\}$;

2. $a_{wi} \in T_{wi}$ iff $c_{wi} \notin T_{wi}$;

3. if t is c_{wi} or a_{wi}, for any other operation $p \in T_{wi}$, $p_i <_{wi} t$; and

4. if $r_{wi}[x]$, $w_{wi}[x] \in T_{wi}$, then either $r_{wi}[x] <_{wi} w_{wi}[x]$ or $w_{wi}[x] <_{wi} r_{wi}[x]$.

5. if $d_{wi}[x]$, $ws_{wi}[x] \in T_{wi}$, then either $d_{wi}[x] <_{wi} ws_{wi}[x]$ or $ws_{wi}[x] <_{wi} d_{wi}[x]$.

6. if $d_{wi}[x]$, $wx_{wi}[x] \in T_{wi}$, then either $d_{wi}[x] <_{wi} wx_{wi}[x]$ or $wx_{wi}[x] <_{wi} d_{wi}[x]$.

The locking mechanism 17 is also very similar to the database systems.

	Wants			
Has	RL	WXL	WSL	DL
RL	Yes	No	No	Yes
WXL	No	No	No	No
WSL	No	No	No	No
DL	Yes	Yes	Signal to abort	Yes

Table 17: Locking Rules for Policy Objects

Definition 79

[Policy-Compliant Subtransaction] A subtransaction is *policy-compliant* if for every operation that subtransaction performs, there exists a policy that authorizes this subtransaction to perform the operation for the entire duration of the operation.

Definition 80

[Policy-Compliant Advanced Transaction] An advanced transaction is *policy-compliant* if all subtransactions belonging to this advanced transaction are policy-compliant.

The history of advanced transactions [32] extends the notion of history for database systems, in that the advanced transactions contain dependencies, who constrain the occurrence of primitive events in a history for advanced transactions.

Definition 81

[History] A *history* H of the concurrent execution of a set of advanced transactions $AT = \{AT_m, \ldots, AT_n\}$ is a partial order of all events associated with all the subtransactions in this set of advanced transactions AT. The events include all the primitive events (begin, commit, abort) and object events (read, write).

Fundamental to advanced transactions is the notion of control-flow dependencies. These dependencies capture the effects of subtransactions on other subtransactions through constraints on histories. In generalized advanced transactions, the occurrence of a primitive event (begin, commit, abort) in a history is constrained by the control-flow dependencies as follows:

(1) An event e_j may occur only after another event e_i in a history H: $e_j \in H \Rightarrow e_i \in H \wedge e_i \prec e_j$. Note that the begin, begin-on-commit, begin-on-abort, serial, termination, and commit dependencies belong to this type.

(2) An event e_i requires the occurrence of another event e_j in a history H: $e_i \in H \Rightarrow e_j \in H$. Note that the strong commit, abort, force-begin-on-begin(commit,abort,termination), exclusion, weak abort, and force-commit-on-abort dependencies belong to this type.

Next we define what we mean by a *policy-compliant* history.

Definition 82

[Policy-compliant History] A history is *policy-compliant* if all the subtransactions and hence all advanced transactions in the history are policy-compliant.

Theorem 28

Any history H generated by this policy update mechanism is policy-compliant.

Proof 28

The proof follows Theorem 23.

8.3.2 Ensure Satisfaction of Dependencies in Policy Updates

We now investigate the effects of policy update operations on advanced transactions and show how to ensure satisfaction of different dependencies. Only the policy restriction update will trigger an

abort event. In traditional database systems, the abort event of T_{wi} will not affect any other subtransactions, since the subtransactions in traditional database systems are standalone and have no relationship with each other. However, in the advanced transaction systems, this is different. With the abort event on T_{wi}, in order to ensure the satisfaction of all dependencies defined in this advanced transaction, more proper action may be needed on other subtransactions in AT_w as well during the policy updating operations. The abort operation of one subtransaction will necessitate certain proper operations (like force to begin, force to abort) to be performed on the other subtransactions in the same advanced transaction as well.

In this section, we inspect the impact of different control-flow dependencies on policy restriction updates in advanced transactions. Suppose there is an advanced transaction AT_w, and it contains two subtransactions T_{wi} and T_{wj} who are coordinated by a control-flow dependency $T_{wi} \rightarrow_d T_{wj}$. Suppose that subtransaction T_{wi} (or T_{wj}) is executing by virtue of certain policy P_{pt}. We assume that at this point, due to some security level changes, policy P_{pt} needs to be write restriction updated. Under this situation, there are three different ways how this policy updating operation could affect the advanced transaction AT_w:

1. T_{wi} is executing by virtue of policy P_{pt}, and P_{pt} is being write restriction updated;
2. T_{wj} is executing by virtue of policy P_{pt}, and P_{pt} is being write restriction updated;
3. Both T_{wi} and T_{wj} are executing by virtue of policy P_{pt}, and P_{pt} is being write restriction updated;

8.3.2.1 P_{pt} is deployed by T_{wi}, with a dependency $T_{wi} \rightarrow_d T_{wj}$

Suppose subtransaction T_{wi} is executing by virtue of policy $P_{pt} =< S_{pt}, O_{pt}, rights >$, and P_{pt} is write restriction updated to $P'_{pt} =< S_{pt}, O_{pt}, less - rights >$. Under this situation, we need to inspect whether some actions are needed for the other subtransaction T_{wj} in the same advanced transaction, in order to ensure the satisfaction of the dependency $T_{wi} \rightarrow_d T_{wj}$ specified among the two subtransactions.

If P_{pt} is write restriction updated, subtransaction T_{wi} must abort. For the fifteen control-flow dependencies defined, some dependencies will require certain actions to be performed on T_{wj} as

Dependency	is T_{wj} affected?	actions needed
$T_{wi} \longrightarrow_{sc} T_{wj}$	No	-
$T_{wi} \longrightarrow_{a} T_{wj}$	Yes	T_{wj} must abort with a_{wi}
$T_{wi} \longrightarrow_{wa} T_{wj}$	Yes	T_{wj} must abort if has not committed
$T_{wi} \longrightarrow_{ex} T_{wj}$	No	-
$T_{wi} \longrightarrow_{fca} T_{wj}$	Yes	T_{wj} must commit with a_{wi}
$T_{wi} \longrightarrow_{fbc} T_{wj}$	No	-
$T_{wi} \longrightarrow_{fbb} T_{wj}$	No	-
$T_{wi} \longrightarrow_{fba} T_{wj}$	Yes	T_{wj} must start with a_{wi}
$T_{wi} \longrightarrow_{fbt} T_{wj}$	Yes	T_{wj} must start with a_{wi}
$T_{wi} \longrightarrow_{c} T_{wj}$	No	-
$T_{wi} \longrightarrow_{t} T_{wj}$	Yes	T_{wj} is allowed to terminate
$T_{wi} \longrightarrow_{b} T_{wj}$	No	-
$T_{wi} \longrightarrow_{s} T_{wj}$	Yes	T_{wj} is allowed to start
$T_{wi} \longrightarrow_{bc} T_{wj}$	Yes	T_{wj} cannot start
$T_{wi} \longrightarrow_{ba} T_{wj}$	Yes	T_{wj} is allowed to start

Table 18: Effects of a_{wi} with Different Dependencies

well. To ensure the satisfaction of these dependencies, proper actions on T_{wj} must be analyzed and adopted. Among these fifteen dependencies, there are nine dependencies can affect the execution of the other subtransaction T_{wj} with abort event of T_{wi}. These dependencies are showing in table 18. And the detailed analysis is listed following.

[Abort dependency] ($T_{wi} \rightarrow_{a} T_{wj}$): if T_{wi} aborts, then T_{wj} must also abort due to the constraints of the abort dependency.

[Weak abort dependency] ($T_{wi} \rightarrow_{wa} T_{wj}$): If T_{wi} aborts and T_{wj} has not committed yet, then T_{wj} must also abort due to the constraints of the weak abort dependency. Actions: abort T_{wi}, abort T_{wj}. However, if T_{wj} has already committed then there is no action needed on T_{wj}.

[Force-commit-on-abort dependency] ($T_{wi} \rightarrow_{fca} T_{wj}$): if T_{wi} aborts, then T_{wj} must commits due to the constraints of the force commit-on-abort dependency. Note that, in our reliable scheduler [119], to ensure this force-commit-on-abort dependency in normal execution, T_{wi} cannot start execution before the commit event of T_{wj}. Therefore, this dependency will not affect T_{wj} in the proposed reliable scheduler.

[Force-begin-on-abort dependency] ($T_{wi} \rightarrow_{fba} T_{wj}$): if T_{wi} aborts, then T_{wj} must begin due to

Dependency	is T_{wi} affected?	actions needed
$T_{wi} \longrightarrow_{sc} T_{wj}$	Yes	T_{wi} must abort
$T_{wi} \longrightarrow_{t} T_{wj}$	Yes	T_{wi} must abort if has not committed yet

Table 19: Effects of a_{wj} with Different Dependencies

the constraints of the force begin-on-abort dependency. Actions: abort T_{wi}, begin T_{wj}.

[Force-begin-on-termination dependency] ($T_{wi} \rightarrow_{fbt} T_{wj}$): if T_{wi} terminates, then T_{wj} must begin due to the constraints of the force begin-on-termination dependency. Actions: abort T_{wi}, begin T_{wj}. Note this is a special case of the force-begin-on-abort dependency.

[Termination dependency] ($T_{wi} \rightarrow_{t} T_{wj}$) : T_{wj} cannot commit or aborts, if T_{wi} has not terminated (abort) yet. After the abort operation on T_{wi}, T_{wj} can terminate now. Actions: abort T_{wi}, T_{wj} can commit/abort now.

[Serial dependency] ($T_{wi} \rightarrow_{s} T_{wj}$): T_{wj} cannot start, if T_{wi} has not terminated (abort) yet. Due to the abort operation on T_{wi}, T_{wj} can start now. Actions: after T_{wi} abort, T_{wj} may start now.

[Begin-on-commit dependency] ($T_{wi} \rightarrow_{bc} T_{wj}$): without the commit event of T_{wi}, T_{wj} cannot begin execution. Since T_{wi} is abort, T_{wj} cannot start. Actions: abort T_{wi}, T_{wj} cannot begin execution.

[Begin-on-abort dependency] ($T_{wi} \rightarrow_{ba} T_{wj}$: T_{wj} cannot start, if T_{wi} has not abort yet. Due to the abort operation on T_{wi}, T_{wj} can start now. Actions: after T_{wi} abort, T_{wj} may start now.

8.3.2.2 P_{pt} is deployed by T_{wj}, with a dependency $T_{wi} \rightarrow_{d} T_{wj}$

Suppose there is a policy $P_{pt} =< S_t, O_t, rights >$, and P_{pt} is restriction updated to $P_{pt} =< S_t, O_t, less-rights >$, under this situation, we need to check whether some actions are needed on subtransaction T_{wi} to ensure the satisfaction of the dependencies specified between these two subtransactions. When P_{pt} is write restriction updated, T_{wj} must abort, two control-flow dependencies can affect the execution of T_{wi}, as showing in table 19.

[Strong commit dependency] ($T_{wi} \rightarrow_{sc} T_{wj}$): if T_{wj} aborts, T_{wi} must abort. An implicit dependency $T_{wj} \rightarrow_{a} T_{wi}$ is implied with the strong commit dependency $T_{wi} \rightarrow_{sc} T_{wj}$. T_{wi} cannot

Dependency	effects of a_{wi} and a_{wj} on AT_w
$T_{wi} \longrightarrow_{fca} T_{wj}$	if both T_{wj}, T_{wi} abort, dependency is violated
$T_{wi} \longrightarrow_{fba} T_{wj}$	T_{wj} is required to start by *fba* dependency, but it cannot start without the adequate privilege by P_{pt}. This is a conflict. We need to rollback the whole advanced transaction.
$T_{wi} \longrightarrow_{fbt} T_{wj}$	same effects as the *fba* dependency

Table 20: Effects of a_{wi} and a_{wj} on the Advanced Transaction

commit after the abort event on T_{wj}. Actions: abort T_{wj}, and abort T_{wi}.

[Termination dependency] ($T_{wi} \rightarrow_t T_{wj}$) : this dependency requires that T_{wj} cannot terminate (commit or abort) until T_{wi} has already terminated. To ensure the satisfaction of this termination dependency, we need to abort T_{wi} first then abort T_{wj} in this write restriction update. Actions: abort T_{wi} if it has not committed, then abort T_{wj}. If T_{wi} has already committed or aborted, no actions needed.

8.3.2.3 P_{pt} is deployed by both T_{wi} and T_{wj}, with a dependency $T_{wi} \rightarrow_d T_{wj}$

Suppose P_{pt} is deployed by both subtransactions T_{wi} and T_{wj}. When P_{pt} is restriction updated, both T_{wi} and T_{wj} are required to abort. Under this situation, we need to check whether some actions are needed to ensure the satisfaction of dependencies. If P_{pt} is restriction updated, three dependencies can affect the execution of the advanced transaction AT_w, as showing in table 20.

[Force-commit-on-abort dependency] ($T_{wi} \rightarrow_{fca} T_{wj}$): the force-commit-on-abort dependency requires that if T_{wi} aborts, then T_{wj} must commit. If both T_{wi} and T_{wj} abort, this dependency is violated. However, in the reliable scheduler proposed in paper [119], the situation of both T_{wi} and T_{wj} are deploying P_{pt} will not appear. In our reliable scheduler, to ensure the force-commit-on-abort dependency, we require that T_{wi} cannot start execution without the commit event of T_{wj}. In other words, if T_{wj} has not committed yet, T_{wi} cannot start in the reliable scheduler. Therefore, when P_{pt} is updated, T_{wj} must abort immediately, and at this moment T_{wi} has not started yet. Thus this dependency will not affect the execution of advanced transaction in case of policy updates in our reliable scheduler.

[Force-begin-on-abort dependency] $T_{wi} \rightarrow_{fba} T_{wj}$: the force-begin-on-abort dependency requires that if T_{wi} abort then T_{wj} must begin. There are two situations here, (I) at the moment of the update operation on P_{pt}, if T_{wj} has already started, this dependency is preserved; (II) at the moment of the update operation on P_{pt}, if T_{wj} has not started yet, we have a problem on the execution of this advanced transaction – T_{wj} cannot begin due to the restriction update on P_{pt}; however, the force-begin-on-abort dependency requires that T_{wj} must begin. The subtransaction T_{wj} is required to both 'begin' and 'not begin', this is a contradiction. Actions need: undo and rollback the whole advanced transaction.

[Force-begin-on-termination dependency] ($T_{wi} \rightarrow_{fbt} T_{wj}$: note this is just a special case of the force-begin-on-abort dependency. Actions need: undo and rollback the whole advanced transaction.

The actions needed are to undo and rollback the whole advanced transaction. The undo recovery mechanism and the correctness proof are discussed in some earlier paper [92, 119].

8.3.3 Policy Updating Algorithm for Advanced Transactions

Algorithm 23

Update policy in advanced transaction

Input: (i) Policy $P_i =< S_i, O_i, R_i >$ that is to be updated, (ii) R_i' – the new rights of P_i, (iii) The advanced transaction specification $AT_w =< S, D, C >$

Output: (i) Updated policy P_i', (ii) A consistent state with all dependency constraints are preserved

Procedure UpdatePolicy(P_i, R_i', AT_w)

begin

if $lub\{(R_i),(R_i')\} = (R_i')$ /* policy relaxation */

begin

acquire $WXL(P_i)$

$P_i' =< S_i, O_i, R_i' >$ /* modify P_i */

release $WXL(P_i)$

end

else /* policy restriction */

begin

acquire $WSL(P_i)$

for every dependency $T_{wi} \rightarrow_d T_{wj}$ in $AT_w =< S, D, C >$

begin

if both T_{wi} and T_{wj} are executing by virtue of P_i

begin

if $d = fbt \mid fba$ and T_{wj} has not started yet

undo the whole advanced transaction

else

signal to abort T_{wi} and T_{wj}

the scheduler updates the execution states for T_{wi} and T_{wj}

end

else if T_{wi} is executing by virtue of P_i

begin

signal to abort T_{wi}

the scheduler updates the execution states for T_{wi}

if $d = a$

notify the scheduler to abort T_{wj}

if $d = wa$ and T_{wj} has not committed yet

notify the scheduler to abort T_{wj}

if $d = fba \mid fbt$

notify the scheduler to start T_{wj}

if $d = t$

notify the scheduler T_{wj} can terminate

if $d = ba \mid s$

notify the scheduler T_{wj} can start

if $d = bc$

notify the scheduler that T_{wj} cannot start

the scheduler updates the execution state for T_{wj}

and it is responsible for enforcing all dependencies with T_{wj}

end

else if T_{wj} is executing by virtue of P_i

begin

signal to abort T_{wj}

the scheduler updates the execution states for T_{wj}

if $d = sc$

notify the scheduler to abort T_{wi}

if $d = t$ and T_{wi} has not terminated yet

notify the scheduler to abort T_{wi}

the scheduler updates the execution state for T_{wi}

and it is responsible for enforcing all dependencies with T_{wi}

end

end for

$P_i' =< S_i, O_i, R_i' >$ /* modify P_i */

release $WSL(P_i)$

end

end

Theorem 29

The proposed algorithm ensures satisfaction of all dependencies of the advanced transaction during policy update.

Proof 29

Recall that the control-flow dependencies could be classified into two subclasses – the event enforcement type and event ordering type. The event enforcement type of dependencies require that, if one event (like c_{wi}, a_{wi}) happens then another event (like c_{wj}, b_{wj}) must also be performed. The event ordering type of dependencies impose sequential orders on two events (like c_{wi} before c_{wj}). The $sc, a, ex, fca, fbb, fba, fbc, fbt$, and wa dependencies are event enforcement type of dependencies.

The c, b, t, ba, bc, s dependencies are event ordering type of dependencies.

For every dependency $T_{wi} \rightarrow_d T_{wj}$, we inspect the effects of an abort event a_{wi} (or a_{wj}) with subtransaction T_{wi} and T_{wj} respectively, and check whether the algorithm preserves the dependency. (I) We first inspect the effects of abort event a_{wi}, the other subtransaction T_{wj} may or may not be affected. Among the event enforcement type of dependencies, the a, wa, fca, fba, fbt dependencies require proper actions to be taken on the other subtransaction T_{wj}. The actions are listed in Table 18. Among the event ordering type of dependencies, the t, s, ba, bc dependencies could affect the begin event of subtransaction T_{wj} (like T_{wj} is allowed to start after abort of T_{wi} or T_{wj} cannot start with abort of T_{wi}. The actions need to be taken are listed in Table 18. (II) We next inspect the effects of the abort event a_{wj} on the other subtransaction T_{wi}. The sc, t dependencies are the only two dependencies that require actions to be taken on T_{wi} in face of the abort events on T_{wj}. (III) We also inspect the effects of the aborts events a_{wi} and a_{wj} on the advanced transaction AT_w. The *fca* dependency requires that if T_{wi} aborts then T_{wj} must commit. The updated policy requires that T_{wj} must abort. There is a conflict between the dependency constraints and the access control policy constraints. In this case, we need to undo the whole advanced transaction. The *fba*, *fbt* dependencies require that T_{wj} must start, but the updated policy requires that T_{wj} cannot start. This also results in a conflict situation. We need to undo the whole advanced transaction in these cases. The proposed algorithm can handle this correctly.

In additional to the subtransactions (like T_{wj}) which are direct neighbor to T_{wi} in the dependency graph, other subtransactions that are non-neighbor to T_{wi} can also be affected due to the transitive nature of dependencies. The abort event on subtransaction a_{wi} can affect many more subtransactions transitively within the same advanced transaction. The reliable scheduler for advanced transactions proposed in paper [119, 121] can handle this situation correctly. In face of the abort event of one subtransaction, the scheduler sets the state of this subtransaction to 'abort' and it can enforce all the dependencies associated with this subtransaction correctly. The algorithm proposed can ensure satisfaction of all dependencies correctly.

CHAPTER IX

CONCLUSIONS

In this dissertation, I proposed a reliable and survivable transaction processing model for generalized advanced transactions. This work provides useful tools and strategies for the advanced transaction developers and users. Unlike the traditional database systems, there is no existing guideline on how to correctly specify an advanced transaction. For the traditional database transactions, the well-known ACID properties are the criteria – the atomicity and durability properties define the notion of reliability in the context of transaction; the consistency and isolation properties define the notion of correctness in execution of transactions.

Prior to this research, there is not much work on the correct specification of advanced transaction. Dependency is the most important feature of advanced transaction models. Different types of control-flow dependencies coordinate the execution of subtransactions. Nevertheless, the issues like conflicting dependencies, and the impacts of dependencies on execution of advanced transactions are not addressed in literature. There is no existing work on how to correctly enforce dependencies and reliably execute advanced transactions.

I proposed mechanisms that will help both the application designer and normal users of advanced transactions. For the normal users, this research provides the tool for detecting and removing different kinds of dependency conflicts and dependency redundancies. The tool will help them to specify well-structured transactions, which can prevent deadlock or contradicting situations due to improper dependency structures. For the application developers, this work will help them to construct reliable execution models for advanced applications. With the proposed reliable scheduling algorithm and failure recovery algorithm, the application developers can build practical scheduler to correctly execute advanced transactions and to enforce all dependencies correctly. The strategies in damage assessment and repair will also enables them to design a survivable system, which can restore to a consistent state after the occurrence of malicious attacks.

In this research, I proposed the notion of generalized advanced transactions. The generalized

advanced transaction is very expressive. It can be used to represent most existing advanced transaction models, like the SAGA, nested transactions, split-transactions, workflow systems, etc. The limitation with most existing extended transaction models are that their applications are very limited, because they are normally designed for a specific environment. Hence, this generalized advanced transaction provides a huge advantage. It can support a variety of existing and yet-to-develop models by limiting the kind of dependencies to be used. Having this unified model as infrastructure, I analyze the dependency properties and build the execution model that can be used in different environments.

In this dissertation, I emphasized on analyzing the properties of control-flow dependencies and discussed the well-structured advanced transactions. I proposed the notion of generalized advanced transactions that can be used to build different advanced transactions. I have shown how dependencies present in advanced transaction models can be analyzed to ensure correct execution. In this dissertation, I focused on two important properties of dependencies: conflict-free and minimality. Conflict-free property ensures that the dependencies are physically realizable. It also ensures that the dependencies are free from deadlocks. Minimality ensures that no redundant dependencies are specified. This helps to eliminate the processing of unnecessary dependencies. Once the interactions of dependencies are well-understood, new kinds of transaction processing models can be synthesized.

I also proposed a reliable transaction execution model. The execution model is able to schedule the well-structured advanced transaction correctly. I analyze the states of subtransactions in execution. I then build scheduling action tables for all the primitive dependencies and the composite dependencies. The scheduling action tables show the correct action to be taken under the current states. I also proposed algorithms to handle the failure recovery problems with advanced transactions. The proposed transaction processing model is reliable in scheduling and failure recovery, focusing on ensuring the satisfaction of all control-flow dependencies at run time. In this dissertation I have shown how to repair attacks caused by one or more malicious subtransactions in an advanced transaction. In addition to the read-write dependencies that are present in the traditional database transactions, advanced transactions have control-flow dependencies as well. These dependencies help spread the damage caused by malicious subtransactions and complicates the repairing

process. Our algorithm removes the effects of malicious advanced transactions which contain malicious subtransactions and tries to minimize undoing the unaffected benign advanced transactions. Only subtransactions that were affected are undone and re-executed.

Dynamic environments pose new challenge to access control. In such situations, the entities change, the configurations change and the operational modes change – all of these require real-time update of access control policies. In this dissertation, I address the problem of how policies can be updated in real-time and how the modified policies can be immediately enforced. We discuss our work in the context of a database system and then extend it into advanced transaction systems. A database consists of data objects that are accessed and modified through transactions. Transactions perform operations by virtue of the privileges given by the access control policies. Access control policies are specified in the form of policy objects that can be read, written, and deployed. We consider an environment in which transactions execute concurrently some of which are policy update transactions.

We start with the simple case where a single policy is defined over a subject and an object and show how updates to such policy can be performed in real-time. In real-world, multiple policies may be specified over a given subject and an object. Priorities may be specified on these policies. We propose a mechanism that addresses the update of such policies where the priorities are also subject to change. Our mechanisms generate serializable and policy-compliant histories. I then consider the advanced transaction system, and discussed how to ensure satisfaction of dependencies when updating a policies. The algorithm proposed preserves the dependency constraints. The mechanisms for real-time update of access control policies could be used by any security critical applications, for instance, financial sectors, business category or military applications.

Some limitations of this work are that it does not provide execution and recovery mechanisms for decentralized advanced transactions. This work supports the applications in distributed systems, but it requires that these applications must have a centralized control (or say a coordinating center site). In applications with centralized control, even if different subtransactions are executing on different sites, the center site will maintain the global control-flow dependency graph and it can coordinate the executions and ensure satisfaction of dependencies. Nevertheless, for certain applications, such a centralized control may not be available or may not be efficient. Under this situation, we need

decentralized control over different sites to interact together and ensure reliable execution. This maintains a task to be solved in future. Another limitation is that we do not consider optimization issues for the scheduling algorithm. How to schedule a sequence of subtransactions and minimize the execution time could be addressed in the future. In the dissertation, we focus on how to ensure satisfaction of dependencies in execution.

A number of work remained to be continued in the future. First, we will extend this work to a distributed environment where there are decentralized control over participating sites. In these applications, the issues like coordination among different sites, message transmissions, and synchronization problems must be solved. Further, the damage assessment and repair for decentralized system is far more complicated. We need to build the global read-write dependency graph as well as global control-flow dependency graph. How to identify the local affected subtransactions and global affected subtransactions is a problem to be considered.

The completion sets specified by a user in a generalized advanced transaction must conform to the given dependencies. If one completion set violates the dependency constraints, it is not feasible – since it will hold resources infinitely and lead to deadlock and unavailability issues in execution. Computing all the correct completion sets manually can be a time-consuming and error-prone work. Generating all possible completion sets automatically is greatly needed by the application developers. In the future, we will propose algorithms to calculate all possible completion sets that satisfy the dependency constraints.

As regarding to the policy update work, in future we plan to extend our approach to handle more complex kinds of authorization policies, such as the Role-Based Access Control (RBAC) models, support for negative authorization policies, and incorporating conditions in authorization policies. Specifically, we plan to investigate how policies specified in the PONDER specification language [34] can be updated. In RBAC model, the role assignment, constraints over roles, and separation of duties complicate the policy restriction operations and policy relaxation operation. We will study these issues next.

APPENDIX A

SCHEDULING ACTION TABLES

The dependency scheduling tables specify the necessary actions that must be taken by the scheduler to ensure the satisfaction of all the dependencies. In this appendix, we list the primitive dependency scheduling tables for all the control-flow dependencies we studied.

1. $[T_{ij} \rightarrow_{sc} T_{ik}]$, the scheduler needs to ensure the commitment of both subtransactions. The scheduling action table TB_{sc} is shown in Table 21.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	–	–	reject T_{ij}
ex_{ij}	–	–	–	–	abort T_{ij}
pr_{ij}	delay T_{ij}	delay T_{ij}	delay T_{ij}	–	abort T_{ij}
cm_{ij}	/	/	/	final	/
ab_{ij}	–	–	–	final	final

Table 21: The Scheduling Action Table for *strong commit* Dependency

2. $[T_{ij} \rightarrow_{c} T_{ik}]$, the scheduler needs to ensure the order of two subtransactions' commitment. The scheduling action table TB_c is shown in Table 22.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	delay T_{ik}	/	–
ex_{ij}	–	–	delay T_{ik}	/	–
pr_{ij}	–	–	delay T_{ik}	/	–
cm_{ij}	–	–	–	final	final
ab_{ij}	–	–	–	final	final

Table 22: The Scheduling Action Table for *commit* Dependency

3. $[T_{ij} \rightarrow_{a} T_{ik}]$, the scheduler should ensure that the abort events of both subtransactions all occur if subtransaction T_{ij} has to abort. The scheduling action table TB_a is shown in Table 23.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	delay T_{ik}	/	–
ex_{ij}	–	–	delay T_{ik}	/	–
pr_{ij}	–	–	delay T_{ik}	/	–
cm_{ij}	–	–	–	final	final
ab_{ij}	–	abort T_{ik}	abort T_{ik}	/	final

Table 23: The Scheduling Action Table for *abort* Dependency

4. $[T_{ij} \rightarrow_t T_{ik}]$,the scheduler should control the execution order of the two subtransactions. The scheduling action table TB_t is shown in Table 24.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	delay T_{ik}	/	/	/	/
ex_{ij}	delay T_{ik}	/	/	/	/
pr_{ij}	delay T_{ik}	/	/	/	/
cm_{ij}	–	–	–	final	final
ab_{ij}	–	–	–	final	final

Table 24: The Scheduling Action Table for *termination* Dependency

5. $[T_{ij} \rightarrow_{ex} T_{ik}]$, the scheduler should ensure that subtransaction T_{ik} aborts if subtransaction T_{ij} commits. The scheduling action table TB_{ex} is shown in Table 25.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	delay T_{ik}	/	–
ex_{ij}	–	–	delay T_{ik}	/	–
pr_{ij}	–	–	delay T_{ik}	/	–
cm_{ij}	reject T_{ik}	abort T_{ik}	abort T_{ik}	/	final
ab_{ij}	–	–	–	final	final

Table 25: The Scheduling Action Table for *exclusion* Dependency

6. $[T_{ij} \rightarrow_{fca} T_{ik}]$, the scheduler needs to enforce the execution order of the two subtransactions to ensure this dependency, because the scheduler cannot control a subtransaction's commitment or abort. The scheduling action table TB_{fca} is shown in Table 26.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	delay T_{ij}	delay T_{ij}	delay T_{ij}	–	reject T_{ij}
ex_{ij}	/	/	/	–	/
pr_{ij}	/	/	/	–	/
cm_{ij}	/	/	/	final	/
ab_{ij}	/	/	/	final	/

Table 26: The Scheduling Action Table for *force-commit-on-abort* Dependency

7. $[T_{ij} \rightarrow_b T_{ik}]$, the scheduler should ensure the order of the two subtransactions' begin primitives. The scheduling action table TB_b is shown in Table 27.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	delay T_{ik}	/	/	/	/
ex_{ij}	–	–	–	–	–
pr_{ij}	–	–	–	–	–
cm_{ij}	–	–	–	final	final
ab_{ij}	–	–	–	final	final

Table 27: The Scheduling Action Table for *begin* Dependency

8. $[T_{ij} \rightarrow_s T_{ik}]$, the scheduler should ensure the execution order of the two subtransactions. The scheduling action table TB_s is shown in Table 28.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	delay T_{ik}	/	/	/	/
ex_{ij}	delay T_{ik}	/	/	/	/
pr_{ij}	delay T_{ik}	/	/	/	/
cm_{ij}	–	–	–	final	final
ab_{ij}	–	–	–	final	final

Table 28: The Scheduling Action Table for *serial* Dependency

9. $[T_{ij} \rightarrow_{bc} T_{ik}]$, the scheduler needs to ensure that the beginning of subtransaction T_{ik} must be after the commitment of subtransaction T_{ij}. The scheduling action table TB_{bc} is shown in Table 29.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	delay T_{ik}	/	/	/	/
ex_{ij}	delay T_{ik}	/	/	/	/
pr_{ij}	delay T_{ik}	/	/	/	/
cm_{ij}	–	–	–	final	final
ab_{ij}	reject T_{ik}	/	/	/	/

Table 29: The Scheduling Action Table for *begin-on-commit* Dependency

10. $[T_{ij} \rightarrow_{ba} T_{ik}]$, the scheduler needs to ensure that the beginning of subtransaction T_{ik} must be after the abort of subtransaction T_{ij}. The scheduling action table TB_{ba} is shown in Table 30.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	delay T_{ik}	/	/	/	/
ex_{ij}	delay T_{ik}	/	/	/	/
pr_{ij}	delay T_{ik}	/	/	/	/
cm_{ij}	reject T_{ik}	/	/	/	/
ab_{ij}	–	–	–	final	final

Table 30: The Scheduling Action Table for *begin-on-abort* Dependency

11. $[T_{ij} \rightarrow_{fbb} T_{ik}]$, the scheduler must ensure that subtransaction T_{ik} begins execution when subtransaction T_{ij} begins. The scheduling action table TB_{fbb} is shown in Table 31.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	–	–	–
ex_{ij}	exec T_{ik}	–	–	–	–
pr_{ij}	/	–	–	–	–
cm_{ij}	/	–	–	final	final
ab_{ij}	/	–	–	final	final

Table 31: The Scheduling Action Table for *force-begin-on-begin* Dependency

12. $[T_{ij} \rightarrow_{fbt} T_{ik}]$, the scheduler should ensure that subtransaction T_{ik} begins execution when subtransaction T_{ij} is in its final state. The scheduling action table TB_{fbt} is shown in Table 32.

13. $[T_{ij} \rightarrow_{fbc} T_{ik}]$, the scheduler must ensure that subtransaction T_{ik} begins execution, when subtransaction T_{ij} commits. The scheduling action table TB_{fbc} is shown in Table 33.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	–	–	–
ex_{ij}	–	–	–	–	–
pr_{ij}	–	–	–	–	–
cm_{ij}	exec T_{ik}	–	–	final	final
ab_{ij}	exec T_{ik}	–	–	final	final

Table 32: The Scheduling Action Table for *force-begin-on-termination* Dependency

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	–	–	–
ex_{ij}	–	–	–	–	–
pr_{ij}	–	–	–	–	–
cm_{ij}	exec T_{ik}	–	–	final	final
ab_{ij}	–	–	–	final	final

Table 33: The Scheduling Action Table for *force-begin-on-commit* Dependency

14. $[T_{ij} \rightarrow_{fba} T_{ik}]$, the scheduler must ensure that subtransaction T_{ik} begins execution, when T_{ij} aborts. The scheduling action table TB_{fba} is shown in Table 34.

action	in_{ik}	ex_{ik}	pr_{ik}	cm_{ik}	ab_{ik}
in_{ij}	–	–	–	–	–
ex_{ij}	–	–	–	–	–
pr_{ij}	–	–	–	–	–
cm_{ij}	–	–	–	final	final
ab_{ij}	exec T_{ik}	–	–	final	final

Table 34: The Scheduling Action Table for *force-begin-on-abort* Dependency

REFERENCES

[1] N. R. Adam, V. Atluri, and W-K. Huang. Modeling and Analysis of Workflows Using Petri Nets. *Journal of Intelligent Information Systems, Special Issue on Workflow and Process Management*, 10(2):131–158, March 1998.

[2] G. Alonso, D. Agrawal, A. Abbadi, M. Kamath, R. G., and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 574–581, February 1996.

[3] P. Ammann, S. Jajodia, and P. Liu. A fault tolerance approach to survivability. In *The Symposium on Protecting NATO Information Systems in the 21st Century*, Washington, DC, October 1999.

[4] P. Ammann, S. Jajodia, and P. Liu. Recovery from Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14:1167–1185, 2002.

[5] P. Ammann, S. Jajodia, C. McCollum, and B. Blaustein. Surviving Information Warfare Attacks on Databases. In *Proceedings of 1997 IEEE Symposium on Security and Privacy*, May 1997.

[6] P. Ammann, S. Jajodia, and I. Ray. Applying Formal Methods to Semantic-Based Decomposition of Transactions. *ACM Transactions on Database Systems*, 22(2):215–254, June 1997.

[7] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proceeding of the Eighteenth International Conference on Very Large Data Bases*, August 1992.

[8] V. Atluri and W-K. Huang. An Authorization Model for Workflows. In *Proceedings of the Fifth European Symposium on Research in Computer Security*, pages 44–64, September 1996.

[9] V. Atluri and W-K. Huang. Enforcing Mandatory and Discretionary Security in Workflow Management Systems. *Journal of Computer Security*, 5(4):303–340, 1997.

[10] V. Atluri, W-K. Huang, and E. Bertino. An Execution Model for Multilevel Secure Workflows. In *Proceedings of the Eleventh IFIP WG11.3 Working Conference on Database Security*, pages 151–165, August 1997.

[11] V. Atluri, W-K. Huang, and E. Bertino. An Execution Model for Multilevel Secure Workflows. In *Proceedings of the Eleventh IFIP WG11.3 International Conference on Database Security XI*, pages 151–165, July 1998.

[12] V. Atluri, W-K. Huang, and E. Bertino. A Semantic-Based Execution Model for Multilevel Secure Workflows. *Journal of Computer Security*, 8(1), August 2000.

[13] P. C. Attie, M. P. Singh, A. P. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 134–145, Dublin, Ireland, August 1993. Morgan Kaufmann.

[14] B.R. Badrinath and K. Ramamritham. Semantics-based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[15] R. S. Barga and C. Pu. A Practical and Modular Implementation of Extended Transaction Models. In *Proceedings of 21th International Conference on Very Large Data Bases*, pages 206–217, September 1995.

[16] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[17] S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artificial Intelligence*, pages 143–148, Las Vegas, NV, 2000.

[18] C. Beeri, P. A. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transaction Systems. *Journal of the ACM*, 36(2):230–269, 1989.

[19] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

[20] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[21] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

[22] E. Bertino, G. Chiola, and L. V. Mancini. Deadlock Detection in the Face of Transaction and Data Dependencies. In *Proceedings of the Nineteenth International Conference on Application and Theory of Petri Nets*, pages 266–285, Lisbon, Portugal, June 1998.

[23] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Transaction on Information Systems Security*, 2(1):65–104, 1999.

[24] A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1994.

[25] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.

[26] Q. Chen and U. Dayal. Failure Handling for Transaction Hierarchies. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 245–254, April 1997.

[27] E. C. Cheng. OMM: An Organization Modeling and Management System. In *Proceedings of Towards Adaptive Workflow Systems Workshop co-located with CSCW 98*, Seattle, WA, November 1998.

[28] P. Chrysanthis. ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis, September 1991.

[29] P. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 194–203, May 1990.

[30] P. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 103–111, September 1991.

[31] P. Chrysanthis and K. Ramamritham. ACTA: The SAGA Continues. In *Database Transaction Models for Advanced Applications, A. K. Elmagarmid Ed., Morgan Kaufmann Publishers*, 1992.

[32] P. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19:450–491, September 1994.

[33] C. J. Crawley and O. A. Bukhres. Failure Handling in CORBAflow: A CORBA-Based Transactional Workflow Architecture. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, volume 6, pages 77–86, Melbourne, Australia, April 1997.

[34] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proceedings of the Policy Workshop*, Bristol, U.K., January 2001.

[35] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, and M. Sloman. Tools for Domain-based Policy Management of Distributed Systems. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, Florence, Italy, April 2002.

[36] N. C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, London, U.K., 2002.

[37] U. Dayal, M. Hsu, and R.Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceeding of the Seventeenth International Conference on Very Large Data Bases*, September 1991.

[38] J. Eder and W. Liebhart. Workflow Recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems*, pages 124–134, Brussels, Belgium, June 1996.

[39] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, Fourth Edition*. Addison-Wesley, Reading, MA, 2003.

[40] J. Thomas Haigh et al. Assured Service Concepts and Models: Security in Distributed Systems. Technical Report RL-TR-92-9, Rome Laboratory, Air Force Material Command, Rome, NY, January 1992.

[41] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[42] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1987.

[43] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2002.

[44] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[45] M. M. Gore and R. K. Ghosh. Recovery in Distributed Extended Long-lived Transaction Models. In *Proceedings of Sixth International Conference on Database Systems for Advanced Applications*, pages 313–320, April 1999.

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

[47] N. Griffeth and H. Velthuijsen. Reasoning About Goals to Resolve Conflicts. In *Proceedings of the International Conference on Intelligent Cooperative Information Systems*, pages 197–204, Los Alamitos, California, 1993.

[48] R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998.

[49] G. T. Heineman. A Transaction Manager Component Supporting Extended Transaction Models. PhD Thesis, Columbia University, 1996.

[50] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991.

[51] M. Hitchens and V. Varadarajan. Tower: A Language for Role-Based Access Control. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.

[52] D. Hollingsworth. Workflow Reference Model. Technical report, Workflow Management Coalition, Brussels, Belgium, 1994.

[53] D. Hollingsworth. The Workflow Reference Model. Workflow Management Coalition, Document Number TC00-1003, January 1995.

[54] I. Houston, M. C. Little, I. Robinson, S. K. Shrivastava, and S. M. Wheater. The CORBA Activity Service Framework for Supporting Extended Transactions. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pages 197–215, November 2001.

[55] W-K. Huang and V. Atluri. SecureFlow: A Secure Web-Enabled Workflow Management System. In *Proceedings of ACM Workshop on Role-Based Access Control 1999*, pages 83–94, 1999.

[56] P. Hung and K. Karlapalem. A Secure Workflow Model. In *Australasian Information Security Workshop*, Adelaide, Australia, 2003.

[57] S. Jajodia and V. Atluri. Alternative Correctness Criteria for Concurrent Execution of Transactions in Multilevel Secure Databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 216–224, Oakland, CA, 1992.

[58] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[59] M. Kamath and K. Ramamritham. Correctness Issues in Workflow Management. *Journal of the Distributed System Engineering*, 3:213–221, 1996.

[60] M. Kamath and K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. In *Proceeding of the Fourteenth International Conference on Data Engineering*, February 1998.

[61] P. Kammer, G. Bolcer, R. Taylor, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work*, 9(3-4):269–292, 2000.

[62] B. Kiepuszewski, R. Muhlberger, and M. Orlowska. Flowback: Providing Backward Recovery for Workflow Systems. In *Proceeding of the ACM SIGMOD International Conference on Management of Data*, pages 555–557, June 1998.

[63] H. F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 95–106, August 1990.

[64] H. F. Korth and G. Speegle. Formal Aspects of Concurrency Control in Long-ouration Transaction Systems Using the NT/PV Model. *ACM Transactions on Database Systems*, 19(3):492–535, September 1994.

[65] C. Lala and B. Panda. On Achieving Fast Damage Appraisal in Case of Cyber Attacks. In *Proceedings of the IEEE Workshop on Information Assurance*, West Point, NY, 2000.

[66] J. L. Lin and M. H. Dunham. A Survey of Distributed Database Checkpointing. *Distributed and Parallel Databases*, 5:289–319, 1997.

[67] M. C. Little. Transactions and Web Services. *Communications of the ACM*, 46(10):49–54, 2003.

[68] C. Liu, M.Orlowska, X. Lin, and X. Zhou. Confirmation: A Solution to Non-compensatability Problem in Workflow Systems. In *Proceedings of the Fifteenth International Conference on Data Engineering*, Sydney, Australia, March 1999.

[69] P. Liu, P. Ammann, and S. Jajodia. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[70] P. Liu and X. Hao. Efficient Damage Assessment and Repair in Resilient Distributed Database Systems. In *Proceedings of the Fifteenth IFIP WG11.3 Working Conference on Data and Application Security*, pages 75–89, Niagara, Ontario, Canada, July 2001.

[71] E. Lupu and M. Sloman. Conflict Analysis for Management Policies. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*, pages 430–443, San Diego, California, May 1997. Chapman & Hall.

[72] Nancy A. Lynch. Multilevel Atomicity—A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[73] L. V. Mancini, I. Ray, S. Jajodia, and E. Bertino. Flexible Transaction Dependencies in Database Systems. *Distributed and Parallel Databases*, 8:399–446, 2000.

[74] D. Manolescu and R. Johnson. Dynamic Object Model and Adaptive Workflow. In *Proceedings of Metadata and Active Object-Model Pattern Mining Workshop*, Denver, CO, November 1999.

[75] C. P. Martin and K. Ramamritham. Delegation: Efficiently Rewriting History. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 266–275, April 1997.

[76] M. G. Mathews. Supporting Dynamic Change in B2B Coordination. Technical report, The MITRE Corporation, June 2001.

[77] N. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building Reconfiguration Primitives into the Law of a System. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 89–97, Annapolis, MD, May 1996.

[78] J. E. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. PhD Thesis 260, MIT, Cambridge, MA, April 1981.

[79] R. Muller and E. Rahm. Rule-Based Dynamic Modification of Workflows in a Medical Domain. In A.P. Buchmann, editor, *Proceedings of BTW*, pages 429–448, Freiburg in Breisgau, Germany, March 1999. Springer.

[80] OMG. Additional Structuring Mechanisms for the OTS Specification. OMG, Document ORBOS, 2000-04-02, September 2000.

[81] R. Ortalo. A Flexible Method for Information Systems Security Policy Specification. In *Proceedings of the Fifth European Symposium on Research in Computer Security*, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.

[82] B. Panda and J. Giordano. Reconstructing the Database after Electronic Attacks. In *Proceedings of the Twelfth IFIP WG11.3 International Working Conference on Database Security*, Chalkidiki, Greece, July 1998.

[83] B. Panda and K. A. Haque. Extended Data Dependency Approach: A Robust Way of Rebuilding Database. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, Madrid, Spain, March 2002.

[84] M. Prochazka. Advanced Transactions in Enterprise JavaBeans. In *Proceedings of the Second International Workshop on Engineering Distributed Objects*, Davis, CA, USA, November 2000.

[85] M. Prochazka. Extending Transactions in Enterprise JavaBeans. Tech. Report No. 2000/3, Department of Software Engineering, Charles University, Prague, January 2000.

[86] M. Prochazka. Advanced Transactions in Component-Based Software Architectures. Ph.D. Thesis, Charles University, University of Evry, Feb 2002, 2002.

[87] C. Pu, G. E. Kaiser, and N. C. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 26–37, August 1988.

[88] I. Ray. Real Time Updation of Security Policies. *Data and Knowledge Engineering*, 49(3):287–309, June 2004.

[89] I. Ray. Applying Semantic Knowledge to Real-Time Update of Access Control Policies. *IEEE Transactions on Knowledge and Data Engineering*, 2005. To appear.

[90] I. Ray and T. Xin. Concurrent and Real-Time Update of Access Control Policies. In *Proceedings of the Fourteenth International Conference on Database and Expert Systems*, Prague, Czech Republic, September 2003.

[91] I. Ray and T. Xin. Implementing Real-Time Update of Access Control Policies. In *Proceedings of the Eighteenth IFIP WG11.3 Working Conference on Data and Applications Security*, July 2004.

[92] I. Ray, T. Xin, and Y. Zhu. Ensuring Task Dependencies During Workflow Recovery. In *Proceedings of the Fifteenth International Conference on Database and Expert Systems*, Zaragoza, Spain, August 2004.

[93] A. Reuter. Contracts: A Means For Extending Control Beyond Transaction Boundaries. In *Proceedings of the Third International Workshop on High Performance Transaction Systems*, September 1989.

[94] C. Ribeiro, A. Zuquete, and P. Ferreira. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[95] M. Robert. Event-Oriented Dynamic Adaption of Workflows: Model, Architecture and Implementation. PhD Dissertation, der Universitut Leipzig, January 2003.

[96] I. Robinson. J2EE Activity Service Specification. IBM Corp., Technical Report, No. JSR-095, 2002.

[97] M. Rusinkiewicz and D. Georgakopoulos. From Transactions to Transactional Workflows: Technology and Applications. In *Proceedings of the First International Workshop on Advances in Databases and Information Systems*, pages 118–119, May 1994.

[98] M. Rusinkiewicz and A. Sheth. *Specification and Execution for Transactional Workflows*. Addison-Wesley, 1994.

[99] M. Rusinkiewicz and A. P. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems*, pages 592–620, 1995.

[100] S. Sadiq. Workflows in Dynamic Environment – Can They Be Managed? In *Proceedings of the Second International Symposium on Cooperative Database Systems for Advanced Applications*, Wollongong, Australia, March 1999.

[101] P. Samarati and S. Vimercati. Access Control: Policies, Models and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design (Tutorial Lectures)*, pages 137–196, September 2000.

[102] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[103] E. A. Schneider, W. Kalsow, L. TeWinkel, and M. Carney. Experimentation with Adaptive Security Policies. Technical Report RL-TR-96-82, Rome Laboratory, Air Force Material Command, Rome, NY, June 1996.

[104] E. A. Schneider, D. G. Weber, and T. de Groot. Temporal Properties of Distributed Systems. Technical Report RADC-TR-89-376, Rome Air Development Center, Rome, NY, September 1989.

[105] L. Sha, J. P. Lehoczky, and E.D. Jensen. Modular Concurrency Control and Failure Recovery. *IEEE Transactions on Computers*, 37(2):146–159, February 1988.

[106] P. M. Shwarz and A. Z. Spector. Synchronizing Shared Abstract Types. *ACM Transactions on Computer Systems*, 2-3:223–250, August 1984.

[107] E. Sibley. Experiments in Organizational Policy Representation: Results to Date. In *Proceedings of the IEEE International Conference on Systems Man and Cybernetics*, pages 337–342, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[108] E. Sibley, J. Michael, and R. Wexelblat. Use of an Experimental Policy Workbench: Description and Preliminary Results. In C. Landwehr and S. Jajodia, editors, *Database Security V: Status and Prospects*, pages 47–76. Elsevier Science Publishers, 1992.

[109] M. P. Singh. Semantical Considerations on Workflows: An Algebra for Intertask Dependencies. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, Electronic Workshops in Computing. Springer, 1995.

[110] R. Sobhan and B. Panda. Reorganization of the Database Log for Information Warfare Data Recovery. In *Proceedings of the Fifteenth IFIP WG11.3 Working Conference on Data and Application Security*, pages 121–134, Niagara, Ontario, Canada, July 2001.

[111] J. Tang and J. Veijalainen. Enforcing Intertask Dependencies in Transactional Workflows. Research Report No. J-2/95. VTT Information Technology, Finland, January 1995.

[112] A. Thomasian. Concurrency Control: Methods, Performance and Analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[113] W.M.P. van der Aalst, T. Basten, H.M.W. Verbeek, P.A.C. Verkoulen, and M. Voorhoeve. Adaptive Workflow. In J.B.L. Filipe, editor, *Enterprise Information Systems*. Kluwer Academic Publishers, 1999.

[114] J. Warne. Extended Transaction Framework. ANSA WorkProgramme, Technical Report APM.1084.00.05, October 1993.

[115] J. Warne. Flexible Transaction Framework for Dependable Workflows. ANSA WorkProgramme, APM.1263.01, June 1994.

[116] H. Wuchter and A. Reuter. The ConTract Model. In *Database Transaction Models for Advanced Applications, A. K. Elmagarmid Ed., Morgan Kaufmann Publishers*, pages 219–263, 1992.

[117] T. Xin and I. Ray. A Lattice-Based Approach for Updating Access Control Policy in Real-Time. *Information Systems*, 2005. Accepted under minor revision.

[118] T. Xin and I. Ray. Detecting Dependency Conflicts in Advanced Transaction Models. In *Proceedings of the Ninth International Database Applications and Engineering Symposium*, Montreal, Canada, July 2005.

[119] T. Xin, Y. Zhu, and I. Ray. Reliable Scheduling of Advanced Transactions. In *Proceedings of the Nineteenth IFIP WG11.3 Working Conference on Data and Applications Security*, Storrs, Connecticut, August 2005.

[120] M. Yu, P. Liu, and W. Zang. Multi-Version Attack Recovery for Workflow Systems. In *Proceedings of the Nineteenth Annual Computer Security Applications Conference*, pages 142–151, December 2003.

[121] Y. Zhu, T. Xin, and I. Ray. Recovering from Malicious Attacks in Workflow Systems. In *Proceedings of the Sixteenth International Conference on Database and Expert Systems*, Copenhagen, Denmark, August 2005.

[122] Y. Zuo and B. Panda. Damage Discovery in Distributed Database Systems. In *Proceedings of the Eighteenth IFIP WG11.3 Working Conference on Data and Applications Security*, July 2004.