Reliable Scheduling of Advanced Transactions

Tai Xin, Yajie Zhu and Indrakshi Ray

Department of Computer Science
Colorado State University
{xin,zhuy,iray}@cs.colostate.edu

Abstract. The traditional transaction processing model is not suitable for many advanced applications, such as those having long duration or those consisting of co-operating activities. Researchers have addressed this problem by proposing various new transaction models capable of processing advanced transactions. Advanced transactions are characterized by having a number of component subtransactions whose execution is 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. In this paper, we focus on the problem of scheduling advanced transactions. Specifically, we show how the different dependencies constrain the execution of the advanced transaction and give algorithms for scheduling advanced transactions that preserve the dependencies. Our scheduler is not confined to any specific advanced transaction processing model, but is capable of handling different kinds of advanced transactions, such as, Saga, Nested Transactions and Workflow.

1 Introduction

Driven by the need for designing high performance and non-traditional applications, a number of advanced transaction models [2, 7, 9, 12, 17, 19] have been proposed in recent years as extensions to the traditional flat transaction model. These advanced transaction models, though differ in forms and applicable environments, have two common properties: made up of *long running activities* and containing *highly cooperative activities*. We refer to these activities as subtransactions in this paper. Subtransactions need to be coordinated to accomplish a specific task. The coordination among subtransactions is achieved through *dependencies*. Existing research work in advanced transactions, like ACTA [8] and ASSET [6], have discussed dependencies as means to characterize the semantics of interactions between subtransactions. Using these dependencies, different kinds of advanced transactions can be generated. Although a lot of research appears in advanced transactions, reliable scheduling and execution have not been adequately addressed.

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_1 and T_2, which requires that T_2 cannot begin until T_1 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_3 and T_4 that requires T_4 to commit if T_3 does so. Suppose the scheduler executes and commits T_3 before T_4. Later if T_4 needs to be aborted for some reason, then we have a complex situation: T_4 needs to abort as well as commit. In such a case, allowing T_4 to complete will cause integrity problems and keeping it incomplete raises issues pertaining to availability.

In this paper, we propose a solution that overcomes the problems mentioned above. We first evaluate the scheduling constraints imposed by each dependency. We discuss the data structures needed by the scheduler, and give the detailed algorithm. In some situations, each pair of subtransactions can be related by multiple dependencies. We show how our algorithm can be extended to handle such scenarios. Note that, our scheduler is extremely general – it can be used for processing any advanced transaction where the transaction can be decomposed into subtransactions that are co-ordinated through dependencies.

The rest of the paper is organized as follows. Section 2 defines our advanced transaction processing model and describes the different kinds of dependencies that may be associated with it. Section 3 describes the different data structures needed by our scheduler. Section 4 presents the details of how an advanced transaction is scheduled by our model. Section 5 discusses related work. Section 6 concludes the paper with pointers to future directions.

2 Our Model for Advanced Transactions

Our definition of advanced transaction is very general; it can be customized for different kinds of transaction models by restricting the type of dependencies that can exist among component subtransactions. An advanced transaction AT is specified by the set of subtransactions in AT, the dependencies between these subtransactions, and the completion sets to specify the complete execution states. All subtransactions specified in an advanced transaction may not execute or commit. A completion set gives the set of transactions that needs to be committed for successfully completing the advanced transaction. The application semantics decides which subtransactions constitute a completion set. The set of subtransactions that commit in an advanced transaction model may vary with different instantiations of the advanced transaction. Thus, an advanced transaction may have multiple completion sets. With this background, we are now ready to formally define our notion of advanced transaction.

Definition 1
[Advanced Transaction] An *advanced transaction* $AT =< S, D, C >$ is defined by S, which is the set of subtransactions in AT, D, which is the set of dependencies between the subtransactions in S, and C, which is the set of completion sets in AT. We assume that the set of dependencies in D do not conflict with each other.

Definition 2
[Subtransaction] A *subtransaction* T_i is the smallest logical unit of work in an advanced transaction. It consists of a set of data operations (read and write) and transac-

tion primitives; the begin, abort and commit primitives of subtransaction T_i are denoted by b_i, a_i and c_i respectively.

Definition 3
[Dependency] A *dependency* specified between a pair of subtransactions T_i and T_j expresses how the execution of a primitive (begin, commit, and abort) of T_i causes (or relates to) the execution of the primitives (begin, commit and abort) of another subtransaction T_j.

A set of dependencies has been defined in the work of ACTA [8]. A comprehensive list of transaction dependency definitions can be found in [3, 6, 8, 14]. Summarizing all these dependencies in previous work, we collect a total of fourteen different types of dependencies. These are given below. In the following descriptions T_i and T_j refer to the transactions and b_i, c_i, a_i refer to the events of T_i that are present in some history H, and the notation $e_i \prec e_j$ denotes that event e_i precedes event e_j in the history H.

[Commit dependency] ($T_i \rightarrow_c T_j$): If both T_i and T_j commit then the commitment of T_i precedes the commitment of T_j. Formally, $c_i \Rightarrow (c_j \Rightarrow (c_i \prec c_j))$.
[Strong commit dependency] ($T_i \rightarrow_{sc} T_j$): If T_i commits then T_j also commits. Formally, $c_i \Rightarrow c_j$.
[Abort dependency] ($T_i \rightarrow_a T_j$): If T_i aborts then T_j aborts. Formally, $a_i \Rightarrow a_j$.
[Termination dependency] ($T_i \rightarrow_t T_j$): Subtransaction T_j cannot commit or abort until T_i either commits or aborts. Formally, $e_j \Rightarrow e_i \prec e_j$, where $e_i \in \{c_i, a_i\}$, $e_j \in \{c_j, a_j\}$.
[Exclusion dependency] ($T_i \rightarrow_{ex} T_j$): If T_i commits and T_j has begun executing, then T_j aborts. Formally, $c_i \Rightarrow (b_j \Rightarrow a_j)$.
[Force-commit-on-abort dependency] ($T_i \rightarrow_{fca} T_j$): If T_i aborts, T_j commits. Formally, $a_i \Rightarrow c_j$.
[Force-begin-on-commit/abort/begin/termination dependency] ($T_i \rightarrow_{fbc/fba/fbb/fbt} T_j$): Subtransaction T_j must begin if T_i commits(aborts/begins/terminates). Formally, $c_i(a_i/b_i/T_i) \Rightarrow b_j$.
[Begin dependency] ($T_i \rightarrow_b T_j$): Subtransaction T_j cannot begin execution until T_i has begun. Formally, $b_j \Rightarrow (b_i \prec b_j)$.
[Serial dependency] ($T_i \rightarrow_s T_j$): Subtransaction T_j cannot begin execution until T_i either commits or aborts. Formally, $b_j \Rightarrow (e_i \prec b_j)$ where $e_i \in \{c_i, a_i\}$.
[Begin-on-commit dependency] ($T_i \rightarrow_{bc} T_j$): Subtransaction T_j cannot begin until T_i commits. Formally, $b_j \Rightarrow (c_i \prec b_j)$.
[Begin-on-abort dependency] ($T_i \rightarrow_{ba} T_j$): Subtransaction T_j cannot begin until T_i aborts. Formally, $b_j \Rightarrow (a_i \prec b_j)$.

Let's see an example of an advanced transaction below.

Example 1
Let $AT = < S, D, C >$ be an advanced transaction where $S = \{T_1, T_2, T_3, T_4\}$, $D = \{T_1 \rightarrow_{bc} T_2, T_1 \rightarrow_{bc} T_3, T_2 \rightarrow_{ex} T_3, T_2 \rightarrow_a T_4\}$, and $C = \{\{T_1, T_2, T_4\}, \{T_1, T_3\}\}$. Thus, this transaction has two complete execution states: $\{T_1, T_2, T_4\}$ and $\{T_1, T_3\}$. The advanced transaction can be represented graphically as shown in Figure 1.

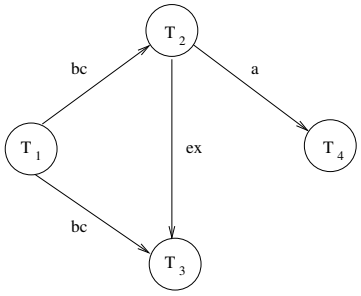

Fig. 1. Dependencies in the Advanced Transaction of Example 1

A real world example of such a transaction may be a workflow associated with making travel arrangements: The subtransactions perform the following tasks. T_1 – Reserve a ticket on Airlines A; T_2 – Purchase the Airlines A ticket; T_3 – Cancels the reservation, and T_4 – Reserves a room in Resort C. There is a *begin-on-commit* dependency between T_1 and T_2 and also between T_1 and T_3. This means that neither T_2 or T_3 can start before T_1 has committed. This ensures that the airlines ticket cannot be purchased or canceled before a reservation has been made. The *exclusion* dependency between T_2 and T_3 ensures that either T_2 can commit or T_3 can commit but not both. In other words, either the airlines ticket must be purchased or the airlines reservation canceled, but not both. And, there is an *abort* dependency between T_4 and T_2 - This means that if T_2 aborts then T_4 must abort. In other words, if the resort room cannot be reserved, then the airlines ticket should not be purchased.

Sometimes one single dependency is not adequate for specifying the relationship between two subtransactions. For example, if we want to specify that (i) T_1 must begin after T_2 has committed and (ii) if T_2 aborts then T_1 must also abort. In such cases, a single dependency is not sufficient for expressing the co-ordination relationship between T_1 and T_2. 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 could generate a composite dependency: $T_2 \rightarrow_{fbc,a} T_1$.

Definition 4
[Composite Dependency] A composite dependency between a pair of subtransactions T_i, T_j in an advanced transaction, denoted by $T_i \rightarrow_{d_1,d_2,\ldots,d_n} T_j$, 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. In this paper, we assume that the advanced transaction specification does not have such conflicts.

2.1 Execution Model

Having presented the structural model of the advanced transaction, we now present our execution model. A subtransaction can be at different states during its lifetime. Rusinkiewicz and Sheth have discussed the states of workflow tasks in a similar manner [17]. In this paper, our approach will extend the their work. We will have the *unscheduled* state to identify that a subtransaction has not been submitted, and, we will require the subtransactions being hold in *prepare* state and cannot transit to final (commit or abort) state until the dependencies have been satisfied.

state specifications suitable for the advanced transaction and dependencies.

Definition 5
[State of a subtransaction] A subtransaction T_i can be in any of the following states: *unscheduled* (un_i), *initiation* (in_i), *execution* (ex_i), *prepare* (pr_i) (means prepare to commit), *committed* (cm_i) and *aborted* (ab_i). Execution of subtransaction primitives causes a subtransaction to change its state. Detailed state transition diagrams are shown in figure 2.

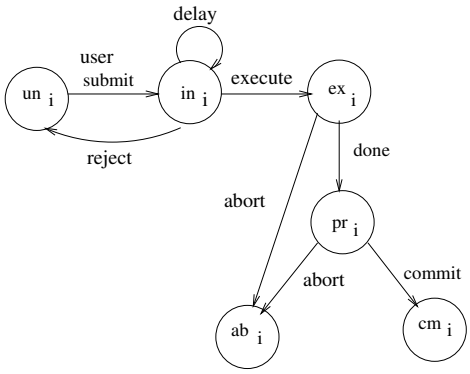

Fig. 2. States of subtransaction T_i

Below we formally define each state, and describe how and when state transitions take place.

- **unscheduled** (un_i), means a subtransaction (T_i) is not sent to a scheduler. At this point, a scheduler can do nothing about it.
- **initiation** (in_i), After subtransaction (T_i) is sent to the scheduler, its state changes to initiation. Now it is waiting to be executed. Later the scheduler can execute, delay or reject this subtransaction.
- **execution** (ex_i), a subtransaction (T_i) 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_i), After a subtransaction (T_i) finishes its execution and 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_i), means a subtransaction (T_i) has committed.
- **aborted** (ab_i), means a subtransaction (T_i) has aborted. There are two ways to enter the aborted state. When a subtransaction is in the execution state, it may be aborted. Also when it is in the prepare state, the scheduler can abort it.

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.

A reliable scheduler of an advanced transaction must be able to complete all the necessary subtransactions in an advanced transaction and not cause any dependency violation. A subtransaction that is never terminated but remains active even after the transaction has terminated is called an orphan subtransaction.

Definition 6
[Reliable Scheduling] The scheduling of an advanced transaction is *reliable* if it satisfies the following constraints.

1. all dependency constraints of the advanced transaction must be satisfied;
2. when execution completes, each subtransaction must be in final state (a committed/aborted state) or unscheduled state. In other words, when execution of an advanced transaction completes, there should be no orphan subtransaction. Notice that an orphan subtransaction will hold resources and possibly cause availability problems.

The above conditions are necessary to avoid availability and integrity problems caused by the advanced transaction.

3 Data Structures required by the Scheduler

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

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 $T_i \rightarrow_x T_j$ depends on the type of dependency existing between T_i and T_j and the states of T_i and T_j. This information is stored in the *scheduling action table*. For each dependency of the form $T_i \rightarrow_x T_j$, we construct a scheduling action table TB_x. This table has six rows and six columns corresponding to the different states of T_i and T_j respectively. An entry in this table is denoted as $EN_x(i, j)$ where i represents a state of the subtransaction (T_i), and j represents a state of the subtransaction (T_j). 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_i and T_j. The subtransactions T_i or T_j can go into the next state without any restriction.

2. *delay $T_i(T_j)$* means that the subtransaction $T_i(T_j)$ cannot make a state transition at this stage. It must wait at the current state until the other subtransaction $T_j(T_i)$ has entered another state.
3. *execute $T_i(T_j)$* means that the subtransaction $T_i(T_j)$ must be executed.
4. *abort $T_i(T_j)$* means that the subtransaction $T_i(T_j)$ must be aborted.
5. *reject $T_i(T_j)$* means that the subtransaction $T_i(T_j)$ will be rejected instead of being scheduled for execution. This entry is only possible when subtransaction $T_i(T_j)$ 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 dependencies. For lack of space, we do not give the tables for all the dependencies. Table 1 shows the scheduling action table for the strong commit dependency.

action	un_j	in_j	ex_j	pr_j	cm_j	ab_j
un_i	–	–	–	–	–	–
in_i	–	–	–	–	–	–
ex_i	–	–	–	–	–	abort T_i
pr_i	delay T_i	delay T_i	delay T_i	delay T_i	–	abort T_i
cm_i	/	/	/	/	final	/
ab_i	–	–	–	–	final	final

Table 1. Scheduling action table for *strong commit* dependency

The first row of the table specifies the actions to be taken when T_i is in the unscheduled state. In this row all the entries are marked with '–' indicating that the scheduler does not impose any constraint on T_i or T_j changing states. The entry in the third row, last column (that is, $EN(ex_i, ab_j)$) is an 'abort T_i'. This means that when T_i is in the execution state, and T_j is aborted, then T_i must be aborted as well. The entry in the fourth row, first column (that is, $EN(pr_i, un_j)$) is 'delay T_i'. This means that when T_i is in the prepare state and T_j is unscheduled, T_i must wait in the prepare stage. The entry in the fifth row, first column (that is, $EN(cm_i, un_j)$) is '/'. This means that the scheduler will not allow this to happen. The entry in the fifth row, fifth column (that is, $EN(cm_i, cm_j)$) is 'final'. This means that both the transactions have reached their final states, and the scheduler need not do anything more.

The following ensures the correctness of our scheduling action table. For lack of space, we omit the proof.

Lemma 1. *The scheduler by taking the actions listed in the scheduling action tables for the primitive dependencies can enforce the dependencies correctly.*

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 table for composite dependencies. This table is called the *composite dependency scheduling table*. The composite dependency scheduling table for the composite dependency consisting of primitive dependencies x, y, and z is denoted by $TBC_{x,y,z}$.

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 the 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 1
Creating Composite Dependency Scheduling Table
Input: (i) $T_i \rightarrow_{d_1,d_2,...,d_n} T_j$ – the composite dependency composed of the primitive dependencies $d_1, d_2, ..., \ldots d_n$ 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.

begin
  for each state (S_i) of subtransaction (T_i) $\in \{un_i, in_i, ex_i, pr_i, ab_i, cm_i\}$
    for each state (S_j) of subtransaction (T_j) $\in \{un_j, in_j, ex_j, pr_j, ab_j, cm_j\}$
    begin
      /* initialization */
      $EN_{TBC}(S_i, S_j)$ = "–"
      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 $EN_k(S_i, S_j)$
        /* finding the highest priority entry in these dependencies */
        if $EN_k(S_i, S_j) > max_p$

$max_p = EN_k(S_i, S_j)$
end for
$EN_{TBC}(S_i, S_j) = max_p$
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.

Lemma 2. *The scheduler can enforce composite dependencies correctly.*

3.3 State Table and Job Queue

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.

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.

4 Execution of an Advanced Transaction

In this section we describe how an advanced transaction is executed. The advanced transaction is executed in three stages: (i) Preparation Stage, (ii) Execution Stage, and (iii) Termination Stage. These stages are described in the following subsections.

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 2
InputAdvancedTransaction
Input: (i) $AT_t =< S, D, C >$ – the advanced transaction.

Procedure InputAdvancedTransaction(AT_t)
begin
receive the input $AT_t =< S, D, C >$
create $StateTable_t$
for each $T_i \in S$
begin
$StateTable_t[T_i] = initiation$ /* set initial states for subtransactions */
enQueue(JobQueue, T_i) /* insert in the job queue */
end for
end

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 identifies the states of the two involved subtransactions. The scheduler then accesses the dependency scheduling action table, and gets the required action for the subtransactions. The action can be one of the following: allow it to commit/abort, send the transaction to execute, delay the subtransaction, and reject the transaction. 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 3
Execution Stage
Input: (i) $AT_t =< S, D, C >$ – 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_t.

Procedure ExecuteAdvancedTransaction(AT_t, **TB**)
begin
while(TRUE)
begin
T_i = deQueue(JobQueue) /* get the job from the job queue */
$Action$ = getAction(T_i, AT_t, **TB**)
if $Action = wait$
enQueue(JobQueue, T_i) /* insert in queue */
else if $Action = abort$
abort T_i
$StateTable_t[T_i] = aborted$
else if $Action = reject$
$StateTable_t[T_i] = unscheduled$
else if $Action = -$ /* no restriction for T_i */

```
            begin
                if StateTable_t[T_i] = initiation
                  send (T_i) to execute      /* execute the operations for T_i */
                  StateTable_t[T_i] = executing
                else if StateTable_t[T_i] = executing
                        get execution results
                        if execution result is completed      /* operations completed */
                          StateTable_t[T_i] = prepare
                          enQueue(JobQueue, T_i)
                        else if execution failed      /* operations failed */
                                StateTable_t[T_i] = aborted
                else if StateTable_t[T_i] = prepare
                        commit T_i
                        StateTable_t[T_i] = committed
            end
        end while
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_i), focusing on ensuring the dependency constrains associated with T_i. We assume that the primitive and composite dependency tables that will be needed by this advanced transaction have already been created.

Algorithm 4
Get Action From ActionTables
Input: (i) T_i – the subtransaction for which the action must be determined, and (ii) $AT_t =< S, D, C >$ – the advanced transaction whose subtransaction is T_i, and (iii) **TB** – the set of primitive and composite scheduling action tables associated with the dependencies of the advanced transaction AT_t.
Output: The action the scheduler should take to for subtransaction T_i

```
Procedure getAction(T_i, AT_t, TB)
begin
    ACTION = '–' /* Initialize ACTION */
    /* find out all the dependencies associated with T_i */
    for every dependency T_m →_d T_n ∈ D
    begin
        if (T_i ≠ T_m) AND (T_i ≠ T_n)
          skip this round, and continue to next round
        else /* this dependency is associated with T_i */
        begin
            if (T_i = T_n) /* d is a dependency pointed to T_i */
              /* get the state of the subtransactions */
              let S_y = StateTable_t[T_i]
```

```
                let S_x = StateTable_t[T_m]
            else if (T_i = T_m) /* d is a dependency that T_i lead out */
                    /* get the state of the subtransactions */
                    let S_x = StateTable_t[T_i]
                    let S_y = StateTable_t[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
    return ACTION;
end
```

4.3 Termination Stage

When all the subtransactions of an advanced transaction have completed execution, the advanced transaction must be terminated. From the state tables, we find out the set of executing, committed and prepared subtransactions. If the set of prepared or executing subtransactions is not empty, then we return the message not terminated. Otherwise, we check whether the set of committed transactions correspond to one of the completion sets specified in the advanced transaction. If so, we return a successful termination message, otherwise we return an unsuccessful termination message. Once the advanced transaction is terminated, the state table corresponding to the advanced transaction is deleted.

Algorithm 5
Termination Stage
Input: (i) $AT_t =< S, D, C >$ – the advanced transaction whose termination is being determined.
Output: (i) result indicating whether the advanced transaction terminated successfully or not.

```
Procedure TerminateAdvancedTransaction(AT_t)
begin
    executing = prepared = committed = {}
    for each subtransaction T_i ∈ S
    begin
        if StateTable_t[T_i] = committed
          committed = committed ∪ T_i
        else if StateTable_t[T_i] = executing
                executing = executing ∪ T_i
        else if StateTable_t[T_i] = prepared
                prepared = prepared ∪ T_i
    end
    /* check whether there are active subtransactions for AT_t */
```

```
    if prepared ≠ {} OR committed ≠ {}
      return 'not terminated'
    else /* all subtransactions are finished */
    begin
        Delete StateTable_t
        /* check whether it matches some completion set */
        for each C_i ∈ C
        begin
            if (C_i = committed)
              return 'terminated successfully'
        end
        /* none of the termination states are satisfied */
        return 'terminated unsuccessfully'
    end
end
```

The following theorem ensures the correctness of the mechanisms.

Theorem 1
The mechanism described above ensures reliable scheduling as per Definition 6.

5 Related Work

In the past two decades, a variety of transaction models and technologies supporting advanced transaction have been proposed. Examples are ACTA [8], ConTracts [16], nested transactions [12], ASSET [6], EJB and CORBA object transaction services [13], workflow management systems [1], concurrency control in advanced databases [5] etc. Chrysanthis and Ramamrithan introduce ACTA [8], 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. However, impacts of dependencies on reliable execution of advanced transactions are not discussed in ACTA.

Mancini, Ray, Jajodia and Bertino have proposed the notion of multiform transactions [11]. 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. The coordinate block, along with the corresponding coordinator module (CM) can manage the execution of the transactions.

Workflows are composite activities typically involving a variety of computational and business activities. Workflow is a type of advanced transaction, since all the workflow models support coordination of a sub units of activities. The importance of workflow models is increasing rapidly due to its suitability in the business application. For these reasons, a lot of research appears in workflow management systems [1, 3, 10, 15].

Singh has discussed the semantical inter-task dependencies on workflows [18]. The author used algebra format to express the dependencies and analyze their properties and semantics in workflow systems. Attie at el. [4] discussed means to specify and enforce intertask dependencies. They illustrate each task as a set of significant events (start, commit, rollback, abort). Intertask dependencies limit the occurrence of such events and specify a temporal order among them. In an earlier work, Rusinkiewicz and Sheth [17] have discussed the specification and execution issues of transactional workflows. They have described the different states of tasks in execution for a workflow system. They also discussed different scheduling approaches, like: scheduler based on predicate Petri Nets models, scheduling using logically parallel language, or using temporal propositional logic. Another contribution of their paper is that they discussed the issues of concurrent execution of workflows - global serializability and global commitment of workflow systems. However, none of these papers address the scheduling actions needed to satisfy the dependency constraints.

6 Conclusion and Future Work

An advanced transaction is composed of a number of cooperating subtransactions that are coordinated by dependencies. The dependencies make the advanced transaction more flexible and powerful. However, incorrect enforcement of dependencies can lead to integrity and availability problems. In this paper, we looked at how the subtransactions of an advanced transaction can be scheduled, such that the dependencies are not violated.

The constraints between the subtransactions of an advanced transaction must be maintained during recovery as well. In future, we would like to investigate how the dependencies impact the recovery algorithms and design a mechanism that is suitable for the recovery of advanced transactions. In future, we also plan to design mechanisms that will allow advanced transactions to recover from malicious attacks.

7 Acknowledgment

This work is partially supported by National Science Foundation under grant number IIS 0242258.

References

1. Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, Mohan Kamath, Roger G., and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *In Proceedings of ICDE 1996*, pages 574–581, 1996.
2. M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proceeding of the 18th International Conference on Very Large DataBases*, August 1992.
3. V. Atluri, W-K. Huang, and E. Bertino. An Execution Model for Multilevel Secure Workflows. In *11th IFIP Working Conference on Database Security and Database Security, XI: Status and Prospects*, pages 151–165, August 1997.

4. Paul C. Attie, Munindar P. Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and enforcing intertask dependencies. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 134–145. Morgan Kaufmann, 1993.
5. Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.
6. 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 Coference on Management of Data*, May 1994.
7. P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19:450–491, September 1994.
8. Panayiotis K. Chrysanthis. ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis, September 1991.
9. U. Dayal, M. Hsu, and R.Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceeding of the 17th International Conference on Very Large DataBases*, September 1991.
10. D. Hollingsworth. Workflow Reference Model. Technical report, Workflow Management Coalition, Brussels, Belgium, 1994.
11. L. V. Mancini, I. Ray, S. Jajodia, and E. Bertino. Flexible transaction dependencies in database systems. *Distributed and Parallel Databases*, 8:399–446, 2000.
12. J. E. Moss. Nested Transactions: an approach to reliable distributed computing. PhD Thesis 260, MIT, Cambridge, MA, April 1981.
13. OMG. Additional Structuring Mechanisms for the OTS Specification. OMG, Document ORBOS, 2000-04-02, Sept. 2000.
14. M. Prochazka. Extending transactions in enterprise javabeans. Tech. Report No. 2000/3, Dep. of SW Engineering, Charles University, Prague, January 2000.
15. Indrakshi Ray, Tai Xin, and Yajie Zhu. Ensuring Task Dependencies During Workflow Recovery. In *Proceedings of the Fifteenth International Conference on Database and Expert Systems*, Aug. 2004.
16. A. Reuter. Contracts: A means for extending control beyond transaction boundaries. In *3rd International Workshop on High Performance Transaction Systems*, Sept. 1989.
17. Marek Rusinkiewicz and Amit P. Sheth. Specification and execution of transactional workflows. In *Modern Database Systems 1995*, pages 592–620, 1995.
18. Munindar 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.
19. Helmut Wuchter and Andreas Reuter. The ConTract Model. In *Database Transaction Models for Advanced Applications, A. K. Elmagarmid Ed., Morgan Kaufmann Publishers*, pages 219–263, 1992.