On the Completion of Workflows*

Tai Xin[1], Indrakshi Ray[1], Parvathi Chundi[2], and Sopak Chaichana[1]

[1] Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873
{xin,iray,chaichan}@cs.colostate.edu
[2] Computer Science Department
University of Nebraska at Omaha
Omaha, NE 68182-0500
pchundi@mail.unomaha.edu

Abstract. Workflow Management Systems (WFMS) coordinate execution of logically related multiple tasks in an organization. A workflow schema is defined using a set of tasks that are coordinated using dependencies. Workflows instantiated from the same schema may differ with respect to the tasks executed. An important issue that must be addressed while designing a workflow is to decide what tasks are needed for the workflow to complete – we refer to this set as the *completion set*. Since different tasks are executed in different workflow instances, a workflow schema may be associated with multiple completion sets. Incorrect specification of completion sets may prohibit some workflow from completing. Manually generating these sets for large workflow schemas can be an error-prone and tedious process. Our goal is to automate this process. We investigate the factors that affect the completion of a workflow. Specifically, we study the impact of control-flow dependencies on completion sets and show how this knowledge can be used for automatically generating these sets. Finally, we provide an algorithm that can be used by application developers to generate the completion sets associated with a workflow schema.

1 Introduction

Workflow management systems (WFMS) recently have gained a lot of attention. They are responsible for coordinating the execution of multiple tasks performed by different entities within an organization. A group of such tasks that forms 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 execution of a workflow must preserve the dependencies and eventually complete. Although a large body of work appears in the area of workflows [1–3, 6–13], very few researchers have addressed the issue of workflow completion. In almost all of these works, the researchers assume that the application developer specifies what is needed for the workflow to complete. However, manually evaluating the conditions needed for workflow completion may be tedious and error-prone. In this paper, we aim to automate this process.

* This work was partially supported by NSF under Award No. IIS 0242258.

A workflow is an instance of some workflow schema. A workflow schema formally specifies the tasks in a workflow and the dependencies between the tasks. Not all tasks specified in a workflow schema may be executed in a workflow instance. We refer to the set of tasks that are needed to complete a workflow instance as the completion set. The set of tasks needed to complete different workflows generated from the same schema may vary because not all instances execute the same set of tasks. Thus, a workflow schema may be associated with multiple completion sets.

Improper specification of completion sets in a workflow may result in deadlock and availability problems. If a completion set violates some dependency constraints, the states required for the workflow to complete can never be reached, and the workflow will be in the execution state infinitely. It will hold resources forever, and cause deadlock and/or unavailability problems. Let us illustrate one such problem with a simple example. Consider a workflow W_w that has a large number of tasks and dependencies. Specifically, there is an *exclusion* dependency existing between tasks T_{wm} and T_{wk}, which requires that T_{wk} must abort if T_{wm} commits. In other words, it is not possible for both these tasks to commit in the same instance. This implies that T_{wm} and T_{wk} can never be placed in the same completion set. Now, suppose an application developer, who is specifying completion sets for this workflow, overlooks this dependency and places both T_{wm} and T_{wk} in the same completion set CT_t. The consequence is that this workflow will never complete and the resource availability of the system will be compromised.

Due to the large number of tasks and dependencies that can exist in a complex workflow, correctly calculating all the completion sets manually can be a time-consuming and error-prone work. To solve this problem, we need an approach that generates all the possible completion sets automatically from a given workflow schema. In this paper, we propose one such approach. We begin by identifying the impact of dependencies on the completion set of a workflow. We then show how the knowledge of the dependencies can be exploited to generate the completion sets. Every completion set produced by our approach will satisfy the dependency constraints and it will be feasible to execute all the tasks in a given completion set. This, in turn, will ensure that the workflow completes.

The rest of the paper is organized as follows. Section 2 defines our workflow processing model and describes the different kinds of dependencies that may be associated with it. Section 3 presents the details of the algorithm and shows how to generate all the possible completion sets correctly. Section 4 concludes the paper with pointers to future directions.

2 Our Model for Workflows

We begin by giving some definitions.

Definition 1. [Workflow Schema] *A* workflow schema *W_w is a triple $< S, D, C >$, where, S is a set of tasks, D is the set of dependencies used to coordinate the execution of the tasks in S, and C is the set of completion sets in W_w, which defines the conditions for complete execution of W_w.*

Definition 2. [Task] *A* task *T_{wi} is the smallest logical unit of work in an workflow. It consists of a set of data operations (read and write) and task primitives (begin, abort and commit).*

The begin, abort and commit primitives of task 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.

Definition 3. [Completion Set] *Each* completion set $C_t \in C$ *is specified by* $(CT_t, \ll_t)$, *where* CT_t *is the set of tasks 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 tasks which must be committed for this workflow to complete, and* $\ll_t$ *is the ordering relation that specifies the commit order of tasks in* CT_t.

In order to properly coordinate the different tasks in a workflow, control-flow dependencies are specified on the task primitives, which control the occurrence and/or ordering of the *begin, commit, and abort* events of different tasks.

Definition 4. [Control-flow Dependency] *A* control-flow dependency *specified between a pair of tasks* 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 task* T_{ik}.

A comprehensive list of transaction dependency definitions can be found in [2,4,5]. We describe the fifteen commonly used dependencies. In the following descriptions T_{ij} and T_{ik} refer to the tasks and b_{ij}, c_{ij}, a_{ij} refer to the events of T_{ij} 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]** ($T_{ij} \rightarrow_t T_{ik}$): Task 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]** ($T_{ij} \rightarrow_{fbc/fba/fbb/fbt} T_{ik}$): Task 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]** ($T_{ij} \rightarrow_b T_{ik}$): Task T_{ik} cannot begin execution until T_{ij} has begun. Formally, $b_{ik} \Rightarrow (b_{ij} \prec b_{ik})$.
- **[Serial dependency]** ($T_{ij} \rightarrow_s T_{ik}$): Task 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]** ($T_{ij} \rightarrow_{bc} T_{ik}$): Task T_{ik} cannot begin until T_{ij} commits. Formally, $b_{ik} \Rightarrow (c_{ij} \prec b_{ik})$.

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

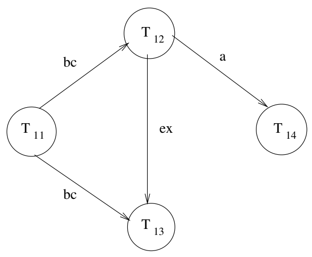

Fig. 1. Dependencies in the Example Workflow

A workflow W_w can be represented in the form of a graph $G_w =< V,E >$ which we term as the *dependency graph*. The task $T_{w1}, T_{w2}, \ldots, T_{wn}$ 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. Let $W_1 =< S,D,C >$ be a workflow where $S = \{T_{11}, T_{12}, T_{13}, T_{14}\}$, $D = \{T_{11} \rightarrow_{bc} T_{12}, T_{11} \rightarrow_{bc} T_{13}, T_{12} \rightarrow_{ex} T_{13}, T_{12} \rightarrow_{a} T_{14}\}$, and $C = \{(\{T_{11}, T_{12}, T_{14}\}, \{T_{11} \ll T_{12}\}), (\{T_{11}, T_{13}\}, \{T_{11} \ll T_{13}\})\}$. The labels on each edge corresponds to the dependencies that exist between the tasks. $L(T_{11}, T_{12}) = \{bc\}$, $L(T_{11}, T_{13}) = \{bc\}$, $L(T_{12}, T_{13}) = \{ex\}$, $L(T_{12}, T_{14}) = \{a\}$. This transaction has two completion sets: $(\{T_{11}, T_{12}, T_{14}\}, \{T_{11} \ll T_{12}\})$ and $(\{T_{11}, T_{13}\}, \{T_{11} \ll T_{13}\})$. This transaction can be represented graphically as shown in Figure 1. A real world example of such a transaction may be a workflow associated with making travel arrangements. The tasks perform the following tasks. (i) Task T_{11} – Reserve a ticket on Airlines A, (ii) Task T_{12} – Purchase the Airlines A ticket, (iii) Task T_{13} – Cancels the reservation, and (iv) Task T_{14} – Reserves a room in Resort C. There is a *begin-on-commit* dependency between T_{11} and T_{12} and also between T_{11} and T_{13}. This means that neither T_{12} nor T_{13} can start before T_{11} has committed. This ensures that the airlines ticket cannot be purchased or canceled before a reservation has been made. The *exclusion* dependency between T_{12} and T_{13} ensures that either T_{12} can commit or T_{13} can commit but not both. In other words, either the airlines ticket must be purchased or the airlines reservation canceled, but not both. Finally, there is an *abort* dependency between T_{14} and T_{12}. This means that if T_{12} aborts then T_{14} must abort. In other words, if the resort room cannot be reserved, then the airlines ticket should not be purchased.

3 Generate the Completion Sets Automatically

Impact of Dependencies on Completion Sets: Different control-flow dependencies have different impacts on deciding the completion set. For instance, with a *begin-on-abort* dependency $T_{wi} \rightarrow_{ba} T_{wj}$, T_{wj} cannot begin and hence it cannot commit without the abort event of T_{wi} in the history. Therefore, if one wants to have T_{wj} in a completion set CT_t, T_{wi} cannot be in the same completion set – $T_{wj} \in CT_t \Rightarrow T_{wi} \notin CT_t$. The control-flow dependencies that impact a completion set CT_t are listed in Table 1.

Dependency	Impact
$T_{wi} \rightarrow_{sc} T_{wj}$	$T_{wi} \in CT_t \Rightarrow T_{wj} \in CT_t$
$T_{wi} \rightarrow_{a} T_{wj}$	$T_{wj} \in CT_t \Rightarrow T_{wi} \in CT_t \wedge T_{wj} \prec T_{wi}$
$T_{wi} \rightarrow_{ex} T_{wj}$	$T_{wi} \in CT_t \Rightarrow T_{wj} \notin CT_t$
$T_{wi} \rightarrow_{fca} T_{wj}$	$T_{wi} \in CT_t \Rightarrow T_{wj} \in CT_t \wedge T_{wj} \prec T_{wi}$
$T_{wi} \rightarrow_{c} T_{wj}$	$T_{wi} \in CT_t \wedge T_{wj} \in CT_t \Rightarrow T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_{t} T_{wj}$	$T_{wi} \in CT_t \wedge T_{wj} \in CT_t \Rightarrow T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_{s} T_{wj}$	$T_{wi} \in CT_t \wedge T_{wj} \in CT_t \Rightarrow T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_{bc} T_{wj}$	$T_{wj} \in CT_t \Rightarrow T_{wi} \in CT_t \wedge T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_{ba} T_{wj}$	$T_{wj} \in CT_t \Rightarrow T_{wi} \notin CT_t$

Table 1. Impacts of Dependencies on Deciding Completion Sets

Strategies for Generating Completion Sets: The algorithm for generating completion sets automatically will use the graph representing the workflow. The dependency graph will include both the directly given dependencies and the implicit dependencies. Implicit dependencies [11] arise because of the interaction of the given dependencies. For instance, the $T_{wi} \rightarrow_{sc} T_{wk}$ and $T_{wk} \rightarrow_{ex} T_{wm}$ dependencies will imply an implicit dependency $T_{wi} \rightarrow_{ex} T_{wm}$. An example of the dependency graph is shown in Figure 2. The steps for generating completion set for this workflow are shown in Figure 3.

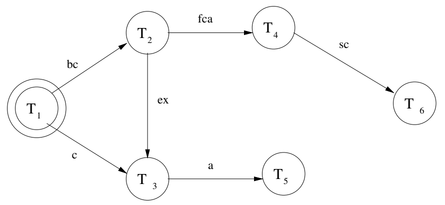

Fig. 2. An Example of Workflow and its Dependency Graph

We assume that every dependency graph has a start node which is the one that has no incoming edges. For instance, in Figure 2, the task T_1 is the start node. We assume that the start node represents the first task to be executed in a workflow and is present in all completion sets. When generating completion sets for the workflow, we construct a tree structure. Each node of this tree is associated with a set of tasks. This set represents the prefix of a completion set which we obtained by traversing the graph so far. The root of this tree contains the start node of the graph. Each leaf node represents a completion set. The strategy for computing completion sets is described below.

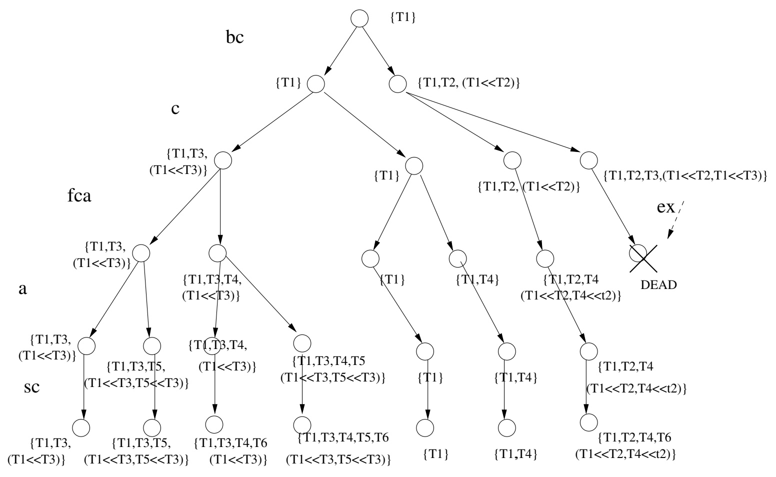

Fig. 3. Generate Completion Sets for Workflow in Figure 2

(i) We insert the start node into the root of the tree as shown in Figure 3.
(ii) We consider one dependency $T_{wm} \rightarrow_d T_{wn}$ at a time, and insert new child node(s) into the tree. We add the task T_{wm} or T_{wn} into the completion set of the child node(s), if it does not violate the dependency constraints. We use Table 1 for this purpose.

1. If it is a *strong commit* or a *force-commit-on-abort* dependency and the completion set contains T_{wm}, then we only generate one child node, where we insert T_{wn} into the existing completion set. For the force-commit-on-abort dependency, we also have to add the $T_{wn} \prec T_{wm}$ in its new completion set.
2. If it is an *exclusion* dependency and the completion set contains T_{wm}, we only generate one child since we cannot insert T_{wn}. The child node has the same completion set as the parent node. However, if we already have both T_{wm} and T_{wn} in the completion set, we have to remove this node from the tree since it now contains an incorrect completion. This is shown in the rightmost node in Figure 3.
3. For *commit, begin-on-commit, termination, serial* dependencies, if the completion set contains T_{wm}, we generate two nodes – one that contains T_{wn} and one that does not. With the child that contains T_{wn}, the ordering requirement $T_{wm} \prec T_{wn}$ is inserted into the completion set.
4. If it is an *abort* dependency, and the completion set of this node contains T_{wn}, then we only generate one child node, where T_{wm} is inserted into the completion set, and also has $T_{wn} \prec T_{wm}$ in the new completion set.
5. If it is a *begin-on-abort* dependency, and the completion set of this node contains T_{wn}, we only generate one child node since we cannot insert T_{wm} into the existing completion set. However, if we already have both T_{wm} and T_{wn} in the completion set, we have to remove this node from the tree.
6. For all other dependencies (like *force-begin-on-begin* dependency), if the completion set of this node contains T_{wm}, we will generate two child nodes. One node

contains T_{mn} and one does not, because these dependencies have no impact on completion sets.

(iii) The operations in step (ii) continue until all the dependencies are considered in the dependency graph. Finally we have the complete tree where every leaf node contains one completion set.

Algorithms to Compute Completion Sets Automatically: We next give the algorithm for building all possible completion sets. The algorithm is organized in two steps - (i) build the derived dependency set and (ii) compute all possible completion sets based on the derived dependency set.

In the first step, the specification of the workflow is used to build the initial dependency set (IDS), which records every dependency $T_{wi} \rightarrow_{d_x} T_{wj}$ as an edge of the form (T_{wi}, T_{wj}, d_x), where T_{wi}, T_{wj}, and d_x represent the source node, the destination node, and the dependency respectively. Then the IDS is used to build the derived dependency set (DDS). This procedure has several rounds of iterations. In each iteration, the algorithm scans all the dependencies currently in the DDS and check whether new edges could be implied by the existing edges. The process is repeated until no more new edges can be derived.

```
**Algorithm 1**
**Input:** the workflow specification $AT_t =< S, D, C >$
**Output:** a derived dependency set (DDS) of the workflow
**Procedure** GenerateDDS($AT_t$)
**begin**
    //Build the initial dependency set, according to the specification
    $IDS = \{\}$; // set of edges
    **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 $T_{wj}$
        $IDS = IDS + (T_{wi}, T_{wj}, d_x)$;
    // initiate the derived dependency set
    done == false;
    $DDS == IDS$;
    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 $(T_{wj}, T_{wi}, bc)$;
            // 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 implicit 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 implicit 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
```

Following, we compute all the possible completion sets based on the DDS obtained above. A queue is used to simulate the breadth-first traversal of the tree. The queue maintains the part of completion sets we have obtained so far. We first consider the start node, generate a completion set containing only this task and insert it into the queue. Then we take one dependency at a time, and compute the new set of completion sets that can be obtained by applying this dependency with the existing completion sets in the queue. The rules for computing new set of completion sets are based on Table 1. The newly obtained set will replace all the old completion sets in the queue and the process is repeated until all the dependencies are considered.

```
Algorithm 2
Input: a derived dependency set (DDS) of the workflow
Output: a set containing all possible completion set for this workflow
Procedure GenerateCompletionSet
begin
      create two queues, one completion sets queue, and one temp queue
      generate a completion set $(CT_t, \ll_t)$ where $CT_t = \{T_{ws}\}$, $\ll_t = \{\}$, $T_{ws}$ = start node
      insert this initial completion set into the completion set queue
      for every edge $(T_{wi}, T_{wj}, d_x)$ in the DDS
          /* consider this dependency with every completion set in the queue */
          for every completion set $CT_t$ in the completion set queue
              if $T_{wi} \notin CT_t$ AND $T_{wj} \notin CT_t$
              /* this dependency is not relevant with this completion set */
                 continue;   /* do nothing here */
              else if $T_{wi} \in CT_t$    /* this completion set contains $T_{wi}$ */
                      if $d_x = sc$
                         $CT_t = CT_t \cup \{T_{wj}\}$
                         insert $(CT_t, \ll_t)$ to the temp queue
```

```
                  if $d_x = fca$
                    $CT_t = CT_t \cup \{T_{wj}\}$
                    $\ll_t = \ll_t \cup (\{T_{wj} \ll T_{wi})$
                    insert $(CT_t, \ll_t)$ to the temp queue
                  if $d_x = ex$
                    if $T_{wi} \in CT_t \wedge T_{wj} \in CT_t$
                      remove $(CT_t \ll_t)$, this set is infeasible
                    else   /* cannot have $T_{wj}$ in the new set */
                          insert $(CT_t, \ll_t)$ to the temp queue
                  if $d_x = c$ OR $d_x = bc$ OR $d_x = t$ OR $d_x = s$
                    /* generate two new sets, one contains $T_{wj}$ and one not */
                    insert $(CT_t, ll_t)$ to the temp queue
                    $CT_s = CT_t \cup \{T_{wj}\}$
                    $\ll_s = \ll_t \cup \{T_{wi} \ll T_{wj}\}$
                    insert $(CT_s, \ll_s)$ to the temp queue
                  if $d_x = ba$
                    if $T_{wj} \in CT_t$   /* have both $T_{wi}, T_{wj}$ */
                      remove $(CT_t, \ll_t)$, this set is infeasible
                  else   /* for all other dependencies */
                      /* generate two new sets, one contains $T_{wj}$ and one not */
                      insert $(CT_t, \ll_t)$ to the temp queue
                      $CT_s = CT_t \cup \{T_{wj}\}$
                      insert $(CT_s, \ll_t)$ to the temp queue
            else if $T_{wj} \in CT_t$   /* this completion set contains $T_{wj}$ */
                  if $d_x = a$
                    $CT_t = CT_t \cup \{T_{wi}\}$
                    $\ll_t = \ll_t \cup \{T_{wj} \ll T_{wi}\}$
                    insert $(CT_t, \ll_t)$ to the temp queue
                  if $d_x = ba$
                    insert $(CT_t, \ll_t)$ to the temp queue   /* cannot have $T_{wi}$ in the new set */
        end for
        /* let temp queue be the new completion set queue, clean up the temp queue */
        let completion set queue equals to the temp queue
        reset the temp queue to be empty
    end for
    return the completion set queue
end
```

4 Conclusion

An workflow is composed of a number of cooperating tasks that are coordinated by dependencies. The dependencies make the workflow more flexible and powerful. Completion sets are also specified in the workflow to identify complete executions. However, incorrect specification of completion sets can lead to deadlock and unavailability problems. The completion sets must conform to the dependencies in the workflows. In this paper, we looked at how the dependencies can impact the completion set, and gave an algorithm to generate all possible completion sets automatically. A lot of work remains to be done. We need to give proof of formal correctness. We also need to evaluate the complexity of the algorithm used in computing the completion sets. In future, we plan to provide more efficient algorithms.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. P. Chrysanthis. ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis, September 1991.
6. D. Hollingsworth. Workflow Reference Model. Technical report, Workflow Management Coalition, Brussels, Belgium, 1994.
7. 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.
8. M. Rusinkiewicz and A. P. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems*, pages 592–620, 1995.
9. 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.
10. W.M.P. van der Aalst, K. M. van Hee, and G.J. Houben. Modelling Workflow Management Systems with High-Level Petri Nets. In *Proceedings of the Second Workshop on Computer-Supported Cooperative Work, Petri Nets and Related Formalisms*, October 1994.
11. 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.
12. 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.
13. 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.