

When does a Workflow Complete?

Parvathi Chundi¹, Tai Xin², and Indrakshi Ray²

¹ Computer Science Department
University of Nebraska at Omaha
Omaha, NE 68182-0500
pchundi@mail.unomaha.edu

² Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873
{xin, iray}@cs.colostate.edu

Abstract. Workflow Management Systems (WFMS) coordinate execution of logically related multiple tasks in an organization. Each workflow that is executed on such a system is an instance of some workflow schema. A workflow schema is defined using a set of tasks that are coordinated using dependencies. Workflows generated 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 of tasks 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. This, in turn, will cause the workflow to hold on to the resources and raise availability problems. 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. We provide an algorithm that can be used by application developers to generate the completion sets associated with a workflow schema. Generating all possible completion sets for a large workflow is computationally intensive. Towards this end, we show how to approximately estimate the number of completion sets. If this number exceeds some threshold specified by the user, then we do not generate all completion sets. In such cases, we provide two options. The first is to generating the first k completion set. However, generating the first k completion sets may not be very meaningful. We provide a second approach in which the users identify one or more vital nodes which must be contained in all completion sets.

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, very few researchers have addressed the issue of workflow completion. In almost all 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 a tedious and error-prone process. In this paper, we aim to automate this process.

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_i . The consequence is that this workflow will never complete and the resource availability of the system will be compromised.

Checking such things manually is impossible for real-world applications. This is because real-world workflow applications or web service transactions can be composed of hundreds of tasks and dependency relationships. For example, an online service transaction is composed of various tasks, such as, login, authentication, inventory browsing, searching, ordering, payment management, fulfillment, confirmation, calculation, rejection, reviewing, and credit checking. Each of these tasks may be performed at a different server or service holder, and the dependencies will enforce an the ordering or constraints between the tasks. Identifying possible paths and completion sets is very meaningful and crucial; some paths can lead to successful execution, some paths may be prone to scams and attacks, some paths must be fine-tuned to improve customer experience, and so on.

The numerous tasks and dependencies may generate an extremely large dependency graph with a huge number of completion sets. However, not all of the completion sets are meaningful to the service provider. The service provider is interested in only some particular paths ending with specific sink nodes, such as, order confirmed or payment rejected. We can identify the paths with these vital nodes, and generate completion sets that involve such nodes. The proposed algorithms provide a mechanism to analyze the

dependency graph, find the completion set and help service provider/workflow planner to build and optimize workflow execution.

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.

Generating all completion sets for a large workflow having many nodes and dependencies may be computationally intensive. We provide an algorithm that estimates the number of potential completion sets for a given workflow. If this number exceeds a given threshold that is set by the application developer, then all the completion sets are not generated. We provide a simple approach that generates the first k completion sets of a workflow. An alternate approach is to designate certain nodes as vital to the workflow and generate completion sets that involve these designated nodes.

The research results obtained is also applicable to web services transactions [3, 17]. A web services transaction specification describes an extensible coordination framework for coordinating tasks. The web services transaction is often a long duration activity and the tasks are executed asynchronously by different parties. Dependencies can be used to coordinate the different tasks of a web services transaction. Our analysis techniques can be applied to the web services transactions to ensure that it is indeed possible to complete such a transaction. This will be a stepping towards standardization of web services.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 defines our workflow processing model and describes the different kinds of dependencies that may be associated with it. Section 4 describes the impacts of dependencies on building completion sets. Section 5 presents the details of the algorithm and shows how to generate all the possible completion sets correctly. Section 7 concludes the paper with pointers to future directions.

2 Related Work

In the past two decades, a variety of transaction models and technologies supporting workflow and other advanced transactions have been proposed. Examples include ACTA [9], ConTracts [19], nested transactions [14], ASSET [7], EJB and CORBA object transaction services [16], workflow management systems [2], concurrency control in advanced databases [6] etc. Chrysanthis and Ramamrithan introduce ACTA [9], 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 formally specifying and reasoning about advanced transaction models.

Workflows are composite activities typically involving a variety of computational and business activities. The importance of workflow models is increasing rapidly due to its suitability for business applications. For these reasons, a lot of research appears in workflow management systems [2, 4, 13, 18].

Reliable scheduling and failure recovery for workflows have been discussed by Ray, Xin and Zhu in papers [18, 23, 25]. Damage assessment and repair for workflows from malicious attacks are discussed in paper [26]. In these papers, the authors consider the constraints of different control-flow dependencies and discuss how to ensure satisfaction of dependencies in execution and recovery.

A large body of work appears in the formal analysis of workflows and advanced transactions [1, 5, 8, 10, 12, 15, 21, 22]. Davulcu et al. [10] provide a framework for specifying, analyzing, and executing workflows based on Concurrent Transaction Logic. Using this logic, the authors describe how to verify whether a workflow is consistent and can be scheduled. Other properties can be verified using this approach. Mukherjee et al. [15] discuss the relative merits and demerits of analyzing workflows using temporal logic, event algebra, and Concurrent Transactional Logic. The Concurrent Transaction Logic is the most suitable for expressing workflows because it can handle generalized constraints and can represent control flow graphs with transition conditions on the arcs. Bonner [8] investigated the expressive power of Concurrent Transaction Logic and proposed some theoretical results in this regard. Vortex [11] workflows are not based on logic. However, abstractions can be derived from Vortex workflows. This abstraction can be represented in temporal logic which can be automatically verified using model checkers. Singh has discussed the semantical inter-task dependencies on workflows [21]. The author used algebra format to express the dependencies and analyze their properties and semantics in workflow systems. Attie et al. [5] 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 [20] 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, such as, 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.

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 Petri Nets and demonstrate its use for the 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 Petri Nets to represent control-flow, value and temporal dependencies; they also demonstrate how to use Petri Nets to conduct analysis

on workflow structures – like inconsistent dependency specifications, terminable with an acceptable state, feasible to complete execution with the given temporal constraints. This work is very useful as it demonstrates Petri Nets as an effective tool for modeling and analysis of workflows. However, to the best of our knowledge, none of these work address issues pertaining to the complete execution of workflow.

In an earlier work [24], we discussed about the completion of workflows. Specifically, we identified the impact dependencies have on the completion sets and proposed algorithms for automatically computing completion sets. This paper augments the previous mentioned work by providing a complexity analysis of the algorithm for computing completion sets. We also provide an algorithm for estimating the number of completion sets. If the number of completion sets is large, we give two approaches for producing a subset of the completion sets. The first is by generating the first k completion sets. The second approach generates completion sets that pass through one or more vital nodes.

3 Our Model for Workflows

3.1 Workflow

We begin by presenting the concept of workflow schema and discuss the kinds of control-flow dependencies that can be supported. A workflow schema is specified by the set of tasks, the dependencies between these tasks, and the completion sets. A *workflow* is an instance of some workflow schema.

All tasks specified in a workflow schema may not execute or commit. A completion set gives the set of tasks that needs to be committed for completing an instance of the workflow. Some of these tasks must be committed in a specific order – the completion set also needs to specify this ordering relation. Moreover, the set of tasks that commit may vary with different instances of execution. Thus, a workflow schema may have multiple completion sets.

Definition 1. [Workflow Schema]: A workflow schema W_w is a triple $\langle S, D, C \rangle$, 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 .

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. The commit and abort events are referred to as termination events.

We use the notation W_w to denote a workflow, and the notation T_{wi} or T_{wj} to denote tasks of the workflow W_w .

Definition 3. [Completion Set] Consider the workflow $W_w = \langle S, D, C \rangle$. The completion component C defines the set of completion sets in W_w . 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_{wi} .

3.2 Dependencies

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 set of control-flow dependencies has been defined in the work of ACTA [9]. A comprehensive list of transaction dependency definitions can be found in [4, 7, 9]. Summarizing all these dependencies in the previous works, 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 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, $b_{ik} \Rightarrow (a_{ij} \prec b_{ik})$.

Note that, the constraints placed by different dependencies might conflict with each other. In this paper, we assume that the workflow specifications are free of such conflicts. For the algorithms to detect and remove dependency conflict, please refer to an earlier paper [23].

A workflow W_w can be represented in the form of a graph $G_w = \langle V, E \rangle$ which we term as the *dependency graph*. The task $T_{w1}, T_{w2}, \dots, T_{wn}$ 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.

The following example shows how a real-world workflow making travel arrangements can be represented by our model.

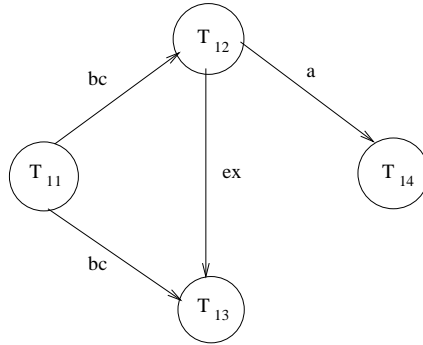


Fig. 1. Dependencies in the Workflow of Example 1

Example 1. Let $W_1 = \langle S, D, C \rangle$ 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.

4 Impacts of Dependencies on Completion Sets

Recall that a workflow $W_w = \langle S, D, C \rangle$ specifies the set of tasks S , the dependencies between these tasks D , and the set C that contains the completion sets of W_w . A completion set of a workflow specifies the tasks that need to be committed and the order in which they must be committed for successful execution of some instance of the workflow.

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 – $T_{wj} \in CT_t \Rightarrow T_{wi} \notin CT_t$. A complete list of the control-flow dependencies and their impacts on deciding a completion set CT_t is given in Table 1.

Dependency	Includes
$T_{wi} \rightarrow_{sc} T_{wj}$	$T_{wi} \in CT_t \Rightarrow T_{wj} \in CT_t$
$T_{wi} \rightarrow_a T_{wj}$	$T_{wj} \in CT_t \Rightarrow T_{wi} \in CT_t \wedge T_{wj} \prec T_{wi}$
$T_{wi} \rightarrow_{wa} T_{wj}$	-
$T_{wi} \rightarrow_{ex} T_{wj}$	$T_{wi} \in CT_t \Rightarrow T_{wj} \notin CT_t$
$T_{wi} \rightarrow_{fca} T_{wj}$	$T_{wi} \in CT_t \Rightarrow T_{wj} \in CT_t \wedge T_{wj} \prec T_{wi}$
$T_{wi} \rightarrow_{fbc} T_{wj}$	-
$T_{wi} \rightarrow_{fbb} T_{wj}$	-
$T_{wi} \rightarrow_{fba} T_{wj}$	-
$T_{wi} \rightarrow_{fbi} T_{wj}$	-
$T_{wi} \rightarrow_c T_{wj}$	$T_{wi} \in CT_t \wedge T_{wj} \in CT_t \Rightarrow T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_t T_{wj}$	$T_{wi} \in CT_t \wedge T_{wj} \in CT_t \Rightarrow T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_b T_{wj}$	-
$T_{wi} \rightarrow_s T_{wj}$	$T_{wi} \in CT_t \wedge T_{wj} \in CT_t \Rightarrow T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_{bc} T_{wj}$	$T_{wj} \in CT_t \Rightarrow T_{wi} \in CT_t \wedge T_{wi} \prec T_{wj}$
$T_{wi} \rightarrow_{ba} T_{wj}$	$T_{wj} \in CT_t \Rightarrow T_{wi} \notin CT_t$

Table 1. Impacts of Dependencies on Deciding Completion Sets

The completion sets specified by a user in a workflow may not conform to the given dependencies. Based on the analysis in Table 1, we provide an algorithm that checks whether the completion set complies with the dependencies in an workflow.

Algorithm 1

Check Whether Completion Set Conforms to Dependency

Input: Specification of workflow $T = \langle S, D, C \rangle$

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
    if  $(T_{wi} \rightarrow_{bc} T_{wj} \in \mathbf{D}) \wedge (T_{wj} \in CT_m) \wedge (T_{wi} \notin CT_m)$ 
      return false
    if  $(T_{wi} \rightarrow_{fca} T_{wj} \in \mathbf{D}) \wedge (T_{wi} \in CT_m) \wedge (T_{wj} \not\ll_m T_{wi})$ 
      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. The force-commit-on-abort dependency $T_{wi} \rightarrow_{fca} T_{wj}$ has the similar effects as a reversed begin-on-commit dependency $T_{wj} \rightarrow_{bc} T_{wi}$ [23], so we also examine this dependency in the algorithm. If no improper specification is detected, the algorithm returns true indicating that the completion set conforms with the dependencies.

5 Generating Completion Sets Automatically

The algorithm for generating completion sets automatically will use the graph representing the workflow which will henceforth be referred to as *workflow DAG*. The dependency graph will include both the directly given dependencies and the implicit dependencies. Implicit dependencies [23] 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.

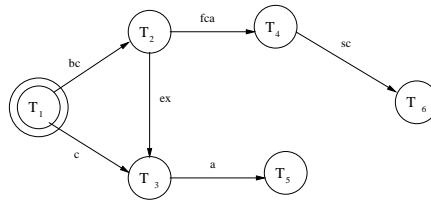


Fig. 2. An Example of Workflow and its Dependency Graph

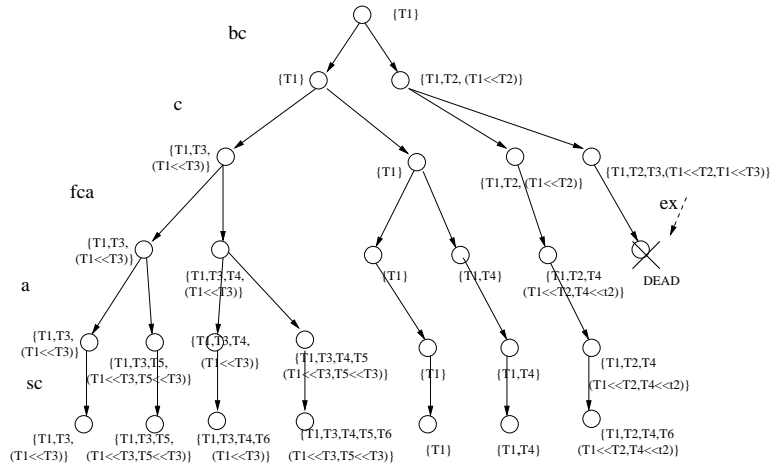


Fig. 3. Generate Completion Sets for Workflow in Figure 2

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.

(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 2

Input: the workflow specification $AT_t = \langle S, D, C \rangle$

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*

$DDS = DDS + e_{new}$

end

end

```

        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 3

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
                     $CT_s = CT_t \cup \{T_{wj}\}$ 

```

```

        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

```

We first provide an argument that Algorithms 2 and 3 generate a complete set of completion sets for the given workflow DAG T . Algorithm 2 first generates all dependencies between each pair of tasks in T . The dependency information is then used to traverse the workflow DAG in a breadth-first manner to generate all completion sets. Let l be the length of the longest path in T . Then, any completion set generated by the algorithm contains at most l tasks. We describe a proof by induction on the length k of a completion set, to show that algorithms 2 and 3 work correctly. For the basis case, let $k = 1$. This happens when a completion set consists of a single task, which is the start node. The above algorithms generate all completion sets containing only the start node since the start node is the first to be inserted in the completion set queue in Algorithm 3 to generate completion sets. For the induction case, let us assume that the algorithm generate all completion sets of size j correctly. Now, we prove that the algorithms generate completion sets of length $k + 1$ correctly. For every pair of nodes in T , Algorithm 2 applies every dependency type systematically to generate all derived and non-derived dependencies between the nodes. Algorithm 3 considers each size k completion set, applies each of the dependency edges computed in Algorithm 2 to extend the set to $k + 1$ size if possible. Since k -size completion set is considered with each dependency edge, it can be easily observed that all $k + 1$ -size completion sets are generated. This completes the proof.

We now discuss how to estimate the complexity of Algorithm 3. Given a workflow DAG, it is easy to see that a possible completion set typically contains all vertices on a path from a source vertex to a sink vertex. If there are ex or ba dependencies between

tasks, then that path may not lead to a legal completion set. However, the number of paths in the given workflow DAG between the source and any sink provides us an upper bound on the number of completion sets. The following theorem shows that there may be exponential number of paths (or possible completion sets) in a workflow DAG.

Theorem 1. *The number of possible completion sets in a workflow DAG consisting of n vertices is $O(2^n)$.*

Proof. Each path p can be mapped to a bit vector of size n . If p contains the i^{th} vertex, the i^{th} bit in the vector is set to 1. Otherwise, it is set to 0. Since there are 2^n bit vectors of size n , the number of potential completion sets is $O(2^n)$ in the worst case.

Since the number of potential completion sets are exponential, it is going to require exponential time to generate all of them. Therefore, Algorithm 3 runs in time $O(2^n)$.

6 Incorporating Task Importance into Completion Set Computation

The algorithms in previous sections compute completion sets where all tasks have the same importance. However, there may be cases some tasks are more important than others. For example, while planning a vacation, hotel reservation may be more important than renting a car. Therefore, one may wish to compute possible completion sets that include one or more tasks specified by the user. In this section, we show how to identify all completion sets with a given task. The same algorithm can be easily extended to compute completion sets that contain a subset of the user specified tasks.

Let X be a user specified node in a workflow DAG. To compute all possible completion sets that include X , we use the following steps. In Step 1, we compute the list of tasks that follow X in some completion set. In Step 2, all nodes that precede X in some completion set are computed. In Step 3, a subgraph of the input workflow DAG is computed that contains only those nodes that are identified in Steps 1 and 2. Algorithm 3 is then used on the subgraph to generate all completion sets.

Algorithm 4

Generate all Completion Sets that include Node X

Input: (i) Workflow DAG $G = (V, E)$ where V denotes the set of vertices and E denotes the set of edges. The DAG is stored in the form of an adjacency list *adjList*. (ii) Node X – The node which must be contained in all completion sets.

Output: The data structure *pathList*(X) gives the potential completion sets that contain node X

Procedure ComputeVitalPath

begin

A_X = set of nodes reachable from X by depth-first traversal³ of G starting at Node X

E_A = set of edges G traversed while computing A_X .

G_T = Transpose of G .

B_X = set of nodes reachable from X by depth-first traversal of G_T starting at Node X .

E_B = set of edges in G traversed while computing B_X .
Merge A_X and B_X to create the set of tasks T_X that appear with X in some completion set.
Merge E_A and E_B to create E_X .
for $i = 1$ to $|V|$ **do**
 if the i^{th} node Y is not in T_X **then**
 remove Y and its adjacency list from G ;
 remove all edges incident on Y from edge set of G ;
 foreach edge e in edge set of G **do**
 if $e \notin E_X$, delete edge from the edge set of G .
 Use Generate Completion Set algorithm on G to compute all completion sets that contain X .
end

The algorithm uses two depth-first traversal of the DAG G to extract the subgraph of G containing X . It first traverses G in a depth-first manner starting at X at node X to first compute all nodes that are reachable from X in G . To compute all nodes from which X can be reached, we use the depth-first traversal of transpose of G starting at X . A union of the nodes and edges traversed in these two traversals will generate the subgraph of G that contains only those nodes that reach and are reachable from X . Algorithms 2 and 3 are then applied to this subgraph to generate all completion sets that include the given node X .

Algorithm 4 takes time $O(|V| + |E|)$ to perform both the depth-first traversals and $O(|V||E|)$ time to compute the subgraph that contains only nodes from A_X and B_X . Algorithm 3 takes exponential time to compute all completion sets from this subgraph. Therefore, Algorithm 4 takes exponential time to compute all completion sets containing the given node X .

Example 2. Consider a complex workflow consisting of the following 11 tasks which are described below. (i) T_{w1} – login to the travel arrangement application, (ii) T_{w2} – input the travel information including traveller’s name, source, and destination cities, (iii) T_{w3} – input information needed for air travel, (iv) T_{w4} – display all flight options for the user to select, (v) T_{w5} – purchase the airline ticket, (vi) T_{w6} – input information needed for car travel, (vii) T_{w7} – reserve a rental car, (viii) T_{w8} – reserve the hotel room, (ix) T_{w9} – indicate incomplete transaction if flight or rental car reservation failed, (x) T_{w10} – print the travel details, and (xi) T_{w11} – logout of the travel application. The DAG for this workflow appears in Figure 4. Suppose T_{w4} is considered a vital node. The depth-first traversal of the workflow DAG in Figure 4 starting at node T_{w4} results in $A_X = \{ T_{w4}, T_{w5}, T_{w8}, T_{w9}, T_{w10}, T_{w11} \}$. All edges that are in the adjacency list of all these nodes are added to E_A . The transpose G_T of the workflow DAG is shown in Figure 5. Set B_X is computed from G_T as $\{ T_{w1}, T_{w2}, T_{w3}, T_{w4} \}$. E_B contains edges $T_{w3} \leftarrow T_{w4}$, $T_{w2} \leftarrow T_{w3}$, and $T_{w1} \leftarrow T_{w2}$.

After merging A_X and B_X , nodes T_{w6} and T_{w7} and their edges are removed from G . In addition, edge $T_{w1} \leftarrow T_{w11}$ is removed. The resulting workflow DAG is shown Figure 6. Algorithm 3 outputs the following completion sets: $\{ T_{w1}, T_{w2}, T_{w3}, T_{w4}, T_{w5}, T_{w9}, T_{w11} \}$ and $\{ T_{w1}, T_{w2}, T_{w3}, T_{w4}, T_{w5}, T_{w8}, T_{w10}, T_{w11} \}$.

In some workflows, there may be more than one vital node. For example, in addition to T_{w4} , the user may decide to include T_{w8} (transaction to reserve a hotel room) as an

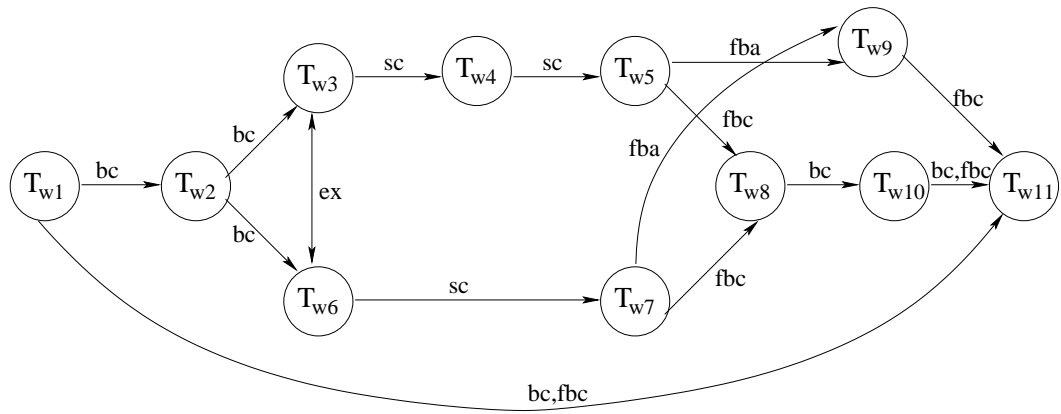


Fig. 4. An Example of a Complex Workflow and its Dependency Graph

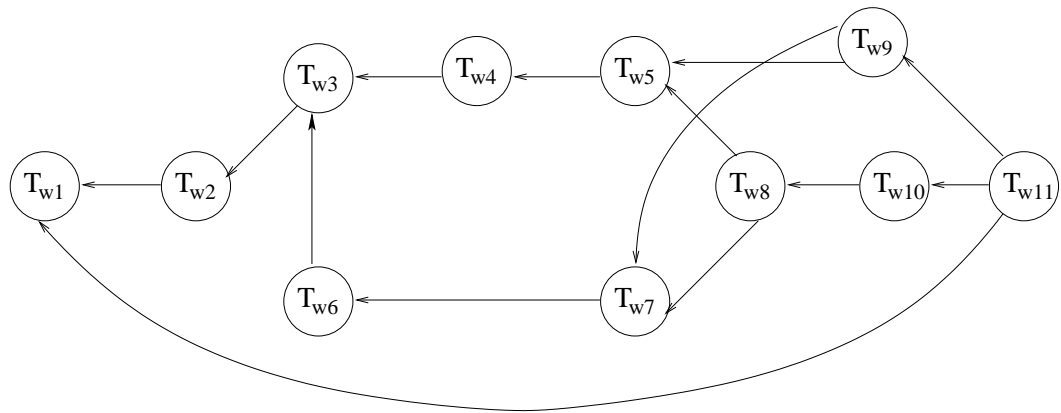


Fig. 5. The Transpose of the Workflow DAG in Figure 4

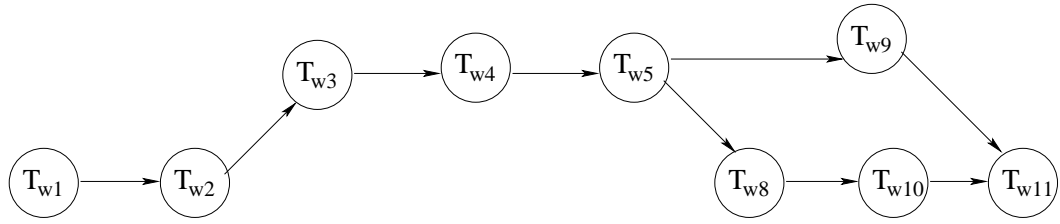


Fig. 6. The Subgraph for Vital Node T_{w4} .

additional vital node. In order to compute all completion sets that contain one or more of the specified vital nodes, we extend Algorithm 4 in the following manner. For each vital node X , compute T_X and E_X . Union T_X and E_X for all vital nodes to create T_{all} and E_{all} . Remove from G nodes that do not appear in merged T_{all} . Remove from G edges that do not appear in E_{all} . Compute the completion sets on the remaining graph.

Figure 7 shows the graph containing the nodes and edges remaining in G when T_{w4} and T_{w8} are both vital nodes. The completion sets generated are: $\{ T_{w1}, T_{w2}, T_{w3}, T_{w4}, T_{w5}, T_{w9}, T_{w11} \}$, $\{ T_{w1}, T_{w2}, T_{w3}, T_{w4}, T_{w5}, T_{w8}, T_{w10}, T_{w11} \}$, and $\{ T_{w1}, T_{w2}, T_{w6}, T_{w7}, T_{w8}, T_{w10}, T_{w11} \}$.

7 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. For large complex workflows, generating all possible completion sets may be computationally intensive. Towards this end, we showed how to estimate the number of potential completion sets. If the number of potential completion sets is large, we provide two alternatives. The

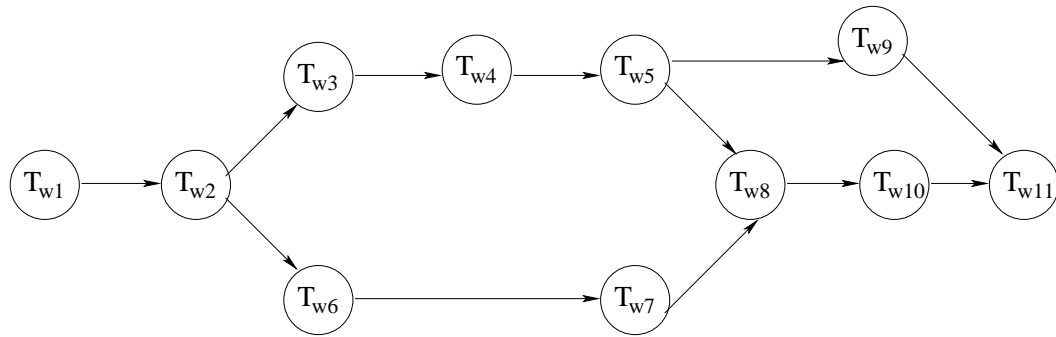


Fig. 7. The Subgraph for Vital Nodes T_{w4} and T_{w8}

first alternative is to generate a subset of the completion sets. The second alternative is to identify one or more vital nodes in the workflow and generate completion sets that involve these node.

In real-world applications, all users are not given permission in a workflow. Authorization constraints are specified over the tasks in a workflow. In future we plan to look at authorization constraints specified on a workflow. These constraints specify which user has the permission to execute the individual tasks in a workflow. The constraints may be specified such that no user has the appropriate authorization and the workflow remains incomplete. In future, we plan to investigate how authorization constraints impact workflow execution.

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. G. Alonso, H. Kuno, F. Casati, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2003.
4. 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.

5. 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.
6. N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.
7. 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.
8. A.J. Bonner. Workflows, Transactions and Datalog. In *Proceedings of the 18th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, May 1999.
9. P. Chrysanthis. ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis, September 1991.
10. H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based Modeling and Analysis of Workflows. In *Proceedings of the 17th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seattle, Washington, June 1998.
11. X. Fu, T. Bultan, R. Hull, and J. Su. Verification of Vortex Workflows. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Genova, Italy, April 2001.
12. R. Gunthor. Extended Transaction Processing based on Dependency Rules. In *Proceedings of the 3rd International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 207–214, Vienna, Austria, April 1993.
13. D. Hollingsworth. Workflow Reference Model. Technical report, Workflow Management Coalition, Brussels, Belgium, 1994.
14. J. E. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. PhD Thesis 260, MIT, Cambridge, MA, April 1981.
15. S. Mukherjee, H. Davulcu, M. Kifer, P. Senku l, and G. Yang. Logic-Based Approaches to Workflow Modeling and Verification. In J. Chomicki and R. v. d. Meyden and G. Saake, editor, *Logics for Emerging Applications of Databases*. Springer, 2004.
16. OMG. Additional Structuring Mechanisms for the OTS Specification. OMG, Document ORBOS, 2000-04-02, September 2000.
17. M. P. Papazoglou. Web Services and Business Transactions. *World Wide Web Journal*, 6(1):49–91, March 2003.
18. 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.
19. 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.
20. M. Rusinkiewicz and A. P. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems*, pages 592–620, 1995.
21. 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.
22. W. M. P. van der Aalst. The Application of Petri-Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 1(8), 1998.
23. 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.

24. T. Xin, I. Ray, P. Chundi, and S. Chaichana. On the Completion of Workflows. In *Proceedings of the Seventeenth International Conference on Database and Expert Systems*, Krakow, Poland, September 2006.
25. 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.
26. 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.