A Chinese Wall Security Model for Decentralized Workflow Systems †

Vijayalakshmi Atluri
MSIS Department and CIMIC
Rutgers University
Newark, NJ 07102, USA
atluri@cimic.rutgers.edu

Soon Ae Chun
MSIS Department and CIMIC
Rutgers University
Newark, NJ 07102, USA
soon@cimic.rutgers.edu

Pietro Mazzoleni*
Dipartimento Di Informatica
Universita Di Milano
Milan, Italy
mazzolen@cimic.rutgers.edu

ABSTRACT

Workflow systems are gaining importance as an infrastructure for automating inter-organizational interactions, such as those in Electronic Commerce. Execution of inter-organizational workflows may raise a number of security issues including those related to *conflict-of-interest* among competing organizations. Moreover, in such an environment, a centralized Workflow Management System is not desirable because: (i) it can be a performance bottleneck, and (ii) the systems are inherently distributed, heterogeneous and autonomous in nature. In this paper, we propose an approach to realize decentralized workflow execution, in which the workflow is divided into partitions called *self-describing workflows,* and handled by a light weight workflow management component, called *workflow stub,* located at each organizational agent. We argue that placing the task execution agents that belong to the same conflict-of-interest class in one self-describing workflow may lead to unfair, and in some cases, undesirable results, akin to being on the wrong side of the *Chinese wall.* We propose a Chinese wall security model for the decentralized workflow environment to resolve such problems, and a restrictive partitioning solution to enforce the proposed model.

Keywords

Decentralized Workflow Execution, Chinese Wall Security Policy, Self-describing workflow

1. INTRODUCTION

Since timely services are critical for any business, there is a great need to automate or re-engineer business processes. Many organizations achieve this by executing the coordinated activities (tasks) that constitute the business process (workflow) through workflow management systems (WFMS). In general, a workflow can be defined as a set of tasks and dependencies that control the coordination requirements among these tasks. The task dependencies can be categorized into control-flow dependencies, value dependencies, and external dependencies [9, 14].

With the rapid growth of internet usage for enterprise-wide and cross-enterprise business applications (such as those in Electronic Commerce), workflow systems are gaining importance as an infrastructure for automating inter-organizational interactions.

Traditionally, the workflow management and scheduling is carried out by a single centralized workflow management engine. This engine is responsible for enacting task execution, monitoring workflow state, and guaranteeing task dependencies. However, in an electronic commerce environment with inter-organizational workflows, a centralized Workflow Management System is not desirable because: (i) scalability is one of the pressing needs since many concurrent workflows or instances of the same workflows are executed simultaneously, and a centralized WFMS can cause a performance bottleneck, and (ii) the systems are inherently distributed, heterogeneous and autonomous in nature, and therefore do not lend themselves to centralized control. In fact, several researchers have recognized the need for decentralized control [2, 18, 10, 6].

In this paper, we propose *a decentralized workflow management model* (DWFMS) where the intertask dependencies are enforced without having to have a centralized WFMS. Our model introduces the notion of *self-describing workflows* and *WFMS stubs.* Self-describing workflows are partitions of a workflow that carry sufficient information so that they can be managed by a local task execution agent rather than the central WFMS. A WFMS stub is a light-weight component that can be attached to a task execution agent, which is responsible for receiving the self-describing workflow, modifying it and re-sending it to the next task execution agent.

Execution of inter-organizational workflows may raise a number of security issues including those related to *conflict-of-interest* among competing organizations. In this paper, we demonstrate that placing the task execution agents that belong to the same conflict-of-interest group in one self-describing workflow may lead to unfair, and in some cases, undesirable results, akin to being the wrong side of the *Chi-*

†This work is supported in part by the National Science Foundation under grant EIA-9983468.

*The work of P. Mazzoleni was conducted while visiting CIMIC, Rutgers University during 2000.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'01 November 5-8, 2001, Philadelphia, Pennsylvania, USA.
Copyright 2001 ACM 1-58113-385-5/01/0011 ...$5.00.

nese wall. We propose a Chinese wall security model for the decentralized workflow environment to resolve such problems, and a restrictive partitioning solution to enforce the proposed model.

The remainder of the paper is organized as follows. In section 2, we present the motivation to this paper with an example by distinguishing the centralized control with its decentralized counterpart. In section 3, we present our workflow model. In section 4, we present our approach to providing decentralized control. In appendix A, we provide a brief review of the Chinese wall security policy. In section 5, we present a variation of the Chinese wall security model suitable for decentralized workflow systems, called the *DW Chinese Wall Policy*, which can be used to eliminate the problems that arise due to conflicts-of-interest. In section 6 we present our approach to decentralized control that enforces the DW Chinese wall policy. Section 7 provides a brief review of related research. Finally, section 8 provides conclusions and future research directions. Due to space limitations, we have not included the proof of the theorems (see [3]).

2. MOTIVATION

A workflow is comprised of a set of tasks, and a set of task dependencies that control the coordination among the tasks. In an inter-organizational workflow, tasks are executed by different, autonomous, distributed systems. We call the system that executes a specific task a *task execution agent*, or simply an *agent*. We denote the agent of a task t_i as $A(t_i)$. In the following, we will take an example to illustrate first how the workflow is executed with centralized control and decentralized control, and then portray the security problems that arise due to decentralized control.

EXAMPLE 1. Consider a business travel planning process that makes reservations for a flight seat, a hotel room and a rental car. The workflow that depicts the process at a travel agent may comprise of the following tasks:
t_1: Input travel information
t_2: Reserve a ticket with Continental Airlines
t_3: if t_2 fails or if the ticket costs more than \$400, reserve a ticket with Delta Airlines
t_4: if the ticket at Continental costs less than \$400, or if the reservation at Delta fails, purchase the ticket at Continental
t_5: if Delta has a ticket, then purchase it at Delta.
t_6: Reserve a room at Sheraton, if there is flight reservation, and
t_7: Rent a car at Hertz
The corresponding workflow can be depicted as a graph, shown in figure 1. Note that "bs" and "bf" in the figure stand for "begin on success" and "begin on failure," respectively. Assume that each task is executed at the appropriate agent, for example, t_2 by Continental, t_3 by Delta, etc. In the above workflow example, the type of dependencies that are of interest to us in this paper are $t_2 \rightarrow t_3$ and $t_2 \rightarrow t_4$, which state that t_3 should begin only if t_2 is not successful or the outcome of t_2 is more than \$400, and t_4 starts when t_2's outcome is \$400 or less. Examples such as this are not unusual (consider priceline.com), where a customer sets a maximum he is willing to pay, but not necessarily looking for the best price. At the same time, he may have preferences for the merchants whom he wants to do business with, for example preferences for a specific set of airlines in a certain order to accrue frequent flyer miles. To keep the example simple, we have not taken into account the case where both t_2 and t_3 result in a failure, but we realize that the example can be enhanced to take into account all the cases.

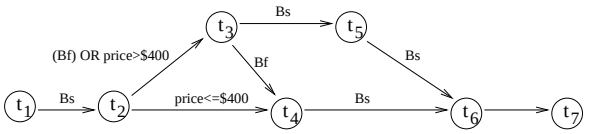

Figure 1: A Travel Plan Workflow

Centralized Control

With centralized control, there exists a single WFMS that is responsible primarily for: (1) distributing the tasks to the appropriate agents, and (2) ensuring the specified task dependencies by sending the tasks to their respective agents only when all requisite conditions are satisfied. To achieve this, the WFMS first sends t_1 to $A(t_1)$, after it receives the response from $A(t_1)$, sends t_2 to $A(t_2)$. When it receives the response from $A(t_2)$, it evaluates the dependencies to choose the next task in the workflow according to the dependency, and sends it to the corresponding agent. For instance, if the result from $A(t_2)$ was price > \$400, the WFMS would send t_3 to $A(t_3)$. After receiving the response, it sends t_4 or t_5 to its corresponding agent, $A(t_4)$ or $A(t_5)$, and so on. In other words, the WFMS is responsible for the control flow at every stage of execution, as shown in Figure 2.

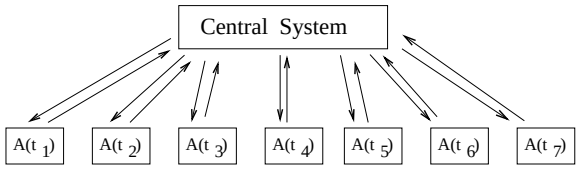

Figure 2: Centralized (Traditional) Workflow Management

Decentralized Control

With decentralized control, the entire workflow is sent to $A(t_1)$ by the central WFMS. After the execution of t_1, $A(t_1)$ forwards the remaining workflow to the following agents, in this case, $A(t_2)$. After executing t_2, $A(t_2)$ evaluates the following dependencies and forwards the remaining workflow to the next appropriate agent. For instance, if the price > \$400, $A(t_2)$ would send the remaining workflow $(t_3, t_4, t_5, t_6, t_7)$ to $A(t_3)$. Alternatively, if the price $\leq$ \$400, it would send the remaining workflow (t_4, t_6, t_7) to $A(t_4)$. $A(t_3)$ executes its task t_3, evaluates the dependency, and makes a choice to send the remaining workflow to the appropriate agent, either to $A(t_4)$ or to $A(t_5)$, and so on. At the end, the last task execution agent(s) need to report the results back to the central WFMS, as shown in figure 3. Note that this way of execution results in fewer message exchanges between the central WFMS and the task execution agents, and also minimizes the control by one single central controlling authority, which is desirable in autonomous environments.

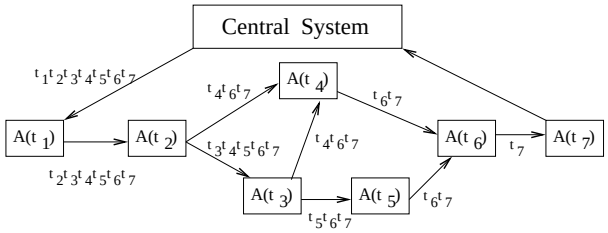

Figure 3: Decentralized Workflow Management

Security Problems due to Decentralized Control

There is a clear problem, if we employ decentralized control to execute the workflow in example 1. After the execution of t_2, $A(t_2)$ must send the remaining workflow to either $A(t_3)$ or $A(t_4)$, based on the outcome of t_2. $A(t_2)$ has the knowledge that if it fails (that is, no ticket is available) or if the ticket costs more than $400, the task needs to be sent to $A(t_3)$ (Delta airlines), which is a competing company. Due to this fact, $A(t_2)$ can manipulate the price of the ticket and may reduce it to $399, which may result in a loss of business to $A(t_3)$ or may prevent the customer from getting a better price than $399 that may potentially be offered by Delta. Note that $A(t_2)$ cannot gain such an advantage if the workflow were executed with centralized control. This is because the central WFMS first sends t_2 to $A(t_2)$ and observes the result, and if it is more than $400 sends t_3 over to $A(t_3)$. Since $A(t_2)$ has no knowledge of the conditional dependency, it outputs its originally intended price. It is important to note that the actions of $A(t_2)$ are legitimate, and do not involve any malicious activity such as changing the control logic of the workflow. The problem still persists even if the dependency information is revealed only to $A(t_3)$. Similar problem exists with $t_2 \longrightarrow t_4$. Thus, with the knowledge of dependency information, one agent can benefit at the cost of the other. It is important to note that revealing only t_2 to $A(t_2)$ by appropriately encrypting the workflow will not work. This is because $A(t_2)$ has to evaluate the dependency and forward the remaining workflow, and therefore $A(t_2)$ should know both the dependency and the following agent. In this paper, we argue that the problem depicted above is similar to that of that addressed by the Chinese Wall Security Policy, and we propose a modified Chinese wall security policy to suit to the decentralized control environment.

3. THE WORKFLOW MODEL

A workflow is a set of tasks with task dependencies defined among them. Formally:

DEFINITION 1. **[Workflow]** A workflow W can be defined as a directed graph (T, D), where T, the set of nodes, denotes the tasks $t_1, t_2 ... t_n$ in W, and D, the set of edges, denotes the intertask dependencies $t_i \xrightarrow{x} t_j$, such that $t_i, t_j \in T$ and x the type of the dependency.

Given a workflow, $W = \langle T, D \rangle$, we define the tasks and the dependencies in the following.

3.1 Workflow Tasks

The task structure can be represented as a state transition diagram with a set of states and a set of primitive operations, as shown in figure 4. At any given time, a task t_i can be in one of the following states (st_i): initial (in_i), executing (ex_i), done (dn_i), committed (cm_i), aborted(ab_i), succeeded (su_i) or failed (fl_i). A primitive moves the task

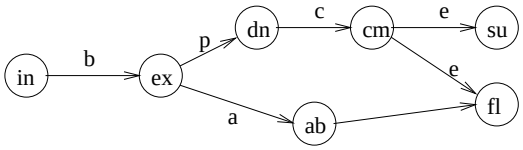

Figure 4: States of a Task

from one state to another. Given a task t_i, we assume the following primitives (pr_i): *begin (b_i), precommit (p_i), commit (c_i), abort (a_i) and evaluate (e_i).* We denote the set of these distinct states and primitive operations by ST and PR, respectively.

Note that, failure of a task can be due to one of the following two reasons: (1) a task cannot execute to its completion due to an internal failure (such as abort), or (2) its output is not as expected although the execution has successfully completed. The latter can be due to an invalid input from the user. For example, the task of reserving a ticket for a flight may commit successfully, but there may not be any seats available. So a successful commit may still result in a failure of a task.

DEFINITION 2. **[Task]** Each task $t_i \in T$ is a 4-tuple $\langle A, C, Input, Output \rangle$, where A denotes the execution agent of t_i, C the set of activities (or operations) within t_i, $Input$ the set of input parameters to t_i, and $Output$ the set of output parameters from t_i.

In the following, we use the notation $A(t_i), C(t_i), Input(t_i)$, and $Output(t_i)$ to denote the task agent, the set of activities, the set of input parameters, and the set of output parameters of t_i, respectively.

EXAMPLE 2. An example of a task, t_1 = *Purchase a ticket at Continental*, is as follows:
$A(t_1) = ContinentalTravelAgent$,
$C(t_1) = \{check\_seat, check\_price, make\_invoice\}$,
$Input(t_1) = \{travel\_date, destination\}$, and
$Output(t_1) = \{invoice\_number, ticket\}$,
which states that t_1 requires a travel date and destination as its input and generates the invoice number and a ticket as its output.

3.2 Workflow Dependencies

Intertask dependencies support a variety of workflow coordination requirements. Basic types of task dependencies include *control-flow dependencies, value-dependencies* and *external dependencies* [1, 13].

1. *Control flow dependencies*: Also referred to as *state dependencies*, these dependencies specify the flow of control based on the state of a task. Formally, a control-flow dependency specifies that *a task t_j invokes a primitive pr_j only if t_i enters state st_i.* For example, a begin-on-success dependency between tasks t_i and t_j denoted as $t_i \xrightarrow{bs} t_j$, states that t_j can begin only if t_i enters a succeeded state.

2. *Value dependencies:* These dependencies specify the flow of control based on the outcome of a task. Formally, *a task t_j can invoke a primitive pr_j only if a task t_i's outcome satisfies a condition c_i.* For example, $t_i \xrightarrow{bs, x>100} t_j$ states that t_j can begin only if t_i has successfully completed and the

value of its outcome, x is > 100. Since the outcome can be evaluated only in case of a successful completion of a task, all value dependencies have to be associated with a "bs" dependency. Therefore, explicit representation can be omitted.

3. *External dependencies:* These dependencies specify the control flow based on certain conditions satisfied on parameters external to the workflow. *A task t_i can invoke a primitive pr_i only if a certain condition c is satisfied where the parameters in c are external to the workflow.* For example, a task t_i can start its execution only at 9:00 am, or a task t_i can start execution only 24 hrs after the completion of task t_k.

Each task t_i, therefore is associated with a set of state dependency variables $\mathcal{S} = ST$, value dependency variables $\mathcal{V} = Output(t_i)$, and external variables $\mathcal{E}$.

DEFINITION 3. **[Dependency Variables and Literals]** A dependency variable dv for a task t_i is defined as follows: If $t_i \in T$ and $v \in DV = \{\mathcal{S} \cup \mathcal{E} \cup \mathcal{V}\}$, then $dv = t_i.v$.
A dependency literal l is a value that a dependency variable can take, and is defined as $l \in L = \{R \cup N \cup G \cup ST\}$, where R is the set of real numbers, N the set of natural numbers, G the set of alphanumeric strings, and ST the set of all possible states for tasks in W.

DEFINITION 4. **[Dependency Expression]**
A dependency expression, de is defined as follows:

- if $dv \in DV$ and $l \in L$, and $op \in \{=, \neq, <, >, \leq, \geq\}$, then $dv\ op\ l$ is a dependency expression.
- if de_1 is a dependency expression, then (de_1) is a dependency expression;
- if de_1 is a dependency expression, then $\neg de_1$ is a dependency expression; and
- if de_1 and de_2 are dependency expressions, $(de_1 \wedge de_2)$ and $(de_1 \vee de_2)$ are dependency expressions.

EXAMPLE 3. Following are examples of dependency expressions.

1. $t_1.state = success$;
2. $(t_1.price > \$400 \wedge t_2.seat \geq 2)$

DEFINITION 5. **[Dependency]** Each dependency $t_i \xrightarrow{d} t_j$ in D, is a 4-tuple $\langle hd, de, tl, pr\rangle$, where hd and tl denote the head (t_i) and tail (t_j) tasks, de the dependency expression, and $pr \in PR$ the primitive of t_j to be invoked when de is true.

EXAMPLE 4. Following is a list of examples of the three types of dependencies:

1. $t_1 \xrightarrow{bc} t_2$: $\langle t1, t1.state = commit, t2, begin\rangle$
2. $t_1 \xrightarrow{bc, price>\$200} t_2$: $\langle t1, (t1.state = commit \wedge t1.price > \$200), t2, begin\rangle$
3. $t_1 \xrightarrow{time=10am, abort} t_2$: $\langle t1, (t1.time = 10am), t2, abort\rangle$

4. OUR APPROACH TO DECENTRALIZED CONTROL

In this section, we will first propose a methodology and architecture to enforce the inter-organizational task dependencies without the need for having to have a centralized

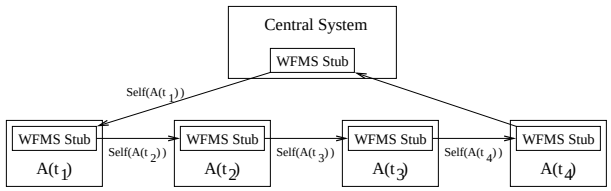

Figure 5: Our Approach to Decentralized Control

WFMS. In regard to this, we propose (1) *self-describing workflows* and (2) *WFMS stubs.* Self-describing workflow carries workflow information, and WFMS stubs are lightweight software component that can be installed at each agency to process a self-describing workflow. In the following, we discuss them in detail.

4.1 Self-describing workflows

Intuitively, a self-describing workflow, comprises of (1) a task t, (2) all the tasks that follow t and the dependencies among them, (3) the agent that executes t, (4) the input objects required to execute t. This information is piggy-backed along with t when sending it to its execution agent. Figure 5 shows how such decentralized control can be achieved using the notion of self-describing workflows.

If we walk through this example, the central WFMS stub constructs a self-describing workflow with the entire workflow, and sends it to $A(t_1)$ first. WFMS Stub at $A(t_1)$ executes t_1, partitions the remainder of the workflow if needed, constructs a self-describing workflow(s), and sends it to the subsequent agent $A(t_2)$ based on the dependency evaluation. That is, as the workflow execution progresses, it gets divided into partitions and forwarded for the next task execution agent. We assume the initial partition is the entire workflow, which is denoted as P_1. Let P_i be the ith partition. Following is a formal definition of the self-describing workflow:

DEFINITION 6. **[Self Describing Workflow]** Given a workflow P_i, we define its self-describing workflow, SELF(P_i), as a tuple $\langle t_i, PreSet(t_i), OutState(t_i), P_i\rangle$, where t_i is the first task in P_i, $PreSet(t_i)$ is preconditions to be satisfied before t_i entering a state st_i, $OutState(t_i)$ is the set of dependency variables in $t_i \xrightarrow{d} t_j$ for all t_j with an outgoing dependency from t_i and their values generated from t_i's execution.

$OutState(t_i)$ can be a control state, value state of a variable, and/or an external state. For instance, OutState(t_1) of dependency 2 in example 4 can be $\{cm, price > \$200\}$, and for that of dependency 3 can be {bs, time=9AM}. Note that $OutState(t_i)$ is used for evaluating dependency expressions, while $Output(t_i)$ is forwarded to the following agents to be used as input to their tasks.

4.2 WFMS stub

A WFMS stub is a small component that can be attached to a task execution agent. This module is responsible for interpreting the given workflow: i.e. (1) evaluate preconditions and execute its task, (2) partitions the remaining workflow, constructs self-describing workflows, (3) evaluate

control information and (4) forwards each to its subsequent agent.

4.2.1 Precondition Evaluation

The WFMS stub at each agent needs to evaluate the preconditions for its task to change its state from one to another through its primitive operations. Although, until now we have referred to the precondition set of a task t_i ($PreSet(t_i)$) that applies to the task as a whole, since the task dependencies may specify invocation of any primitive, we need to distinguish them for each primitive. Before the task changes its state from one to another, preconditions attached to each primitive operation need to be evaluated. Following is the definition of preconditions of a task t_i for each primitive operation pr_i.

DEFINITION 7. [**Precondition Set**] Given a task t_i in W, we define the *precondition set*, PreSet(t_i) = $Pre^b_{t_i} \cup Pre^c_{t_i} \cup Pre^a_{t_i}$, such that each $Pre^{pr}_{t_i} = \{de_1 \vee de_2 \vee ... \vee de_n | t_k \xrightarrow{d} t_i$ where $k = 1..n$ $\}$ where $d = \langle t_k, de, t_i, pr \rangle$.

For the sake of simplicity, in this paper, we assume that all joins and splits are OR-joins and OR-splits. In the above definition, $Pre^b_{t_i}$, $Pre^c_{t_i}$ and $Pre^a_{t_i}$ are dependency expressions that must be satisfied to invoke primitive operations, begin, commit, and abort, respectively. We have not included the preconditions for the evaluate primitive because, typically there will not be any dependency specification that requires to invoke it.

For example, $PreSet(t_3)$ in our example 1 comprise of: $Pre^b_{t_3} = (t2.state = fl \vee t2.price > \$400)$, $Pre^c_{t_3} = \emptyset$ and $Pre^a_{t_3} = \emptyset$.

Given PreSet(t_i), we say $Pre^{pr}_{t_i}$ is satisfied at a given state if the de in $Pre^{pr}_{t_i}$ is evaluated true in $OutState(t_k)$.[1]

4.2.2 Workflow Partitioning

Once the task is completed, the WFMS stub prepares self-describing workflows for the following task agents, by first partitioning the remaining workflow. Following is an algorithm to partition and then generate a self-describing workflow.

ALGORITHM 1. [**Self-describing Workflow Construction**]

Partitioning: Given W_i at A(t_i),
For each t_j where $t_i \xrightarrow{x} t_j$ exists,
p = a connected component $< t_j, ... >$ where t_j is its root
P_j = {(T,D)| T = a set of tasks $\{t_j, t_k, ..., t_e\}$ in p and
D = a set of dependencies among tasks in T}
Generation of Self-describing Workflows:
Given a partition P_j,
t_j = first task in Pj
SELF(P_j) = $\langle t_j, PreSet(t_j), OutStates(t_j), P_j \rangle$

We illustrate the working of the partitioning with the help of an example shown in figure 6. After task agent $A(t_2)$ finishes its task execution with $P2$ in figure 6(A), it partitions the remaining workflow into two, $P3$ and $P4$ as shown in figure 6(B). Notice the t_6 and t_7 are included in both $P3$ and $P4$ since we assume there is OR-join at t_6, so they are not omitted in case one partition results in a failure during execution. Subsequently, $A(t_4)$ would partition $P4$ into $P6$, and at $A(t_6)$ $P6$ gets partitioned into P7 as in figure 6(C). The workflow partitioning is a necessary step for decentralized execution. To handle COI, we modify this partitioning algorithm by further restricting the way in which partitioning is done (in algorithm 2 in section 6).

[1]If $Pre^{pr}_{t_i} = \emptyset$, then the preconditions are always satisfied for that primitive operation.

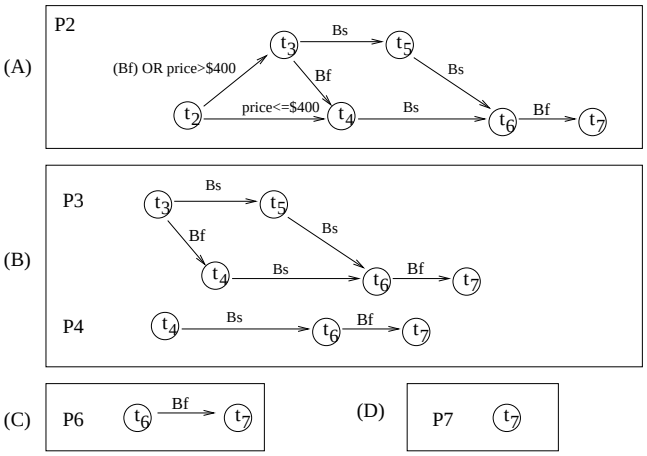

Figure 6: An example to illustrate partitioning

4.2.3 The WFMS stub

In the following, we describe the functionality of the WFMS stub at each A(t_i). The WFMS stub encounters the following three cases. Since the algorithm is similar to the secure WFMS stub algorithm (algorithm 3), we provide only an informal discussion here.

Case 1: WFMS stub does not need to evaluate the preconditions of its following task(s) t_j to send SELF(P_j). This is possible if $Pre^b_{t_j} = \emptyset$. In this case, the task(s) following t_i can be executed in parallel with that of t_i. Therefore, the WFMS stub does not need to evaluate the precondition of t_j at this point, but t_i and t_j can start their execution in parallel. Therefore, the WFMS stub at $A(t_i)$ first constructs the SELF(P_j), sends it over to $A(t_j)$, and executes its own task t_i. Only after the execution of t_i is complete, it evaluates $PreSet(t_j)$ and sends a signal to $A(t_j)$ indicating the completion of its execution. For example, consider the dependency $t_2 \xrightarrow{c} t_3$ in the workflow shown in figure 7. Since this is a commit dependency, only the precondition for the commit primitive is non-empty, but that of the begin primitive is empty. In this case t_2 and t_3 will be executed in parallel.

Case 2: WFMS stub needs to evaluate the preconditions of its following task(s) t_j before sending the SELF(P_j) to $A(t_j)$. This is required if $Pre^b_{t_j} \neq \emptyset$. Therefore, WFMS stub first constructs SELF(P_j), executes t_i, evaluates $Pre^b_{t_j}$, and sends SELF(P_j) to $A(t_j)$ only if the $Pre^b_{t_j}$ is true. For example, in $t_1 \xrightarrow{bs} t_2$ in figure 7, SELF(P_2) will not be sent to $A(t_2)$ if $Pre^b_{t_2}$ is not true.

Case 3: The WFMS stub needs to evaluate the precondition of its own task t_i. When a task is executed in parallel along

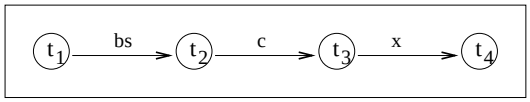

Figure 7: An example to describe WFMS stub

with its preceding task as in case 1, it has to wait (in the done state) for the signal to terminate its execution. This work will be in addition to taking care of one of the previous two cases. For example, the WFMS stub at $A(t_3)$ has to evaluate $Pre^c_{t_j}$ before commiting. In addition, it has to do the functions of either case 1 or case 2 based on $t_3 \xrightarrow{x} t_4$.

5. CHINESE WALL SECURITY MODEL FOR DECENTRALIZED WORKFLOWS

In this section, we propose a variation of the Chinese Wall Security model that is capable of addressing the problems of conflict-of-interest in the decentralized workflow environment. We call this as *DW Chinese Wall Security Policy*. Note that the conventional Chinese Wall security model addresses the issue of a company information leaking to another company, which is in the same conflict of interest group. In our model, the sensitive information is not inherent to a company (task agent), but rather the workflow definition itself. First, the sensitive dependency information could leak to a task agent, thus the output of a task can be manipulated. Second, the output or "response" of a task can be sensitive and can leak to another task. In the following, we will first define objects, subjects, read and write operations by drawing an analogy with those of the original Chinese wall security model. Based on these definitions, we then define our security model.

Objects: Objects include, the *data objects* in a company data set, and the dependencies in a workflow. We refer the latter as *dependency objects*. We categorize the data objects in the workflow into two: *sensitive* and *non-sensitive*. Sensitive data objects are those that change the execution flow of the workflow. In other words, these are the objects involved in the dependencies. We are concerned about only sensitive objects, so from now on, data objects refer to only sensitive objects.

- Object COI Property: A distinguishing property that we impose here is that, if x is a data object involved in a dependency object o, then o belongs to the company of x as well as to all companies in the same COI class.

Let us consider dependency $t_2 \xrightarrow{bf \vee price > 400} t_3$ in example 1 to illustrate sensitive and non-sensitive objects. Since the execution flow depends on the value of the price written by $A(t_2)$, *price* is a sensitive object. On the other hand, consider the workflow which simply gathers the price from each airline and reports them back to the user where the user decides the airline of his choice. In such a case, although price is one of the output parameters of t_2, since there is no dependency defined over *price*, it cannot influence the execution flow. Therefore, in this case, price is a non-sensitive object. There are two sensitive dependency objects in the workflow in example 1, which is $price > \$400$ and $price \leq \$400$.

Subjects: A subject is the task execution agent that executes a task in a workflow. A subject S, by definition belongs to one company and therefore belongs to that COI class, which is denoted as $COI(S)$ (unlike the conventional Chinese wall policy in which a subject's association with a company is determined by his first access.) A subject is allowed to access (both read and write) any data object from its company dataset.

Read and Write operations: A read operation includes reading data and dependency objects, and evaluating the dependency expressions. A write operation includes writing to data objects, and generating self-describing workflows.

DEFINITION 8. [**DW Chinese Wall Security Policy**]

1. [**Evaluation/Read Rule**]: A subject S can read an object O,
   if O belongs to its own company data set,
   or if O is a dependency object that does not belong to another company in COI(S).

2. [**Write Rule**]: A subject S can write an object O,
   if S can read O

The read rule says that a subject is allowed to read its own company data objects. A subject is also allowed to read any dependency object that does not belong to a company which is in the same COI as that of the subject's company. Considering once again example 1, the dependency $t_2 \xrightarrow{(bf) \vee (price > 400)} t_3$ belongs to both $A(t_2)$ and $A(t_3)$, according to our object COI property. Hence both $A(t_2)$ and $A(t_3)$ are not allowed to read this, thus not allowed to evaluate it. The write rule says that a subject is allowed to write any object, both data and the dependency type, if it is allowed to read. Note that writing an object includes partitioning the workflow to generate self-describing workflows.

6. DECENTRALIZED CONTROL WITH THE DW CHINESE WALL POLICY

In this section, we provide the partitioning and WFMS stub algorithms that enforce the DW Chinese wall security policy.

6.1 Restrictive Partitioning

According to the read and write rules of this policy, a task execution agent cannot read, evaluate preconditions, or write to any sensitive object that belongs to a different company within its own COI class. In other words, it not only is not allowed to view, but also not allowed to construct a self-describing workflow that involves sensitive objects or tasks that belong to the same COI class. To accomplish this, we restrict the partitions using the following rule.

DEFINITION 9. [**Restrictive Partitions Rule**] Let t_i be a task in W. Let P_i be a partition sent to $A(t_i)$. A restrictive partition P_j of t_i is such that there exist no sensitive objects belonging to $COI(A(t_i))$.

DEFINITION 10. [**Critical Partition**] A partition P_i is said to be a *critical partition* if it does not comply with the restrictive partition rule.

A trivial solution to obtain restrictive partitions is as follows. When a workflow is submitted to the central WFMS, it partitions the entire workflow so that no two tasks that belong to the same COI class exist in a partition. However, the central WFMS stub has to ensure the execution of each partition as a separate workflow. Although this solution is simple and straight forward, this requires relying heavily on the central WFMS for the execution of the entire workflow. In the worst case, it effectively results in centralized control.

The challenge, therefore, is to decompose a workflow into partitions that satisfy the restrictive partition rule (1) without having to introduce any additional centralized control, and (2) without making the intermediate task execution agents take over part of the control.

In the following, we propose a restrictive partition algorithm that meets these two challenges. We divide $Pre_{t_i}^{pr}$ as $Pre_{t_i}^{local}$ and $Pre_{t_i}^{remote}$, where $Pre_{t_i}^{local}$ comprises of the predicate involving non-sensitive objects, and $Pre_{t_i}^{remote}$ comprises of the predicate involving sensitive objects.

ALGORITHM 2. **[Restrictive Partition]**

Given P_i,
P_j = Partition(P_i)
for each critical partition P_j
Given t_m such that $t_j \xrightarrow{d} t_m$ in P_j
Case 1: $t_j \longrightarrow t_m$ is sensitive for both $A(t_j)$ and $A(t_m)$
/*COI($A(t_j)$)=COI($A(t_m)$) and d belongs to both */
add a dummy task t_i^d at $A(t_i)$ such that C(t_i)=$\emptyset$
split P_j as follows:
P_j : $t_j \xrightarrow{signal} t_i^d$
P_i : remove the dependency $(t_j \xrightarrow{d} t_m)$,
add dependency $t_i^d \xrightarrow{d'} t_m$ such that $d' = d$
$Pre_{t_j}^{pr}$=$Pre_{t_j}^{local}$ and $Pre_{t_i}^{pr}$=$Pre_{t_j}^{remote}$
Output(t_j) = Outstate(t_j)∪Output(t_j)
Case 2: $t_j \longrightarrow t_m$ is sensitive only for $A(t_j)$
/*$A(t_m)$ evaluates the sensitive part of the dependency*/
split P_j as follows:
P_j : $t_j \xrightarrow{signal} t_m$
P_m : remove $t_j \longrightarrow t_m$ from P_j
$Pre_{t_j}^{pr}$=$Pre_{t_j}^{local}$ and $Pre_{t_m}^{pr}$=$Pre_{t_j}^{remote}$
Output(t_j) = Outstate(t_j)∪Output(t_j)
Case 3: d is not sensitive to either $A(t_j)$ or $A(t_m)$
split P_j into
P_j : $t_j \xrightarrow{signal} t_m$
P_m : remove $t_j \longrightarrow t_m$ from P_j
$Pre_{t_j}^{pr}$=$Pre_{t_j}^{remote}$ and $Pre_{t_m}^{pr}$=$Pre_{t_j}^{local}$
Output(t_j) = Outstate(t_j)∪Output(t_j)

In the following, we explain the three different cases of restrictive partitioning: Case 1: when the sensitive information pertains to both task agents ($A(t_j)$ and $A(t_m)$), the sensitive dependency information is split and routed through a neutral task agent ($A(t_i)$), thus avoiding the information leakage to both $A(t_j)$ and $A(t_m)$; Case 2: when the sensitive information is only to $A(t_j)$, then the sensitive part of the dependency will be evaluated at $A(t_m)$, leaving $A(t_j)$ with non-sensitive information, thus preventing $A(t_j)$ from manipulation; Case 3: when the sensitive information pertains to $A(t_m)$ only or somewhere along the path there exists sensitive information to $A(t_j)$, then the algorithm splits the rest of the workflow into two to prevent $A(t_j)$ to get hold of senstive information, or try to hide sensitive information from $A(t_m)$ by splitting the dependency between $A(t_j)$ and $A(t_m)$.

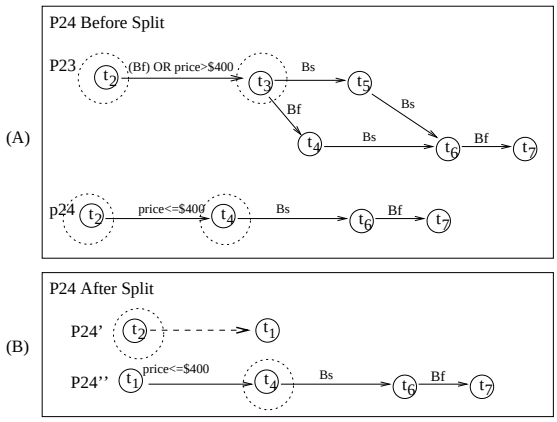

Figure 8: An example for restrictive partition for adjacent tasks

Note that when the above algorithm makes a restrictive partition, it removes the actual dependency connecting these two restrictive partitions and adds an additional *signal* dependency that still connects these two partitions. A signal dependency simply sends a signal. The reason for adding such a dependency is to ensure that the dependency information is not lost even when it is removed. In other words, it preserves the semantics of the workflow while partitioning.

EXAMPLE 5. We illustrate case 1 of the restrictive partition algorithm using the travel plan workflow shown in figure 1. Assume P_1 be the workflow shown in figure 1 sent to $A(t_1)$. When t_1 is finished the WFMS stub at $A(t_1)$ has to split the remaining workflow for the next task agent $A(t_2)$. The partition P_2 is critical because the dependency $t_2 \xrightarrow{bf \vee price > \$400} t_3$ and $t_2 \xrightarrow{price <= \$400} t_4$ belong to $A(t_2)$ who cannot read them (due to the read rule). Since there exist two outgoing edges from t_2, the critical partition P_2 needs to consider both sub-partitions, P_{23} and P_{24} as shown in figure 8(a). We have used dashed circle to indicate the agents that belong to the same COI class.

To show how the restrictive partition algorithm works, we will consider only the sub-partition P_{24} since the solution for the partition P_{23} is almost the same. This is case 1 of the algorithm 2 since the dependency object belongs to both COI($A(t_m)$) and COI($A(t_j)$) and COI($A(t_m)$) = COI($A(t_j)$).

(1) The first step is to remove the dependency $t_2 \xrightarrow{price < \$400} t_4$, and split P_{24} into two: $P'_{24} = \{t_2\}$ and $P''_{24} = \{t_4 \xrightarrow{bs} t_6, t_6 \xrightarrow{bf} t_7\}$.

(2) The second step is to split dependency into sensitive (remote) and non-sensitive parts (local), and hide the sensitive dependency information. We introduce a dummy task t_1^d at $A(t_1)$, whose function is only to keep the ouput from t_2 and use it to evaluate the dependency between t_2 and t_4. In this case the dependency comprises of all sensitive predicates, and therefore, it has to be evaluated at $A(t_1)$ (all the dependency is $Pre_{t_1^d}^{remote}$).

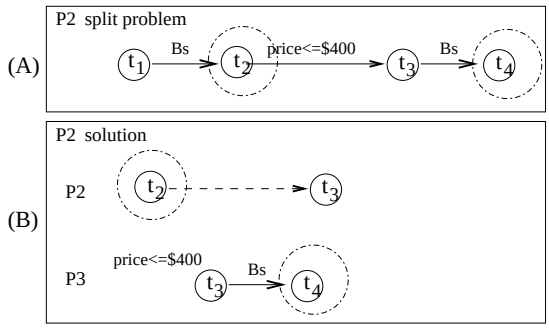

Figure 9: An example for restrictive partition for non-adjacent tasks

(3) The last step is to add a signal dependency between t_2 and t_1^d so that $P'_{24} = \{t_2 \overset{signal}{\longrightarrow} t_1^d\}$, as shown in figure 8(B). The purpose of the signal dependency is simply that when $A(t_2)$ completes the execution it will generate a self-describing workflow to inform $A(t_1)$ that it can start evaluating the sensitive dependency with Output and Outstate forwarded by the stub in $A(t_2)$. We have used a dashed edge to represent the signal dependency.

In this manner, $A(t_2)$ neither has the knowledge of the remainder of the workflow (i.e. t_4, t_6, t_7), nor does it know on what sensitive control flow logic t_1 would proceed. We preserve the control flow logic of the original $t_2 \overset{price<\$400}{\longrightarrow} t_4$ by including this dependency as a Precondition in $A(t_1^d)$.

For step 2, if the dependency contains both control and sensitive value dependency (as in $t_2 \overset{bf \vee price>\$400}{\longrightarrow} t_3$ in P_{23}), we have to split the dependency expression into two: *local* and *remote* dependency expressions. Local dependency expressions (i.e. the control flow bf) should be evaluated after the execution of original task (in that case t_2), while remote dependency expressions (i.e. sensitive dependency like the value dependency $price > \$400$) should be evaluated in the dummy task (t_1^d), so that reading the sensitive information by the same COI class, $A(t_2)$ as well as $A(t_3)$, is prohibited. $A(t_1)$ contains enough information to evaluate the precondition $price < \$400$, and if the condition evaluates to true, it splits the partition P''_{24} as (t_4, t_6, t_7), and sends the workflow to the next agent like usual.

Example 6. This example illustrates the senario of case 2. In the workflow shown on figure 9(A) we suppose that the tasks t_2 and the task t_4 are in the same COI; We also suppose that the dependency $t_2 \overset{price<=\$400}{\longrightarrow} t_3$ is sensitive for $A(t_2)$ (and of course for $A(t_4)$).

When t_1 has to split the workflow to send to $A(t_2)$ it has to take care of the fact that $A(t_2)$ cannot read the dependency $t_2 \overset{price<=\$400}{\longrightarrow} t_3$. In this case, different from the one in example 5, $A(t_3)$ can read the dependency and can evaluate that.

To do that, $A(t_1)$ has to split the workflow into two parts. $P2$, which includes t_2 and a signal dependency from t_2 to t_3. $P3$ includes the task t_3 and all the tasks following that. To evaluate the sensitive dependency, we have to split it into two. $Pre_{t_2}^{local}$ including the non-sensitive objects that can be evaluated in $A(t_2)$ $Pre_{t_2}^{remote}$ including the sensitive objects that have to be evaluate in the task t_3 using the Outstate(t_2) that are also sent to $A(t_3)$ for permitting the evaluation.

In this case, there are only sensitive objects that have to be evaluated in $A(t_3)$. This does not reveal sensitive objects to $A(t_4)$ because the conditions have already been evaluated before the workflow is sent to it.

Theorem 1. The restrictive partition algorithm (algorithm 2) enforces the read and write rules of the DW Chinese wall policy.

6.2 Secure WFMS stub

In order to make the WFMS stub at the central system same as that in the agents, we introduce the following. We assume there exist two dummy tasks t_0 and t_F that comprise of empty set of operations such that PreSet(t_0) = PreSet(t_F) = ϕ, $Input(t_0) = Output(t_0)$ and $Input(t_F) = Output(t_F)$. The central WFMS is the task execution agent responsible for executing both t_0 and t_F.

Definition 11. Given a task t_i in a self-describing workflow SELF(P), we say t_i is tail(SELF(P)) if there exists no t_j such that $t_i \rightarrow t_j$.

For example, in figure 6(B), t_7 = tail(SELF($P4$)) and t_7 = tail(SELF($P6$)).

When a workflow W is sent to the central WFMS, it includes a t_0 to the initial task and a dependency $t_0 \overset{bs}{\longrightarrow} t_1$ such that t_1 is one of the tasks in W that do not have any dependency $t_i \rightarrow t_1$ in W. Then the central WFMS sends the workflow to the WFMS stub in its own location. The functions of the WFMS stub (at the central as well as at the execution agents) are outlined in the following algorithm. This algorithm employs the DW Chinese Wall policy using the restrictive partitioning.

Algorithm 3. **[Secure WFMS stub at A(t_i)]**

Given a SELF(t_i)
extract the task t_i to be executed in A(t_i)
P_j = Restrictive Partition(P_i)
construct SELF(P_j)
if ($Pre_{t_j}^b = \emptyset$)
/*The task t_j has to be executed in parallel with t_i*/
  then forward SELF(P_j) to A(t_j)
execute(t_i) until $state(t_i) = done$
if ($Pre_{t_i}^b = \emptyset$)
/* t_i is executed in parallel with the preceding task and it has to wait to complete*/
  then wait
    until ($sync - signal \neq yes \vee \neg timeout$)
  raise an error **if** (timeout)
if ($Pre_{t_i}^{pr}$ = true) where $pr \in \{commit, abort\}$
  then terminate execution
  else raise an error
if (OutState(t_i) satisfies dependency expression de in PreSet(t_j) $\wedge$ Pre_{t_j} = true)
  then
    if ($Pre_{t_j}^b = \emptyset$)
      then send sync-signal to A(t_j)
        about the completion of its execution
      else forward SELF(P_j) to A(t_j)

The above WFMS stub algorithm uses restrictive partition algorithm to split the workflow for task agents that are

in conflict-of-interest. It also shows how a task (t_i) is executed sequentially or in parallel with the subsequent task (t_j). In case of parallel execution, the *sync-signal* needs to be sent between task agents in order to coordinate the task executions. For instance, the dependency specification in $t_i \xrightarrow{c} t_j$ allows to begin parallel execution of t_i and t_j. However, once t_j is in *done* state, it needs to wait for a synchronization signal from t_i in order to *commit*.

Once the self-describing workflows are generated, it should be proven that the composition of partitioned self-describing workflows do not lose the dependencies originally in the self-describing workflows. The following defines the equivalence of a SELF(P) and the composition of its partitioned workflows. Theorem 1 proves that when the partitioned self-describing workflows are assembled together, they are equivalent to the original SELF(P).

DEFINITION 12. **[Equivalence]** Given two self-describing workflows SELF(P) and SELF(P'), we say that SELF(P) is *equivalent* to SELF(P'), denoted as SELF(P) $\equiv$ SELF(P') if, (1) the set of all operations in SELF(P) is same as that of SELF(P'), and (2) for each t_i, the PreSet(t_i) in SELF(P) = PreSet(t_i) in SELF(P').

THEOREM 2. Let SELF(P_i) be a self-describing workflow. Assume SELF(P_i) is decomposed into SELF(P_{i_1}), SELF(P_{i_2}), ... SELF(P_{i_n}) using algorithm 2. Then SELF(P_i) $\equiv \cup_{j=1}^{n}$SELF(P_{i_j}).

7. RELATED WORK

In recent years, several approaches and architectures for decentralized workflow execution have been proposed [2, 18, 10, 6]. Our approach to decentralized control differs from these in the following aspects. In these approaches, a workflow is pre-partitioned in a central server, and the partitions are made statically in the central server and distributed to each execution agent, whereas our approach partitions as the workflow progresses with its execution. Therefore, our approach can accommodate dynamism easily. None of these approaches address the conflict-of-interest issues while partitioning. Some workflow security issues have been addressed in [4, 1], but are geared towards access control.

Work in the area of mobile code security where code is executed by untrusted hosts is also relevant to our work [7, 17, 8, 16]. The security concern here is to ensure that sensitive information in the "floating" software is not exploited by malicious hosts for its advantage, and vice versa where the mobile software does not leak the sensitive information of the host to somebody else. Proposed solutions use cryptography where only the relevant code that needs to be executed is visible to the host. The code is typically encrypted using an "Onion Ring" approach, which is explained briefly below.

Suppose the code comprises of three parts, P1, P2, and P3 to be executed by three hosts H1, H2, and H3, respectively. P3 is first encrypted with H3's public key, this encrypted P3 and P2 together are encrypted by H2's public key, and so on. With this, H1 can access only P1, but cannot access neither P2 nor P3. Such solutions cannot resolve the COI issues resulting due to decentralized control since each host should know the control/dependency information in order to enforce it. If the dependency information d in $t_i \xrightarrow{d} t_j$ is encrypted with t_j's public key, t_i would not be able to know on which condition it should forward the remaining workflow to $A(t_j)$. If dependency d information is encrypted with $A(t_i)$'s public key, then the COI problem we addressed in this paper still remains since t_i has already read d.

[11, 12] propose a *decentralized label model* for more fine-grained information sharing, while reducing the potential information leakage through uncontrolled propagation, among distrusted applications as in mobile codes. Information at a host is labeled with a pair $\langle owner, readerlist \rangle$ where owner can specify the allowed flow of information, i.e. allowed readers in a program. However, it does not address the policies on the conflicts of interest among hosts. Its primary concern is to deal with privacy and confidentiality of information by controlling information propagation.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have first proposed a model for decentralized control of workflows, using the notions of *self- describing workflows* and *workflow stubs*. We have shown that fair execution of workflows in a decentralized workflow management system needs to take into consideration the Chinese wall policy and the conflicts of interest (COI) groups of task agents. We have proposed a Chinese Wall Security policy for decentralized control, in which we modify the original Chinese wall policy. We have then proposed algorithms to enforce these read and write rules using *restrictive partitions*.

While, the partition algorithm generates a non-restrictive self-describing partition if it does not contain sensitive objects, it generates restrictive self-describing partitions if it contains sensitive objects involving the same COI group. This approach allows to hide the sensitive information contained in dependencies so that the task agents cannot manipulate their output for their own advantage.

In our future work we intend to extend our framework to handle the AND join and split while partitioning. Note that although we have portrayed our approach as a solution to resolve the issue of conflicts of interest, one can adopt it under other considerations to restrict the partitions. For example, factors that affect the partitioning of workflows for distributed execution may include reliable network connections and geographic proximity among task agencies, heterogeneity of information systems used among task agencies, degree of autonomy for changing workflows by task agencies, and so on. We intend to explore these factors for restricting partitions in the future. We will also investigate how dynamic changes and exceptions of workflows can be modeled in decentralized WFMS for secure and fair execution of workflows.

9. REFERENCES

[1] Nabil R. Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and Analysis of Workflows using Petri nets. *Journal of Intelligent Information Systems, Special Issue on Workflow and Process Management,*, 10(2), March 1998.

[2] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Gunthor, and M. Kamath. EXotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proceedings of the IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations*, Trondheim, August 1995.

[3] Vijay Atluri, Soon Ae Chun, and Pietro Mazzoleni. A chinese wall security model for decentralized workflow systems. Cimic-technical report, MSIS Department, CIMIC-Rutgers University, November 2000.

[4] Elisa Bertino, Elena Ferrari, and Vijayalakshmi Atluri. A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems. In *Proc. of the 2nd ACM Workshop on Role-based Access Control*, November 1997.

[5] D.F.C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 206 –214, 1989.

[6] S. Das, K. Kochut, J. Miller, A. Sheth, and D. Worah. ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METEOR$_2$. Technical Report UGA-CS-TR-97-001, University of Georgia, February 1997.

[7] Richard Dean. *Formal Aspects of Mobile Code Security* . PhD thesis, Computer Science Department, Princeton University, 1999.

[8] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Issues and Requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, 1995.

[9] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, pages 119–153, 1995.

[10] P. Muth, D. Wodtke, and J. Weissenfels. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems*, 10(2), 1998.

[11] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, San Anonio, TX, January 1999.

[12] Andrew C. Myers. *Mostly-Static Decentralized Inforamtion Flow Control*. PhD thesis, Laboratory for Computer and Information Science, MIT, 1999.

[13] Marek Rusinkiewicz and Amit Sheth. Specification and Execution of Transactional Workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1994.

[14] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. *IEEE Computer*, 24(12):46–53, December 1991.

[15] Ravi S. Sandhu. A Lattice Interpretation of the Chinese Wall Policy. In *Proc. 15th NIST-NCSC Computer Security Conf.*, pages 329–339, Washington, D.C., October 1992.

[16] E. Vigna. *Mobile Agents and Security*. Springer, Berlin Heidelberg, 1998.

[17] Dan Seth Wallach. *A New Approach to Mobile Security*. PhD thesis, Computer Science Department, Princeton University, 1999.

[18] D. Wodtke and G. Weikum. A Formal Foundation For Distributed Workflow Execution Based on State Charts. In *Proc. International Conference on Database Theory*, Delphi, Greece, January 1997.

APPENDIX

A. THE CHINESE WALL SECURITY POLICY

The Chinese wall policy was identified by Brewer and Nash [5, 15] for information flow in a commercial sector. It is defined as follows. All company information is categorized into mutually disjoint conflict of interest classes, as shown in figure 10. For example, Banks, Oil Companies, Air Lines are the different conflict-of-interest (COI) classes. The Chinese wall policy states that information flows from one company to another that cause conflict of interest for individual consultants should be prevented. Thus, if a subject accesses Bank A information, it is not allowed to access any information within the same COI class, for example, that of Bank B. However, it can access information of another COI class, for example, oil company A.

The Brewer-Nash model [5] proposes the following mandatory read and write rules:

1. BN Read Rule: Subject S can read object O only if
   O is from the same company information as some object read by S,
   or O belongs to a COI class within which S has not read any object.
2. BN Write Rule: Subject S can write object O only if
   S can read O by the BN Read rule, and no object can be read which is in different company dataset to the one for which write access is requested.

The BN write rule prevents information leakage by Trojan Horses. For example, suppose John has read access to Bank A objects and Travel agency T objects, and Jane has read access to Bank B objects and Travel agency T objects. If John is allowed write access to T's objects, a Trojan Horse can transfer information from Bank A's objects to T's objects which is read by Jane, who then can read information about both Bank A and Bank B.

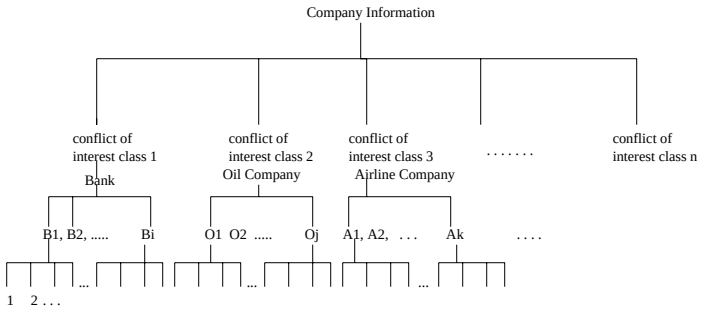

Figure 10: Company Information for the Chinese Wall Policy