Chinese Wall Security for Decentralized Workflow Management Systems *

Vijayalakshmi Atluri,[1] Soon Ae Chun[1] and Pietro Mazzoleni[2]
[1]MSIS Department and CIMIC, Rutgers University
180 University Avenue, Newark, NJ 07102
{atluri,soon}@cimic.rutgers.edu
[2]University of Milan
mazzoleni@dsi.unimi.it

Contact Author: Prof. Vijay Atluri, MSIS Department and CIMIC, 180 University Avenue, Rutgers University, Newark, NJ 07102, U.S.A.
Telephone: 973–353–1642, Fax: 973–353–5003, Email: atluri@cimic.rutgers.edu

Abstract

Workflow systems are gaining importance as an infrastructure for automating inter-organizational interactions, such as those in Electronic Commerce. 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. Decentralized 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 first provide 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. Second, we identify the limitations of the traditional workflow model with respect to expressing the various types of join dependencies and extend the traditional workflow model suitably. Distinguishing the different types of dependencies among tasks is essential in the efficient execution of self-describing workflows. Finally, we recognize 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.* Therefore, to address the conflict-of-interest issues that arise in competitive business environments, we propose a *decentralized workflow Chinese wall security model.* We propose a restrictive partitioning solution to enforce the proposed model.

Keywords: Workflow management systems, Distributed Systems, Chinese wall security

*This work was supported in part by the National Science Foundation under grant EIA-9983468. The work of P. Mazzoleni was conducted while he was visiting Rutgers during 2001-2002. A preliminary version of this work appeared in the *Proceedings of the ACM Conference on Computer and Communication Security, 2001.*

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), where a workflow is 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, 19].

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 such as *virtual enterprise* and *digital government* systems, in which the business processes often cross the organizational boundaries, and therefore do not lend themselves to centralized control. Although centralized systems lend themselves for better control and therefore are more suitable for ensuring the desired security, they cannot preserve the autonomy. Therefore, the real challenge is to ensure security without sacrificing autonomy. In fact, several researchers have recognized the need for decentralized control [2, 27, 13, 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.

Decentralized 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 *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.

Traditionally, join relations like AND-join or OR-join are logical relations among individual dependencies. We recognize that there arise many scenarios in which a join relation cannot be expressed simply by an AND-join or OR-join. We demonstrate, with a real-world example, that attempting to represent a complex join relation as an AND-join or OR-join may result in different semantics of the workflow dependencies. We have proposed an extension to the traditional workflow model that is capable of representing these join relationships. Synchronization of task execution in our proposed algorithms to generate self-describing workflows and restrictive partitions are highly influenced by these join relations.

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, followed by a brief overview of the proposed approach. In section 3, we review the Chinese wall security policy. In section 4, we present our workflow model including our extension of the model with join dependencies. In section 5, we present our approach to providing decentralized control. In section 6, we present a variation of the Chinese wall security model suitable for decentralized workflow systems, called the *DW Chinese Wall Policy*,

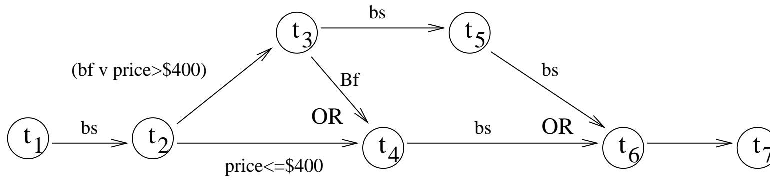

Figure 1: A Travel Plan Workflow

which can be used to eliminate the problems that arise due to conflict-of-interest. In section 7, we present our approach to decentralized control that enforces the DW Chinese wall policy. In section 8, we provide proof of correctness and security of our approach. Section 9 provides a brief review of related research. Finally, section 10 provides conclusions and future research directions.

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 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 consists 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, or when t_3 fails. 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.

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

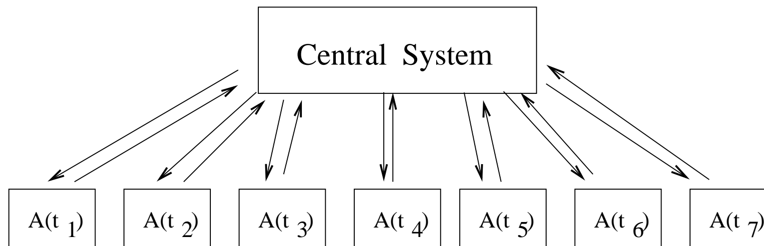

Figure 2: Centralized (Traditional) Workflow Management

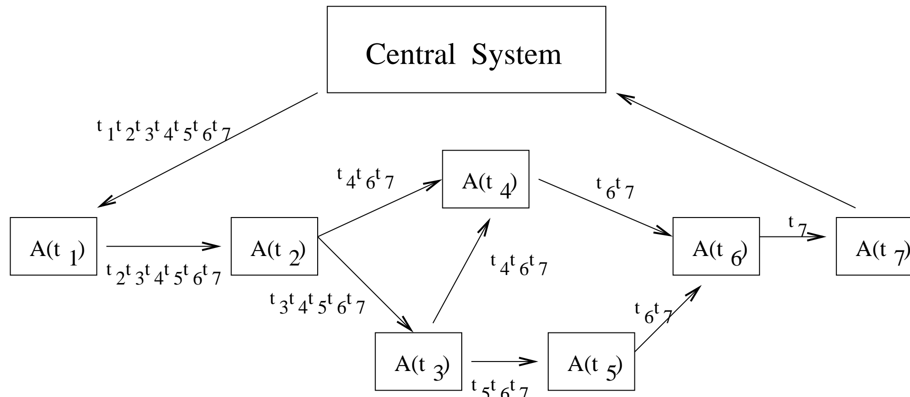

Figure 3: Decentralized Workflow Management

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.

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) needs 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.

Security Problems due to Decentralized Control

There is a clear problem, if we employ decentralized control to execute the above example. A task agent may learn (1) the output state (success or failure) of the previous task agent, or (2) the results of the previous task agent. This information can be potentially used by a task agent to control the execution of a subsequent task as well as the flow of execution of the entire workflow in such a way that it gains advantage over its competing task agents. We illustrate this problem below by returning to our example once again. 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, as being addressed by the work on mobile code security. The problem still persists even if the dependency information is revealed only to $A(t_3)$. 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 it should know both the dependency and the following agent.

The problem depicted above is similar to that addressed by the Chinese Wall Security policy (see section 3). We present a Chinese Wall Security Model for decentralized control to provide secure, fair and trusted execution of a workflow.

2.1 Overview of our Approach

To provide a good understanding of the approach taken to address the problem identified above, we provide an outline of our approach in this section. As a first step, we enhance the workflow model with join dependencies that can correctly specify join relations among task dependencies. We then propose a model for the *decentralized workflow management system* (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. In particular, an agent executing a task t_i, say $A(t_i)$ generates a self-describing workflow, called $Self(P_j)$. This $Self(P_j)$ is comprised of information including the task to be executed (t_j), the agent that is responsible to execute this task ($A(t_j)$), the preconditions to be satisfied that enable the execution of the task ($PreSet(t_j)$), the output state of the task t_i ($OutState(t_i)$), and the remaining workflow containing t_j and its following tasks (partition P_j).

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, identifying the tasks to be executed by that agent, modifying it and re-sending it to the next task execution agent. It is important to note here that some part of the precondition can be evaluated, say at $A(t_i)$ while the other part can only be evaluated at $A(t_j)$. We denote them as *immediate* and *deferred* preconditions, respectively. The WFMS stub identifies these two parts and sends the self-describing workflow to the subsequent task execution agent only when the immediate precondition is satisfied. Only the deferred precondition will be sent as part of the self-describing workflow, which will then be evaluated by $A(t_j)$. The WFMS stub is also responsible for merging the results, if there is exist a join relation and the self-describing workflows arrive from two or more of its prior task execution agents.

We extend the Chinese wall security model for decentralized workflow execution (*DW Chinese Wall Security Policy*), in which we essentially identify *sensitive* objects that allow manipulation of the execution flow to gain competitive advantage, and prevent them from being shared by competing task execution agents. We realize that these sensitive objects are nothing but those that are involved in the task dependencies. We impose access restrictions on these sensitive objects by *restrictive partitioning*, which ensures that no two task agents of the same conflict-of-interest class are in one partition. Our approach ensures that the dependencies involving sensitive objects are evaluated by a task agent for which this object is not a sensitive object.

3 The Chinese Wall Security Policy

In this section, we provide a brief review of the Chinese wall policy that was identified by Brewer and Nash [5] for information flow in a commercial sector. In this model data is viewed as consisting of objects each of which belongs to a *company dataset.* The company datasets are categorized into mutually disjoint *conflict-of-interest* (COI) classes, as shown in figure 4. For example, Banks, Oil Companies, Air Lines are the different conflict-of-interest 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 a different company dataset to the one for which write access is requested.

The BN write rule prevents information leakage by Trojan Horses. For example [20], 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 infected subject executing with John's privileges can transfer information from Bank A's objects to T's objects, which can be read by Jane, who then can read information about both Bank A and Bank B.

4 The Workflow Model

In this section, we develop the necessary formalism to specify the workflow and our secure decentralized approach. 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 pair (G, J) where G is a directed graph (T, D) with $T = \{t_1, t_2 ... t_n\}$, the set of nodes of G that denote the set of tasks in W, $D = \{d_1, d_2 ... d_m\}$, the set of edges of G that denotes the set of intertask dependencies $t_i \xrightarrow{x} t_j$ such that $t_i, t_j \in T$ and x the type of the dependency, and $J = \{j_1, j_2 ... j_p\}$ denotes the set of join relations among the dependencies in D.

Given a workflow, $W = \langle (T, D), J \rangle$, we formally define the tasks, the dependencies and join relations in the reminder of this section.

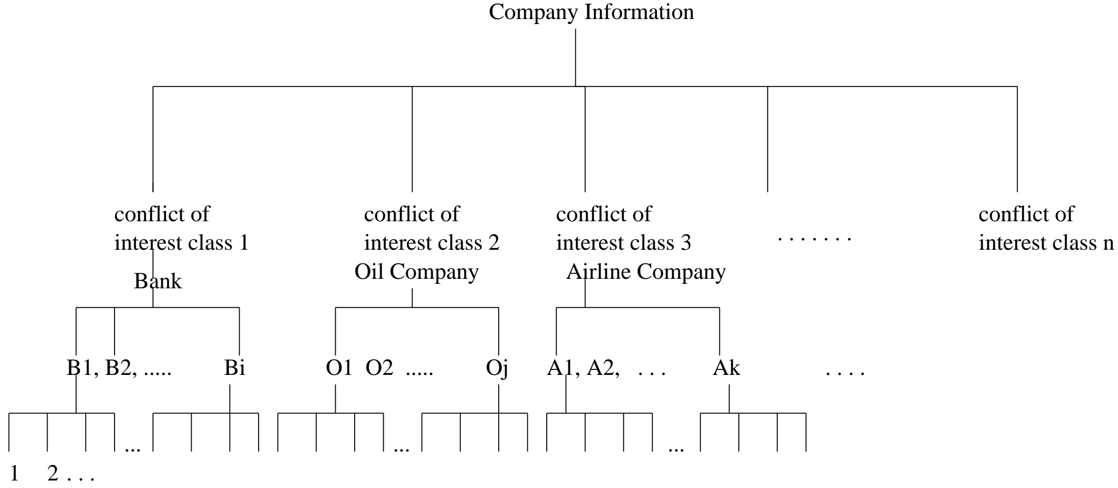

Figure 4: Company Information for the Chinese Wall Policy

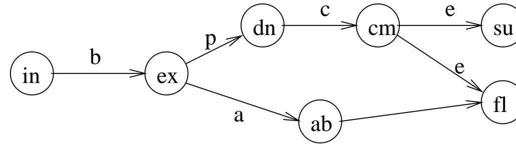

Figure 5: States of a Task

4.1 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 5. Given a task t_i, we assume it comprises of a set of operations to be performed, called OP. 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 (pr_i) moves the task from one state to another, which can be one of the following: *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 primitives 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 $\langle A, \Omega, Input, Output\rangle$, where A denotes the execution agent of t_i, $\Omega = OP \cup PR$, $Input$ the set of objects allowed as inputs to t_i, and $Output$ the set of objects

expected as output from t_i.

In the following, we use the notation $A(t_i), \Omega(t_i), Input(t_i)$, and $Output(t_i)$ to denote the task agent, the set of operations and primitives, the set of input objects, and the set of output objects of t_i, respectively.

Example 2 An example of a task, t_1 = *Purchase a ticket at Continental*, is as follows:
$A(t_1)$ = Continental Travel Agent, $\Omega(t_1) = \{check_seat, check_price, make_invoice\}$,
$Input(t_1) = \{travel_date, destination\}$, and $Output(t_1) = \{invoice_number, ticket\}$

When a task completes its execution, it moves to one of the output states in ST, and generates output objects with a certain finite values. We use $OutState(t_i)$ to denote the output state of a task t_i. In example 1, $OutState(t_2)$ ={s, price=$200}, which denotes that the output state of the task *Reservation at Continental* is successful (su), and the price of the ticket is $200.

4.2 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, 18].

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 $\mathcal{E}$. In the following, we formally define dependencies and the conditions specified on dependencies as *dependency expressions*.

Definition 3 [Literals and Variables]
A literal l is an element in $L = \{R \cup AN \cup ST\}$, where R is the set of real numbers, AN the set of alphanumeric strings, and ST the set of all possible states for tasks in W.
Given a set of tasks T, and a set of dependency variables DV= $\mathcal{S} \cup \mathcal{E} \cup \mathcal{V}$ associated with each $t_i \in T$, a variable v is defined as:
1. If $t_i \in T$ and $vn \in DV$, then $t_i.vn$ is a variable.
2. If v_1 and v_2 are variables, and $op \in \{+, -, /, *\}$ then v_1 op v_2 is a variable.

Definition 4 [Atomic Expression] An atomic expression, a, is defined as follows:

- if v is a variable and $l \in L$, and $op \in \{=, \neq, <, >, \leq, \geq\}$, then $v\ op\ l$ is an atomic expression; and
- if v_1 and v_2 are variables, and $op \in \{=, \neq, <, >, \leq, \geq\}$, then $v_1\ op\ v_2$ is an atomic expression.

Definition 5 [Dependency Expression] A dependency expression, de is defined as follows:

- if a is an atomic expression, then a 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. $t_1.state = su$ is a dependency expression
2. $(t_1.price > \$400 \wedge t_2.seat \geq 2)$ is a dependency expression

Definition 6 [Dependency] Each dependency $t_i \xrightarrow{x} t_j$ 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. $t_1 \xrightarrow{bc} t_2$: $\langle t_1, t_1.state = commit, t_2, begin \rangle$
2. $t_1 \xrightarrow{bc,price>\$200} t_2$: $\langle t_1, (t_1.state = commit \wedge t_1.price > \$200), t_2, begin \rangle$
3. $t_1 \xrightarrow{time=10am,abort} t_2$: $\langle t_1, (t_1.time = 10am), t_2, abort \rangle$

4.3 Join Relations

Join Relations: Join relations represent the logical relationships among the dependencies when a task is involved in multiple task dependencies such as AND-join or OR-join. For example, assume $d_1 = t_i \xrightarrow{x_1} t_j$ and $d_2 = t_k \xrightarrow{x_2} t_j$, where $d_1 = \langle t_i, de_1, t_j, pr_j \rangle$, and $d_2 = \langle t_k, de_2, t_j, pr_j \rangle$. An AND-join states that $de_1 \wedge de_2$ must be true to initiate t_j or rather to invoke the primitive of t_j, pr_j. If a join relationship is not specified between two dependencies d_1 and d_2, then it is not clear whether both dependencies d_1 and d_2 need to be satisfied to initiate t_j, or either one of the dependencies is sufficient. In other words, it is not clear whether it is a OR-Join or an AND-join.

Split Relations: Consider a split relation as in $d_1 = t_i \xrightarrow{x_1} t_j$ and $d_2 = t_i \xrightarrow{x_2} t_k$. Unlike join relations, split relations do not necessarily have to be explicitly specified. An OR-split can be realized by appropriately choosing the conditions in the dependencies, as in $t_2 \xrightarrow{bf \vee price>\$400} t_3$ and $t_2 \xrightarrow{price \leq \$400} t_4$ in example 1. In this case, even if one does not explicitly specify that the split at t_2 is an OR-split, based on the preconditions of the dependencies $t_2 \rightarrow t_3$ and $t_2 \rightarrow t_4$, it is obvious that only one among t_3 and t_4 will be executed, but not both. Therefore, we do not consider split relations further in this paper, but focus only on the join relations.

In this paper, we recognize that there arise many scenarios in which a join relation cannot be expressed exclusively either by an AND-join or by an OR-join. We illustrate this with the following example.

Example 5 Consider a part of a workflow depicted in figure 6, representing a simple hotel reservation. The requirement is to accommodate 10 people by reserving 3 double rooms and 4 single rooms, where the reservation can be made at two nearby hotels, the Country Hill and the Hilton. This can be represented as the following tasks and dependencies.

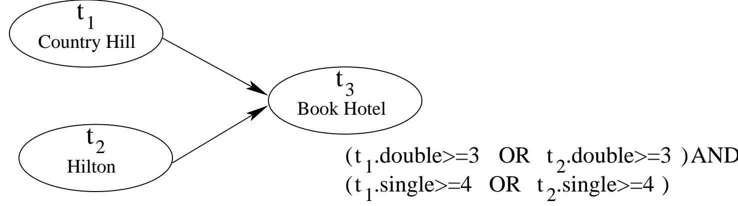

Figure 6: Example of join condition

1. $t_1 = \langle A(t_1)$= Hotel Hilton, $C(t_1)$ = (check availability of rooms), $Input(t_1)$ = (Date, number and type of rooms), $Output(t_1)$ = (availability) $\rangle$
2. $t_2 = \langle A(t_2)$ = Hotel Country Hill, $C(t_2)$ = (check availability of rooms), $Input(t_2)$ = (Date, number and type of rooms), $Output(t_2)$ = (availability)$\rangle$
3. $t_3 = \langle A(t_3)$= Travel agency, $C(t_3)$ = (Book hotel), $Input(t_3)$ = (Date, number and type of rooms, hotel), $Output(t_3)$ = (reservation number) $\rangle$
4. $d_1 = t_1 \xrightarrow{x_1} t_3 = \langle t_1, de_1 = True, t_3, begin\rangle$
5. $d_2 = t_2 \xrightarrow{x_2} t_3 = \langle t_1, de_2 = True, t_3, begin\rangle$
6. jc_1 = *join condition* = $(t_1.double \geq 3 \vee t_2.double \geq 3) \wedge (t_1.single \geq 4 \vee t_2.single \geq 4)$

The above join condition jc_1 states that as long as 6 people can stay together in 3 double rooms and 4 people together in 4 single rooms, the hotel choice of these two groups is not important. It is important to notice that this join condition jc_1 cannot be expressed as a simple OR-join or AND-join of $de_1 = ((t_1.double \geq 3 \wedge (t_1.single \geq 4))$ and $de_2 = ((t_2.double \geq 3 \wedge t_2.single \geq 4))$, since the resultant join condition $jc_2 = ((t_1.double \geq 3 \wedge (t_1.single \geq 4) \vee (t_2.double \geq 3 \wedge t_2.single \geq 4)$ is not equivalent to jc_1.

A simple assignment of truth values proves the non-equivalence of jc_1 and jc_2. Assume that $(t_1.double \geq 3)$ is True, $(t_1.single \geq 4)$ is False, $(t_2.double \geq 3)$ is True and $(t_2.single \geq 4)$ is False. In such a case, jc_1 results in True, where as jc_2 results in False. In addition to being logically not equivalent, these two join conditions have different semantics. The join condition jc_2 states that all 10 people should be booked in the same hotel, either Country Hill or Hilton, which is different from the intention of jc_1.

It is evident from the above example that it is not always feasible to specify the join relation as a logical AND or OR expressions among individual dependency expressions. Following is a different example that supports the above statement in which the join condition involves the output variables of more than one task.

Example 6 Assume there exist two dependencies $t_1 \rightarrow t_3$ and $t_2 \rightarrow t_3$ and the join condition states that t_3 must begin if $t_1.price + t_2.price < 400$ is True. In such a case, the dependency condition can only be evaluated at $A(t_3)$, but cannot be evaluated by neither $A(t_1)$ nor $A(t_2)$.

Here the dependency can be evaluated only at the join task. We denote such dependencies as *self dependencies*, which can be formally defined as follows:

Definition 7 [Self Dependency] A self dependency $t_i \xrightarrow{x} t_i$ is a 4-tuple $\langle hd, de, tl, pr\rangle$, where $hd = tl = t_i$, de is the dependency expression and $pr \in PR$ is the primitive of t_i to be invoked when de is True.

For example, the condition that can be evaluated only at t_3 in example 6 can be represented as a self dependency $t_3 \rightarrow t_3 = \langle t_3, (t_1.price + t_2.price \leq \$400), t_3, begin\rangle$. Given a self dependency, we are now ready to define a join relation, which is a logical expression among dependencies, called *join expression*.

Definition 8 [Join Expression] A join expression je is defined as follows:

- if $d_1 = \langle hd_1, de_1, tl_1, pr_1\rangle$ is a dependency, then d_1 is a join expression;
- if $d_1 = \langle hd_1, de_1, tl_1, pr_1\rangle$ is a self dependency, then d_1 is a join expression;
- if $d_1 = \langle hd_1, de_1, tl_1, pr_1\rangle$ and $d_2 = \langle hd_2, de_2, tl_2, pr_2\rangle$ are two dependencies, then $(d_1 \wedge d_2)$ and $(d_1 \vee d_2)$ are join expressions;
- if $d_1 = \langle hd_1, de_1, tl_1, pr_1\rangle$ is a dependency and je_1 is a join expression, then $(d_1 \wedge je_1)$, $(d_1 \vee je_1)$ are join expressions; and
- If je_1 and je_2 are join expressions, then $(je_1 \wedge je_2)$, $(je_1 \vee je_2)$ are join expressions

Therefore, instead of explicitly representing a join condition as an AND-join or OR-join, we represent that as a join relation whose join expression is composed of dependency expressions. The join relation can be formally defined as follows:

Definition 9 [Join Relation] A join relation j_i is a tuple $\langle t_i, je_i\rangle$, where t_i is the task at which the join relation is specified and je_i is a join expression.

Example 7 The join condition in the workflow in example 5 can be represented as a self dependency $t_3 \xrightarrow{x} t_3 = \langle t_3, (t_1.double \geq 3 \vee t_2.double \geq 3) \wedge (t_1.single \geq 4 \vee t_2.single \geq 4), t_3, begin\rangle$, and a join relation $j_3 = \langle t_3, ((d_1 \wedge d_2) \wedge d_3)\rangle$.

4.4 Precondition Set

The precondition set of t_i, denoted as $PreSet(t_i)$ is a set of conditional expressions that have to be met in order to invoke a primitive in t_i so that t_i moves from one state to the other based on the workflow specification. $Preset(t_i)$ can thus be derived from the task dependencies and join relations. Since the dependencies may specify invocation of one of primitive operations (i.e. begin, commit, abort), we need to distinguish the preconditions for each primitive. For example, $t_i \xrightarrow{bc} t_j$ invokes begin operation for t_j on t_i's commit, while $t_i \xrightarrow{c} t_j$ invokes commit operation for t_j on t_i's commit.

For each task t_i, we thus have three preconditions: Pre_i^b, Pre_i^c and Pre_i^a that must be satisfied to invoke the primitive operations, *begin, commit*, and *abort*, respectively. (Note here that based on the nature of the dependencies, some of the preconditions may always be True.) We have not included the preconditions for the *evaluate* primitive because, typically there will not be any dependency specification that requires to invoke it. $PreSet(t_i)$ thus includes the preconditions for only the three primitive operations for t_i, namely, begin, commit and abort. That is, $PreSet(t_i) = \{Pre_i^b, Pre_i^c, Pre_i^a\}$.

In the following, we derive the preconditions of a task t_i for each primitive operation pr_i from the join expression. Due to definitions 8 and 9, a task dependency is a special case of a join relation. Given a join expression, the following definition helps to extract the part of the join expression relevant to a task primitive.

Definition 10 [Precondition] Let je_i be the join expression of t_i. We define the precondition for pr_j primitive of task t_i as follows: $Pre_i^{pr_j} = \{je_i |$ for each d_k in je_i, substitute d_k with $de_k\}$.

Example 8 Some examples of preconditions for a primitive are:

1. Considering t_3 in example 1, $Pre_3^b = (t_2.state = fl \vee t_2.price > \$400)$, $Pre_3^c = \emptyset$ and $Pre_3^a = \emptyset$.
2. Considering t_3 in example 6, $Pre_3^b = (t_1.double \geq 3 \vee t_2.double \geq 3) \wedge (t_1.single \geq 4 \vee t_2.single \geq 4)$, $Pre_3^c = \emptyset$ and $Pre_3^a = \emptyset$.

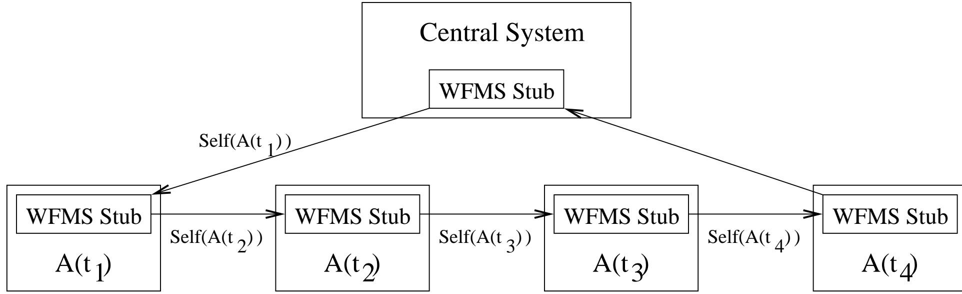

Figure 7: Our Approach to Decentralized Control

5 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 WFMS. In regard to this, we propose (1) *self-describing workflows* and (2) *WFMS stubs*. In the following, we discuss them in detail.

5.1 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 input objects required to execute t, (4) the task agent $A(t)$ that executes t, (5) the precondition to execute t and (6) the output state of the task prior to t. This information is piggy-backed along with t when sending it to its execution agent.

Figure 7 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, 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 to the next task execution agent. We assume that 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 11 [Self Describing Workflow] We define a self-describing workflow, $Self(P_i)$, as a tuple $\langle t_i, A(t_i), PreSet(t_i), OutState(t_i), P_i \rangle$, where t_i is the first task in P_i, $A(t_i)$ the agent that executes t_i, $PreSet(t_i)$ the set of preconditions to be satisfied for t_i to enter a state st_i, $OutState(t_i)$ the output state of t_i, and P_i the workflow partition.

5.2 WFMS stub

A WFMS stub is a small component that can be attached to a task execution agent. This module is responsible for: (1) evaluating the preconditions and sending the tasks to be executed at that site to its execution agent, (2) partitioning the remaining workflow to construct self-describing workflows, (3) evaluating control information in the dependency expression to decide where to send the self-describing workflows, and (4) forwarding each self-describing workflow to the relevant subsequent agents. Since dependency conditions are stored as preconditions, the WFMS stub needs to identify the part of the precondition that can be evaluated at the

current execution agent, say $A(t_i)$ and the part that can be evaluated at the following agent, say $A(t_j)$. Thus the stub at $A(t_i)$ needs to split the precondition of $Pre(t_j)$ of t_j. These two parts of the precondition are referred to as *immediate* and *deferred*, respectively. More details on this are provided in the later sections.

5.2.1 Workflow Partition

The WFMS stub at $A(t_i)$ prepares self-describing workflows for the following task agents $A(t_j)$ by first partitioning the remaining workflow. Following is an algorithm for generating a self-describing workflow.

Algorithm 1 [Workflow Partition Algorithm]

Partitioning(P_i):
Given P_i at A(t_i),
For each t_j where $t_i \xrightarrow{x} t_j$ exists in P_i,
P_j = $\{(T, D) \mid T$ = a set of tasks such that each $t_k \in T$ is reachable from t_j
D= a set of dependencies among tasks in $T\}$
Generate a Self-describing Workflow:
$Self(P_j) = \langle t_j, A(t_j), PreSet(t_j), OutState(t_i), P_j\rangle$

The self-describing workflow $Self(P_j)$ at $A(t_i)$ generated from the above algorithm consists of: the next task to be executed in the workflow (t_j), the agent that executes t_j ($A(t_j)$), the precondition for t_j ($PreSet(t_j)$), t_i's output state after its execution ($OutState(t_i)$), and the workflow partition (P_j). We illustrate the working of the partitioning with the help of an example shown in Figure 8. After task agent $A(t_2)$ finishes the execution of t_2 in $P2$ (shown in Figure 8(A)), it partitions the remaining workflow into two, $P3$ an $P4$, as shown in Figure 8(B). Subsequently, $A(t_4)$ would partition $P4$ into $P6$, as shown in Figure 8(C), and $A(t_6)$ would partition $P6$ into P7, as shown in Figure 8(D). Notice that t_6 and t_7 are included in both $P3$ and $P4$. In fact, if the join relation at t_6 is a simple AND-join, including the workflow that follows t_6 (in this case t_7) in both $P3$ and $P4$ would be redundant and may result in t_7 being executed twice. Algorithm 6, described later, prevents such redundant execution. However, if the join relation at t_6 is a simple OR-join, it is necessary to include t_7 in both $P3$ and $P4$ because even if one path is not sucessful, the workflow could still successfully complete its execution via the other path.

5.2.2 Evaluation of the Preconditions

Specification of the dependency, $t_i \rightarrow t_j$, implies certain preconditions on the primitives of t_j. Therefore, these preconditions need to be evaluated first. It is beneficial, and sometimes essential, to evaluate the preconditions, as much as possible at $A(t_i)$. For example, consider a dependency $t_i \xrightarrow{bs \vee price > \$400} t_j$. After the execution of t_i, the precondition ($bs \vee price > \$400$) must be evaluated in order to decide whether to begin t_j or not. Suppose t_i gets aborted, or the $price \leq \$400$, t_j and all the workflow that follows t_j need not be executed at all. Therefore, it is unnecessary to send all the workflow information (i.e. self-describing workflow) to $A(t_j)$, without first considering the outcome of t_i. Such unnecessary flow of information can be avoided if the precondition evaluation is performed at $A(t_i)$. A more compelling case arises when there is an OR-split at t_i. If $Pre(t_j)$ is not evaluated at $A(t_i)$, a self-describing workflow need to be sent to all the subsequent task agents that appear as parallel nodes in the workflow, even though only one of them actually needs to be forwarded since all the other split cases will turn out to be False. Thus, delaying evaluation of the precondition at $A(t_j)$ might cause unnecessary trasfer of workflow information to task agents. However, it is not always feasible to evaluate the entire precondition at $A(t_i)$. Therefore, it is necessary to first identify the part of the precondition that can be evaluated at $A(t_i)$, and the part that can be evaluated only at $A(t_j)$. In other words, the stub at $A(t_i)$ needs to split the precondition of $Pre(t_j)$ for immediate and deferred evaluation. While

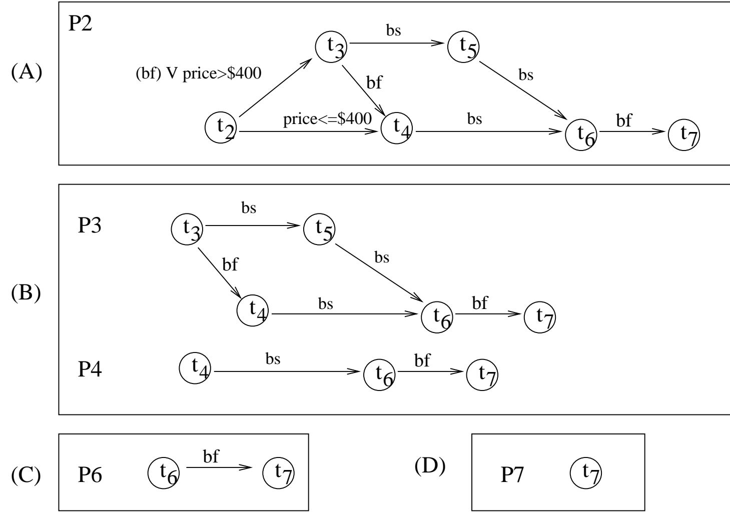

Figure 8: An example to describe partitions

immediate evaluation not only avoids any unnecessary transfer of data, it allows sending only the result of the evaluation, i.e., the truth value, to $A(t_j)$, thereby minimizing the amount of information to be communicated.

5.2.3 The WFMS Stub Algorithm

In the following, we describe the functionality of the WFMS stub at each A(t_i). Given a set of tasks t_i, t_j and t_p, we assume t_p is executed prior to t_i, and t_j after t_i. The WFMS stub algorithm consists of the following steps:

Step 1: After extracting the information from Self(P_i), $A(t_i)$ first evaluates Pre_i^b to check if it can start the execution of t_i. If Pre_i^b is False, the begin primitive of t_i is not invoked. Otherwise, the WFMS stub continues with the next step. In case of a join dependency, the stub at $A(t_i)$ may need to wait for the self-describing workflows of other prior tasks in order to execute t_i. As an example, consider the join dependency, $t_i \xrightarrow{x} t_i = \langle t_i, (t_j.status = begin) \wedge (t_k.status = begin), t_i, begin \rangle$. In order to start t_i, the stub needs to wait for the self-describing workflows arriving from both A(t_k) and A(t_j). If the self-describing worflow from $A(t_k)$ arrives first, $A(t_i)$ remains in a "wait" state until the self-describing workflow from $A(t_j)$ arrives and the precondition of t_i is satisfied.

Step 2: There are cases when the WFMS stub does not need to evaluate the preconditions of its following task t_j. This is possible if there does not exist a precondition for the begin primitive of t_j, i.e., $Pre_j^b = \emptyset$. In this case, the task following t_i can be executed in parallel with that of t_i. In such a case, the WFMS stub at $A(t_i)$ first constructs $Self(P_j)$, and sends it over to $A(t_j)$. For example, consider the dependency $t_2 \xrightarrow{c} t_3$ in the workflow shown in figure 9. 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 can be executed in parallel.

Step 3: $A(t_i)$ executes its own task t_i until it reaches either a done or an abort state.

Step 4: When t_i reaches the done state, the WFMS stub checks Pre_j^c and if it is False, it aborts t_i, otherwise

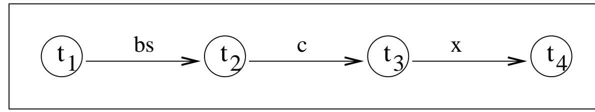

Figure 9: An example to describe WFMS stub

it commits t_i. When a task is executed in parallel along with its preceding task as in step 2, it has to wait (in the done state) for the signal to terminate its execution.

Step 5: After t_i commits (or aborts), the Stub evaluates the $PreSet(t_j)$ associated with the commit or abort state. "Begin on Commit"(bc), "Commit on Commit"(c) and "Abort on abort"(a) are examples of such dependencies evaluated in this step. When $PreSet(t_j)$ evaluates to be True, a self-describing workflow $Self(P_j)$ is generated and sent to the subsequent agents $A(t_j)$.

Step 6: Finally, evaluation of the results obtained from the execution of the task has to be done. This step tests for semantic success or failure of a task. An example of a semantic failure would be, "A flight reservation task ends successfully but there are no seats available". Dependencies such as "begin on success"(bs), "begin on failure"(bf) and "Commit on success"(cs) are evaluated in this step.

The self-describing workflows thus generated, when composed together, must be equivalent to the original workflow. The following defines the equivalence of a $Self(P)$ and the composition of its partitioned workflows.

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 $Self(P')$.

6 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 conflict-of-interest problems that occur in a decentralized workflow environment. We refer to this as the *DW Chinese Wall Security Policy*. 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 objects in the workflow into two: *sensitive* and *non-sensitive*. Intuitively speaking, sensitive data objects are those that could be manipulated to change the execution flow of the workflow. For example, some value dependencies may be considered as sensitive objects.

Let us consider once again example 1 to illustrate sensitive and non-sensitive objects. Consider the dependency $t_2 \stackrel{bf \vee price > 400}{\longrightarrow} t_3$. 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 later decides the airline of his or her 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 are: $price >$ \$400 and $price \leq$ \$400.

Dependency objects have the following COI property:

- 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.

From this, we could formally define a sensitive object as follows:

Definition 13 [Sensitive object] A value dependency object o in a workflow W is sensitive, iff o belongs to a company C and there exists at least one other company $C' \in W$ where $C' \in COI(C)$.

This definition distinguishes sensitive dependency objects from non-sensitive dependency objects. Thus, non-sensitive data objects include data objects that are not involved in any dependency (non-dependency objects), control dependencies as well as value dependency objects that are not in conflict of interest among particular companies in the same workflow.

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 its 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 14 [The DW Chinese Wall Security Policy]

1. **[Evaluation/Read Rule]**: A subject S can read an object O in a workflow W,
   (a) if O belongs to a non-dependency company data set or
   (b) if O is a dependency object where O belongs to S and there is no other company S' such that $S' \in COI(S)$ in the same W.

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 which are not sensitive. 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.

Note that, at a first glance, one may think that this write rule does not include all aspects of the original BN write rule. However, in a workflow context, only task agents are allowed to execute the tasks and therefore no subject can write to a different company data set other than its own. However, the only way a subject can influence the values written to an object is via dependencies by manipulation of its own output. As such, the problem is "similar" to that addressed in the original Chinese wall model, but not the "same." Since the definition of sensitive object and the read rule capture these issues, it would be redundant to state them in the write rule.

7 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. According to the read and write rules of this policy, a task execution agent A cannot read, evaluate, or write any sensitive object that belongs to a different company that is in the same COI class of A. In other words, it is allowed neither to view, nor to construct a self-describing workflow that involves sensitive objects. To accomplish this, we restrict the partitions using the following rule.

Definition 15 [Restrictive Partitions Rule] Let t_i be a task in W, and $P_i \subseteq W$ be a partition sent to $A(t_i)$. A restrictive partition P_i of t_i is such that there exist no sensitive objects for $A(t_i)$ in P_i.

Definition 16 [Critical Partition] A partition P_i is said to be a *critical partition* if it does not comply with the restrictive partition rule.

If an object o belongs to a task agent $A(t_i)$ and there is no other task agent $A(t_k) \in COI(A(t_i))$ in the same partition P_i, then P_i is considered as a restrictive partition.

A trivial solution to obtain restrictive partitions is as follows. When a workflow is submitted to the central WFMS, it partitions the entire workflow into restrictive partitions in advance, i.e. no two task agents belong to the same COI class in each 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 where each partition contains one task.

The challenge, therefore, is to decompose a workflow into partitions that satisfy the restrictive partition rule, without having to introduce any additional centralized control. In the following, we propose our approach to restrictive partitioning that meets this challenge.

7.1 Secure Precondition Splitting

As mentioned in section 5, the dependency expression in $Pre(t_j)$ needs to be split into a condition that can be immediately executed with respect to a task t_i, and a part that can be deferred for evaluation with respect to a task t_i. Moreover, another reason to split the preconditions that involve a sensitive dependency is that the sensitive object must be prohibited from being evaluated or partitioned by the task agents who are in conflict of interest. The following definition distinguishes these two parts of the precondition.

Definition 17 [Immediate and Deferred Expression for t_i] Given a precondition $Pre(t_j)$ and dependency $t_i \rightarrow t_j$, an atomic expression a is

1. $immediate$ with respect to t_i, denoted as $immediate(t_i)$, if a contains a task name t_i in its variable.
2. $deferred$ with respect to t_i, denoted as $deferred(t_i)$, if a contains a task name other than t_i in its variable.

Example 9 In the following, we show the immediate and deferred parts of the precondition for two cases – a single dependency and a join relation.

1. Consider a dependency $t_1 \xrightarrow{x_1} t_2$, where $Pre_2^b = (t_1.status = su \wedge t_2.\textit{temperature_machine} > 400F)$. This is a typical example of workflow used in the heavy industry where a process can be done properly only under certain physical conditions. Here $A(t_1)$ is able to evaluate just part of $Pre(t_2)$, i.e., only $(t_1.status = su)$. But it cannot evaluate the remaining part, $(t_2.\textit{temperature_machine} > 400F)$ since it has no information about the temperature reached by the machine at t_2 at a given time. Therefore, this part of the precondition has to be sent over to $A(t_2)$ for its evaluation. Thus, $immediate(t_1) = (t_1.status = su)$ and $deferred(t_1) = (t_2.\textit{temperature_machine} > 400F)$.

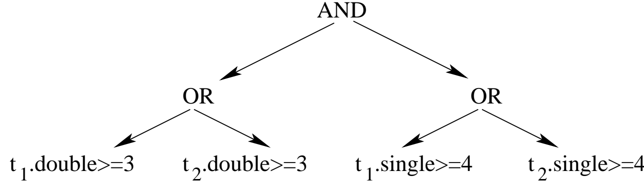

Figure 10: Example of binary tree representation of precondition

2. Consider a join relation $j_1 = \langle t_3, d_1 \wedge d_2 \wedge d_3 \rangle$ where $de_1 = (t_1.status = su)$, $de_2 = (t_2.status = su)$ and $de_3 = (t_1.price{+}t_2.price \geq \$200)$. It is obvious that while $immediate(t_1) = t_1.status = su$ and $deferred(t_1) = (t_2.status = su) \wedge (t_1.price + t_2.price \geq \$200)$, $immediate(t_2) = t_2.status = su$ and $deferred(t_2) = (t_1.status = su) \wedge (t_1.price + t_2.price \geq \$200)$. The join task agent $A(t_3)$ needs to take care of the synchronization of the outputs of t_1 and t_2.

In the following, we present algorithms that split a precondition expression into immediate precondition, denoted as $immediatePre(t_i)$, and deferred precondition, denoted as $deferredPre(t_i)$. A precondition expression is represented as a binary tree where each leaf node is an atomic expression. Non-leaf nodes are logical connectors such as AND and OR that connect left and right expressions to represent a complex logical expression. To illustrate the dependency splitting algorithm, we use the following example:

Example 10 Considering once again the workflow in example 7, the precondition for the begin primitive of task t_3 of this dependency $Pre^b_3 = (t_1.double \geq 3 \vee t_2.double \geq 3) \wedge (t_1.single \geq 4 \vee t_2.single \geq 4)$. Figure 10 shows the tree representation of Pre^b_3.

As mentioned earlier, the preconditions that involve a sensitive dependency need to be split. The following definition formalizes these concepts.

Definition 18 [Sensitive Expression for t_i] Let there be a workflow partition P_i, a dependency $t_i \rightarrow t_j$ in P_i, and precondition $Pre(t_j)$. An atomic expression $a \in Pre(t_i)$ is $sensitive$ with respect to t_i, denoted as $sensitive(t_i)$, if a contains a task name t_i in its variable name, vn, such that vn belongs to $A(t_i)$, and there exists a task t_k in P_i such that $A(t_k) \in COI(A(t_i))$.

Algorithms 2 and 3 split a precondition into immediate and deferred expressions, respectively, and ensure that the immediate precondition does not include atomic expressions considered sensitive to the task involved.

Given $Pre(t_j)$ as input to the WFMS stub at $A(t_i)$, algorithm 2 traverses the complete tree in post-order manner, and generates the immediate tree by substituting each leaf containing deferred expressions with the label $dexp$. We use $dexp$ to represent sensitive expressions even though they are part of the immediate expression. We recursively collapse each subtree containing only deferred expressions with a single $dexp$. This way, the size of the tree is kept to its minimum, thereby reducing the traversal cost while evaluating the expression. For the rest of the tree, we use Tree.exp to denote the expression associated with each node in the tree. In each leaf node in the resulting tree, it can be an immediate expression or a $dexp$. We denote the truth value of the tree as Tree.val. The immediate tree for the example in figure 10 generated by algorithm 2 is shown in figure 11.

Algorithm 2 [Immediate Tree in WFMS stub at $A(t_i)$]

```
Input: Tree, t_j /* a binary tree for Pre(t_j)*/
Output: immediatePre(t_j)
ImmediateTree:
    if ((left(Tree) ≠ NULL) ∧ (right(Tree) ≠ NULL)){
      ImmediateTree(left(Tree),t_j);
      ImmediateTree(right(Tree),t_j);
      if ((left(Tree).val == False) ∧ (right(Tree).val == False)) {
        /* The Subtree contains only deferred expressions, so it collapses to a single dexp*/
        right(Tree)=left(Tree)=NULL
        Tree.exp=dexp
        Tree.val=False }
      else {Tree.val=True }}
    else {
      /*Evaluation of the leaves of the tree*/
      if (Tree.exp == immediate(t_j)) {
        if (Tree.exp == sensitive(t_j) {
          Tree.exp = dexp
          Tree.val =False
        else {
            Tree.val=True }
      else {Tree.exp = dexp
           Tree.val=False }
```

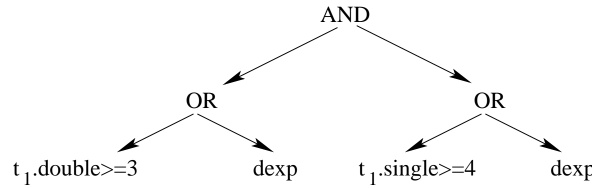

Figure 11: An Example of Immediate tree

Algorithm for *deferredPre*(t_j) is similar to that of *immediatePre*(t_j), except for the following. Given $Pre(t_j)$ as a binary tree, Deferred Tree in Algorithm 3 replaces nodes with immediate expressions and the subtree containing all nodes with immediate expressions with unique *signal variables*. Signal variables are place holders used to transmit and receive the results (i.e. truth values) of evaluated immediate expressions. These can be defined as follows:

Definition 19 [Signal Variables] Given $t_i \in T$, literal $complete$, and $n \in N$, where N is a set of natural numbers, a signal variable $s_i \in S = \{s_1, s_2, ...\}$ for t_i is defined as a string:

1. $t_i.signal\#n$ is a signal variable that ranges over truth values $\{True, False, Undecided\}$.

2. $t_i.signal.complete$ is a signal variable that ranges over truth values $\{True, False\}$ for evaluation result of complete $Pre(t_i)$.

Given $Pre(t_j)$ as a binary tree, algorithm 3 executing at WFMS stub of $A(t_i)$ keeps the $deferred(t_j)$ and $sensitive(t_j)$ in $Pre(t_j)$ and converts non-sensitive $immediate(t_j)$ expressions into unique signal variables. Its output is a deferred precondition, $deferredPre(t_j)$. Note that this algorithm makes a function call to $signal_number(Tree)$, which takes a signal variable $t_j.signal\#n$ in $Tree.exp$ and returns its number n. This function is used by the algorithm to keep the counter n consistent when a subtree containing only signal variables collapses to a single signal.

Algorithm 3 Deferred Tree in WFMS stub at $A(t_i)$

Input: Tree, t_j
    k = 0 /* variable for unique n in $t_j.signal\#n$ */
Output: $deferredPre(t_j)$
DeferredTree:

```
if ((left(Tree) ≠ NULL) ∧ (right(Tree) ≠ NULL)){
  DeferredTree(left(Tree),t_j,k)
  DeferredTree(right(Tree),t_j,k)
  if ((left(Tree).val == True) ∧ (right(Tree).val == True)) {
    /* left(tree) and right(tree) are both signals. Hence, the complete subtree is folded up
    and a single signal is used*/
    k = min(signal_number(left(Tree)), signal_number(right(Tree))
    Tree.exp = t_j.signal#k
    right(Tree)=left(Tree)=NULL
    k= k+1
    Tree.val=True}
else {
  if (Tree.exp is immediate(t_j)) {
    if (Tree.exp is sensitive(t_j) {
      /* keep sensitive expression in deferred tree */
      Tree.val=False }
    else {
        Tree.exp = t_j.signal#k
        k= k+1
        Tree.val=True }
  else {Tree.val=False }
  }
```

Figure 12 shows the resulting deferred tree for the example in figure 10. In summary, the immediate and deferred tree algorithms splits the Pre_3^b of example 10 as follows: $immediatePre(t_2)$: $((t_1.double \geq 3) \vee dexp)) \wedge ((t_1.single \geq 4) \vee dexp)$, and the $deferredPre(t_2)$: $(t_1.signal\#1 \vee t_2.double \geq 3) \wedge (t_1.signal\#2 \vee t_2.single \geq 4)$.

Now let us consider another example of precondition splitting. Assume $Pre(t_2) = ((t_1.double \geq 3) \wedge (t_1.single \geq 4)) \vee ((t_2.double \geq 3) \wedge (t_2.single \geq 4))$. In this case, the immediate tree algorithm will substitute the whole right subtree with $dexp$, as shown in figure 13, resulting in $immediatePre(t_2) = ((t_1.double \geq 3) \wedge (t_1.single \geq 4)) \vee dexp$. The deferred algorithm also substitutes the whole left subtree with a single signal, as shown in figure 14, resulting in $deferredPre(t_2) = t_1.signal\#1 \vee ((t_2.double \geq 3) \wedge (t_2.single \geq 4))$.

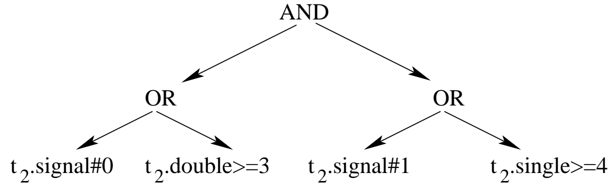

Figure 12: An Example of Deferred tree

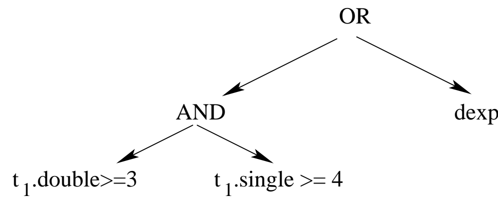

Figure 13: Another Example of Immediate tree

7.2 Precondition Evaluation

Given that a precondition $Pre(t_j)$ is split into $immediatePre(t_i)$ and $deferredPre(t_i)$, a task agent $A(t_i)$ can evaluate $immediatePre(t_i)$ right after t_i's execution, while deferring evaluation of $deferredPre(t_i)$ to $A(t_j)$. In the evaluation of precondition expressions, the stub uses the following principles[1]:

1. Evaluate dependencies only if needed
2. Minimize the amount of information to be sent
3. Reveal only the need-to-know information

The following examples illustrate these evaluation principles:

1. Let $Pre(t_j)$ = $((t_i.status = su \wedge t_i.price < \$100) \vee t_j.time$ = 10:00 pm). If the task t_i enters a successfully finished state, and the price of room is less than $100, the left part of the expression is True, resulting the whole expression to be True since it is connected with OR. Thus, the right hand side expression $t_j.time$ = 10:00 pm does not need to be evaluated. $A(t_i)$ could send the self describing workflow, a special signal $t_1.signal.complete$=True so that t_j can start without any further evaluation. On the other hand, if the task t_i results in a fail state, the truth value of the $Pre(t_j)$ cannot be determined. In this case, $deferredPre(t_i)$ needs to be evaluated at $A(t_j)$.

[1]The motivation for these principles is similar to the shortcut in programming.

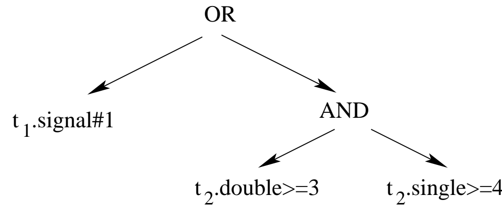

Figure 14: Another Example of Deferred tree

2. Let $Pre(t_j)$ = $((t_i.status = su \wedge t_i.price < \$100)$ $\wedge(t_j.time$ = 10:00 pm)). If t_i fails, then the complete $Pre(t_j)$ evaluates to $False$, and $deferred(t_i)$ = $(t_j.time$ = 10:00pm) does not have to be evaluated. In this case the $deferredPre(t_i)$ does not have to be constructed at all. The signal variable $t_i.signal.complete = False$ is sent to t_j to notify that $Pre(t_i)$ is not $True$.

In the following sections, we present algorithms for evaluating immediate and deferred preconditions that implement the above general evaluation principles.

7.2.1 Immediate Precondition Evaluation

The WFMS stub at $A(t_i)$ evaluates $immediatePre(t_j)$, by traversing the precondition expression tree in a pre-order manner. Hence, the truth values of the immediate expressions are determined at the leaf nodes first. A function $eval$ reads from $OutState(t_i)$ the values assigned to variables in the expression and evaluates it. In case of Tree.exp=$dexp$, the function $eval$ returns $Undecided$.

Once the leaf node is evaluated, the next subtree is evaluated based on the truth values assigned for the left and right branches, and according to the logical operator (AND or OR) of the node itself. The details of these steps are presented in Algorithm 4.

Algorithm 4 [Evaluate Tree at $A(t_i)$]

Input: Tree, t_i
Output: $True$ if Tree is completely evaluated as True
$False$ if Tree is completely evaluated as False
$Undecided$ for partially evaluated tree
evaluateTree:

```
if ((left(Tree) ≠ NULL) ∧ (right(Tree) ≠ NULL)){
   /*The algorithm evaluates the leaves of the immediatePre(t_i) tree*/
   Tree.val = eval(Tree.exp)
   return(Tree.val) }
else{
     left = evaluateTree(left(Tree),t_i);
     right = evaluateTree(right(Tree)t_i);
     case(op(Tree.exp) == AND ): {
           case (left == False ∨ right == False): {return False}
           case (left == True ∧ right == True): {return True}
           otherwise : {return Undecided }}
     case(op(Tree.exp) == OR): {
           case (left == True ∨ right == True): {return True}
```

case (left == $False \wedge$ right == $False$): {return $False$}
otherwise: {return $Undecided$ }}
Otherwise: return $Undecided$ }

Evaluation of the immediate tree can result in one of the following three cases:

1. SUCCESS: The evaluation of immediate precondition completes and results in $True$ at $A(t_i)$. In this case the evaluation of the deferred tree is not necessary. The stub can prepare the self-describing workflow for the next task without any further $deferredPre(t_i)$ to be evaluated at $A(t_j)$, by sending a special signal $t_i.signal.complete = True$.

2. FAIL: The evaluation of immediate precondition completes the whole tree evaluation as $False$. If the stub reaches this situation, it does not have to send the self describing workflow to the next task agent, but send a special signal $t_i.signal.complete = False$, and this path is blocked[2].

3. INDETERMINATE: In case of most join relations, the evaluation of immediate precondition is $Undecided$. The reason for this situation is that the complete precondition at $A(t_i)$ includes some deferred expressions whose truth value cannot be resolved by $A(t_i)$, resulting in partially evaluated tree.

In case of a partially evaluated tree, the WFMS stub at $A(t_i)$ sends the $deferredPre(t_i)$ to the next task agent $A(t_j)$ along with results from the evaluation of $immediatePre(t_i)$ using $OutState(t_i)$. Signal variables and their value pairs from immediate evaluation at $A(t_i)$ generated by algorithm 5 need to be included in the self-describing workflow so that $A(t_j)$ will be able to evaluate the $deferredPre(t_i)$. The next step is to prepare signal variables in $OutState(t_i)$ for each evaluated expressions. This step is spelled out in algorithm 5. Note that the signal variables are generated in numeric sequence so that when the tree is traversed by another task agent, there will be no ambiguity.

Algorithm 5 [Assign Signal at $A(t_i)$]
/* Assign truth value of task t_i to signal variables */

Input:Tree = Output of the algorithm evaluateTree
t_i = task which we want generate the signal
k = variable used for the signals, initially = 0;
Output: Generate signals for evaluated nodes of task t_i
assignSignal:

if ((left(Tree) $\neq$ NULL) $\wedge$ (right(Tree) $\neq$ NULL)){
if ((Tree.exp == immediate(t_i)) $\wedge$ (Tree.val $\in \{False, True\}$)): {
Tree.exp = $t_i.signal\#k$
OutState = OutState $\cup$ { $t_i.signal\#k$ = Tree.val}
k= k+1 }
else {
Tree.val = $Undecided$ }
else {
assignSignal(left(Tree),t_i,k)
assignSignal(right(Tree),t_i,k)
/*The subtree value is compute based on the left/right branches and
the logical expression contained into the node evaluated */

[2]If all the paths in a workflow are blocked, an exception should be raised.

```
    case (Tree.exp == ∧ ): {
        case (left(Tree).val == False ∨ right(Tree).val == False): {
            Tree.val = False }
        case (left(Tree).val == True ∧ right(Tree).val == True): {
            Tree.val = True }
        Otherwise: return {Undecided} }
    case (Tree.exp == ∨): {
        case (left(Tree).val==False ∧ right(Tree).val == False): {
            Tree.val = False }
        case (left(Tree).val==True ∨ right(Tree).val == True): {
            Tree.val = True }
        Otherwise: return {Undecided} }
    /*The subtree is collapsed and both the value and the number of signal in Outstate
    are changed consequently */
    OutState= OutState − {left(Tree).exp=left(tree).val, right(Tree).exp=right(Tree).val }
    k = min(signal_number(left), signal_number(right))
    Tree.exp = t_i.signal#k
    OutState = OutState ∪ {Tree.exp = Tree.val}
    k =k+1
}
```

The above algorithm traverses the tree in a pre-order manner and first checks the immediate expressions that have been already evaluated by algorithm 4, and generates a signal according to the truth value stored in the node. Making use of the same ideas in algorithm 3, the tree is traversed backward and the subtrees containing only signals are systematically collapsed in a way that the signals and their values added into Outstate exactly match the ones generated by algorithm 3.

7.2.2 Deferred Precondition Evaluation

For better readability, we explain the evaluation process using the join condition in example 10. When task t_1 finishes its execution, $A(t_1)$ will send $deferredPre(t_1)$ to the $A(t_3)$ in order to complete the evaluation of the precondition for t_3. Note that its evaluation is not complete at $A(t_1)$. The $deferredPre(t_1)$ has the following logical expression that $A(t_3)$ has to evaluate:
1. $(t_1.signal\#0 \vee t_2.double \geq 3) \wedge (t_1.signal\#1 \vee t_2.single \geq 4)$
Given this $deferredPre(t_3)$ as its precondition, $A(t_3)$ evaluates any immediate expression in it. If evaluation of $immediatePre(t_3)$ could evaluate the entire expression, the evaluation of the remaining part is not necessary. For example, in the case of $((t_1.signal\#0 \vee t_2.double \geq 3) \vee (t_3.time = 10am))$, evaluation of $immediate(t_3)$, i.e. $(t_3.time = 10am)$ can determine the truth value of the entire expression. Thus, it could start the execution of t_3. However, if there is no $immediatePre(t_3)$ or it cannot evaluate the whole expression, then the process of evaluation of precondition starts when $Outstate$ from t_1 arrives with signal and their values. Assume $OutState(t_1)$ = $\{t_1.signal\#0 = False, t_1.signal\#1 = False\}$. Evaluating the $deferred(t_1)$ at $A(t_3)$ will result in:
2. $(False \vee t_2.double > 3) \wedge (False \vee t_2.single \geq 4)$.
This expression cannot be evaluated unless we have more information from $A(t_2)$. Once $A(t_2)$ executes its activities and sends $Self(P_3)$ to $A(t_3)$, which contains the $deferredPre(t_2)$, its signal variable and value pairs in $OutState(t_2)$ as shown in the following logical expression:
3. $(t_1.double \geq 3 \vee t_2.signal\#0) \wedge (t_1.single \geq 4 \vee t_2.signal\#1)$

Once $Self(P_3)$ from $A(t_2)$ arrives with $deferredPre(t_2)$, the evaluation of deferred precondition resumes with $(False \vee t_2.double > 3) \wedge (False \vee t_2.single \geq 4)$ as in 2 above. First, it replaces the $immediate(t_2)$ in $(False \vee t_2.double > 3) \wedge (False \vee t_2.single \geq 4)$ with signal variables, using algorithm 3, resulting in:

4. $(False \vee t_2.signal\#0) \wedge (False \vee t_2.signal\#1)$

Using the $OutState(t_2)$, this expression $(False \vee t_2.signal\#0) \wedge (False \vee t_2.signal\#1)$ in (4) could be evaluated. Let $OutState(t_2) = (t_2.signal\#0 = True, t_2.signal\#1 = True)$. Using this, the entire deferred preconditions are merged and evaluated to be:

5. $(False \vee True) \wedge (False \vee True)$,

which results in $True$. Now that the $Pre(t_3)$ is satisfied, task t_3 can start its execution.

Evaluation of deferred precondition for join relations requires an extra step that merges each deferred tree as soon as it arrives. This evaluation process continues only until the truth value for the whole precondition is determined. This way, evaluation of $Pre(t_3)$ and execution of t_3 do not have to be delayed until all the results from other dependent tasks arrive. As the partial information from previous tasks are available, the WFMS stub at join task agent $A(t_3)$ tries to evaluate the whole expression based on the available information. When the evaluation of the whole expression is possible, it does not have to wait for further results. The algorithm for deferred Precondition evaluation by the WFMS stub at $A(t_i)$ is shown in algorithm 6. The expression with signals can be evaluated only with signal values that are transmitted from previous tasks. This is a simple tree traversal evaluating each leaf node with given signal values. Intermediate nodes with logical operators $op \in \{\wedge, \vee\}$ are evaluated by linking left and right nodes with the *op*, according to the logical rules. This process continues until the root node is evaluated to be either *True* or *False*.

Algorithm 6 [Merge and Evaluate Precondition at $A(t_i)$]
/* Merge $deferredPre(t_i)$ as it arrives and evaluate it at t_k */

Input: $deferredPre(t_i)$
Output: Evaluated Tree
MergeEvaluate:
EvalT = $(deferredPre(t_i))$
While (EvalT.val == $Undecided$) {
  EvalT.val = evaluateTree(EvalT, t_i)
  (a) if (EvalT.val $\neq Undecided$) { return(EvalT.val)}
  wait
  until (($Self(P_j)$ arrives) $\vee$ (time_out))
  (b) if (time_out) { return(error) }
  (c) if ($t_j.signal.complete \in \{True, False\}$) {
    EvalT.val= $t_j.signal.complete$
    return(EvalT.val) }
  EvalT = DeferredTree(EvalT, t_j) }
}

For example, the above signal expression $(t_1.signal\#0 \vee t_2.signal\#0) \wedge (t_1.signal\#1 \vee t_2.signal\#1)$ is evaluated first at leaf nodes substituting $t_1.signal\#0$ with $False$, $t_1.signal\#1$ with $False$, $t_2.signal\#0$ with $True$ and $t_2.signal\#1$ with $True$, resulting in $(False \vee True) \wedge (False \vee True)$. The next step is to evaluate intermediate nodes $(False \vee True)$ into $True$ and $(False \vee True)$ into $True$, resulting in $(True \wedge True)$. The final evaluation is at the root level node with $(True \wedge True)$, resulting in $True$. Thus, the precondition at at t_3 is evaluated to be *True*, thereby starting the execution of task t_3.

7.3 Secure Workflow Partitioning

Given a partition P_j at $A(t_i)$ with $t_i \rightarrow t_j$, P_j needs to be further partitioned if it is a critical partition. The following algorithm generates partitions that obey the restrictive partition rule. The basic idea behind it is to use a neutral task agent to evaluate sensitive deferred dependency information. It considers two cases: (1) the sensitive dependency object lies between two adjacent task agents, in which case, it needs to use a third neutral task agent who could evaluate the sensitive expression; and (2) the sensitive dependency object exists among task agents that are not adjacent, in which case the intermediate task agent could evaluate the sensitive dependencies. In both cases, partition P_j is further partitioned into two: P_{j1} and P_{j2}, separating $A(t_j)$ from other agents in the same $COI(A(t_j))$. Next, the precondition is split into the immediate precondition without any expression sensitive for $A(t_j)$, and the deferred precondition with the sensitive part of the expression for the neutral task agent. In this way, the sensitive information does not leak to $A(t_j)$ and to $COI(A(t_j))$.

Algorithm 7 [Restrictive Partition at $A(t_i)$]

Input: Partition P_i,t_j
Output:Partition P_j such that $\neg\exists t_k \in P_j \mid COI(A(t_j)) = COI(A(t_k))$
Restrictive Partition:
Given P_i at $A(t_i)$,
P_j = Partitioning(P_i)
for each partition P_j
  if $\exists t_k \in P_j \mid COI(A(t_j)) = COI(A(t_k))$ { /* if P_j is critical */
    Given t_m such that $t_j \xrightarrow{d} t_m$ in P_j
    P_m = remove the task t_j and the dependency d from P_j
    $immediatePre(t_j)$= ImmediateTree(d,t_j)
    $deferredPre(t_j)$= DeferredTree(d,t_j,0)
    Case 1: $COI(A(t_j)) = COI(A(t_m))$
        /*both t_j and t_m cannot read the sensitive part of d */
          add a dummy task t_i^d at $A(t_i)$ such that C(t_i)=$\emptyset$
          d_{secure}=$\langle\, t_j, immediatePre(t_j), t_i^d, pr \,\rangle$
          $P_j = t_j \xrightarrow{d_{secure}} t_i^d$ /* add task and dependency to P_j */
          d_{secure}= $\langle\, t_i^d, deferredPre(t_j), t_m, pr \,\rangle$
          $P_d = P_m \cup t_i^d \xrightarrow{d_{secure}} t_m$
          Forward Self(P_i^d) to $A(t_i)$
          Return P_j
    Case 2: $COI(A(t_j)) \neq COI(A(t_m))$
        $Pre_m^{pr} = deferredPre(t_j)$
        Foward Self(P_m) to A(t_m)
        $d_{secure} = \langle\, t_j, immediatePre(t_j), t_m, pr \,\rangle$
        $P_j = t_j \xrightarrow{d_{secure}} t_m$
        Return P_j }
  else { Return P_j }

Note that when the above algorithm makes restrictive partitions, P_i and P_j in the adjacent case and P_j and P_m for the non-adjacent case, it splits the actual dependency connecting these two restrictive partitions into immediate and deferred trees, and there are *signals* sent between these partitions that send results of dependency evaluation. This way, the sensitive dependency is hidden from task agents who are in conflict of interest. We use our travel plan example (example 1) to better explain our algorithm.

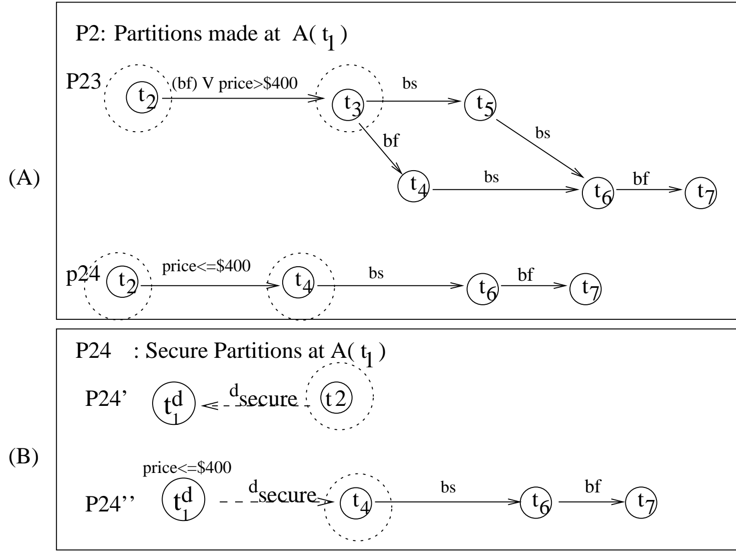

Figure 15: An example for restrictive partitions for adjacent tasks

Example 11 Assume P_1 be the workflow shown in figure 1 sent to the $A(t_1)$. When t_1 finishes its execution, the WFMS stub at $A(t_1)$ has to split the remaining workflow. As can be seen, partition P_2 is critical because the dependencies $t_2 \stackrel{bf \vee price > \$400}{\longrightarrow} t_3$ and $t_2 \stackrel{price \leq \$400}{\longrightarrow} t_4$ belong to task t_2 that cannot read them (due to the read rule). Since there exist two outgoing edges from t_2, the critical partition P_2 is split into two new partitions, P_{23} and P_{24} as shown in figure 15(A).

To show how the restrictive partition algorithm works, we will consider only the partition P_{24} since the solution for the partition P_{23} is almost the same. We are in the case where the dependency object belongs to both $COI(t_m)$ and $COI(t_j)$. Hence first we remove the dependency $t_2 \stackrel{price < \$400}{\longrightarrow} t_4$ and we split P_{24} into two: $P'_{24} = \{t_2\}$ and $P''_{24} = \{t_4 \stackrel{bs}{\longrightarrow} t_6, t_6 \stackrel{bf}{\longrightarrow} t_7\}$.

Now we have to evaluate the dependency $t_2 \stackrel{price < \$400}{\longrightarrow} t_4$. We introduce a dummy task t_1^d at $A(t_1)$, whose function is only to keep the output from t_2 and use it to evaluate the sensitive deferred dependency between t_2 and t_4. In this case the dependency consists of all sensitive predicates. Therefore, the algorithm splits the precondition into $immediatePre(t_2)$ and sensitive $deferredPre(t_2)$. The deferred precondition, $deferredPre(t_2)$, that is sensitive to both t_2 and t_4 has to be evaluated at $A(t_1^d)$. The $immediatePre(t_2)$ contains a signal between t_2 and t_1^d so that $P'_{24} = \{t_2 \stackrel{d_{secure}}{\longrightarrow} t_1^d\}$ as shown in 15(B). When $A(t_2)$ completes the execution it will evaluate immediate tree and prepare $output$ for t_1^d in a self-describing workflow to inform $A(t_1^d)$ that it can start evaluation of the sensitive deferred dependency with $Output$ and $Outstate$ forwarded by the stub in $A(t_2)$. We have used a dashed edge to represent the signal.

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 $A(t_1^d)$ would proceed. We preserve the control flow logic of the original $t_2 \stackrel{price \leq \$400}{\longrightarrow} t_4$ by including this dependency as a precondition $Pre(t_1^d)$.

If the dependency contains both control-flow dependency and sensitive value dependency (as in $t_2 \stackrel{bf \vee price > \$400}{\longrightarrow} t_3$ in P_{23}), the precondition splitting will generate immediate precondition containing dependency expressions (i.e. the control flow bf) that should be evaluated after the execution of original task (in that case t_2), and sensitive deferred precondition (i.e. sensitive dependency like the value dependency $price > \$400$) that should be evaluated in the dummy task (t_1^d), so that reading the sensitive information by the agents of the same COI class, $A(t_2)$ as well as $A(t_3)$, is prohibited. $A(t_1^d)$ contains enough information to evaluate the precondition

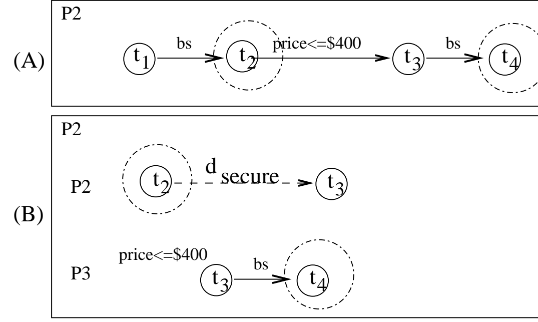

Figure 16: An example for restrictive partitions for non-adjacent tasks

$price < \$400$ and if the condition evaluates to *True*, it splits the partition P''_{24} as (t_4, t_6, t_7) and send the workflow to the next task like usual.

Example 12 We explain handling of the non-adjacent cases with this example. In the workflow shown in figure 16(A) we suppose that non-adjacent tasks t_2 and t_4 are in the same COI. We also suppose that the dependency $t_2 \overset{price \leq \$400}{\longrightarrow} t_3$ is sensitive for $A(t_2)$ (and of course for $A(t_4)$). When $A(t_1)$ 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 $t_2 \overset{price \leq \$400}{\longrightarrow} t_3$. In this case, different from the one in example 11, the agent $A(t_3)$ which is not in conflict with $A(t_2)$ or $A(t_4)$ can read the dependency and can evaluate that.

To do that, t_1 has to split the workflow into two parts. $P2$, which includes t_2 and a signal 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 the dependency into two. $ImmediatePre(t_2)$ including the non-sensitive objects that can be evaluated in $A(t_2)$, and $deferredPre(t_2)$ including the sensitive objects that have to be evaluated in $A(t_3)$ using the $Outstate(t_2)$ sent to $A(t_3)$. In this fashion, sensitive objects are evaluated by $A(t_3)$, by not revealing them to $A(t_4)$ as the conditions are already evaluated before the workflow is sent to it.

Algorithm 8 shows how self-describing workflows are executed at each task agent's WFMS stub. It handles the conflict-of-interest problem among task agents by employing the DW Chinese Wall policy using the restrictive partitioning algorithm and secure dependency splitting. The functions of the WFMS stub (at the central as well as at the execution agents) are outlined in this algorithm.

Algorithm 8 [Secure WFMS Stub at $A(t_i)$]

Secure WFMS Stub:

Given a $Self(P_i)$
  Extract the task t_i to be executed in $A(t_i)$
  Extract Outstate from $Self(P_i)$ and then extract t_p, the preceding task from the outstate
  if (t_p.signal.complete == F) { raise an exception }
  $P_j \leftarrow$ Restrictive Partition(P_i)
  If ($Pre^b_i \neq \emptyset$) {
    Result $\leftarrow$ MergeEvaluate(Pre^b_i,t_p,t_i)
    if (Result $\neq$ True) { raise an exception }}
  If (Immediate(Pre^b_j, t_i) == $\emptyset$) {

```
    PreSet(t_j) ← deferredPre(Pre_j^pr, t_i)
    Outstate ← Null
    Foward Self(P_j) to A(t_j) }
execute t_i until ((state(t_i) == done) ∨ (state(t_i) == abort))
if (state(t_i) == done) {
  if (Pre_i^c ≠ ∅ ) {
     Result ← MergeEvaluate(Pre_i^c,t_p,t_i)
     if (Result ≠ True ) { abort }
     else { commit }
  else { commit } }
if (there exists t_i.status=="cm" ∨ t_i.status=="ab") in Pre_j^pr {
  Generate_self(Pre_j^pr,t_i,t_j,P_j) }
  Forward Self(t_j) to A(t_j) }
else { evaluate
  if (in Pre_j^pr is there t_i.status=="su" ∨ t_i.status=="fl") {
    Generate_self(Pre_j^pr,t_i,t_j,P_j) }}
```

Here we explain the details of the above alogorithm. Assume t_i is the task under consideration, which is preceded by t_p and followed by t_j. The algorithm first checks if Pre_i^b is satisfied. If so, before starting the execution of t_i, it checks if t_j (the following task) can be executed in parallel with t_i, which is the case if Pre_j^b is empty. If so, it will generate $self(P_j)$ to $A(t_j)$. After executing t_i (that is, t_i reaches its done or abort state), $A(t_i)$ evaluates the Pre_i^c before it commits t_i. Finally, the $A(t_i)$ evaluates the precondition of the following task by evaluating t_i's outputs and sends the $self(P_j)$ to $A(t_j)$.

7.4 Summary of our approach

In this section, we provide a summary of the approach proposed in this section. When a self-describing workflow $Self(P_i)$ reaches a task execution agent $A(t_i)$, its WFMS stub (algorithm 8) extracts the task (in this case, t_i) that must be executed by that task agent. In addition to this, to facilitate the decentralized control, it performs the following functions:

1. Evaluation of the precondition.

2. Construction of self-describing workflows.

The evaluation of the precondition is done either for t_i itself, or for the task(s) that follow t_i. The former case arises when t_i is involved in a join dependency or when it is being executed in parallel along with its preceding task. In this case, the task has to wait for information from its preceding tasks and then evaluate the precondition of t_i. In all other cases the evaluation of the precondition of the tasks that follow is done by the WFMS stub at $A(t_i)$. Evaluating a dependency is done by evaluating the logical expression (algorithm 4). When it cannot be completely resolved, it waits for additional information. It is important to note that the precondition is evaluated only if and when it is really needed.

When a workflow needs to be sent to the following task agent(s), a $Self(P_j)$ for each task t_j that follows t_i is generated. In order to construct it, the precondition is first split into two: the part that can be evaluated at $A(t_i)$ (using algorithm 2), and the logical expression that has to be deferred and hence sent to the task agent following t_i (using algorithm 3). An important feature of these two algorithms is that in case of a critical partition, the split is done in a way that the DW Chinese Wall policy is satisfied. More specifically,

the sensitive part is evaluated either by a task agent for which this is not sensitive, or by a dummy task. The Outstate is generated in a way to keep the information flow between the tasks at minimum, by sending only the data that is really needed to complete the evaluation of the precondition. The workflow, thus is partitioned using algorithm 7, and $Self(P_j)$ is constructed and sent to $A(t_j)$.

8 Proofs

8.1 Proof of Security

Theorem 1 The restrictive partition algorithm (algorithm 7) enforces the read and write rules of the DW Chinese wall policy.

Proof: As per section 6, let subjects be the task execution agents and objects be data objects or dependency objects. It follows from definition 13, an object o is said to be a sensitive object if o belongs to a company C and there exists another company in the same COI class as that of C. Since only a task execution agent $A(t_i)$ is allowed to execute the operations of t_i, no subject other than $A(t_i)$ can read or write data objects that belongs to that company data set. Therefore, condition (a) of the evaluation/read rule as well as the write rule of the DW Chinese wall security policy (definition 14) are satisfied.

From definitions 15 and 16, a critical partition P_j of t_j is a partition of a workflow that does not consist of sensitive objects for $A(t_j)$. As per definition 18, a sensitive expression in P_j does not comprise of any sensitive objects with respect to t_j.

Without loss of generality, let us consider that there exists one dependency $t_j \rightarrow t_k$ that follow t_i. The restrictive partitioning algorithm (algorithm 7) executed at $A(t_i)$ computes the immediate tree and deferred tree for t_j. From algorithm 2 (second statement in the else part), an object that is sensitive to t_j, will not be included in the *immediatePre*(t_j). Similarly, from algorithm 3 (second statement in the else part), the *deferredPre*(t_j) comprises of sensitive objects to t_j. Hence, no precondition that involves a sensitive object is sent to a task execution agent. In other words, it follows from definition 13, no task execution agent is allowed to read a sensitive object o that belongs to a company C such that there exists another company in the same COI class. Hence condition (b) of the evaluation/read rule of the DW Chinese wall security policy (definition 14) is satisfied. Since none of the sensitive objects nor partitions that contain them can be read by a task execution agent, the write rule of the DW Chinese wall security policy is also satisfied. Hence the proof. □

8.2 Proof of Correctness

Theorem 2 Let $Self(P_i)$ be a self-describing workflow. Assume $Self(P_i)$ is decomposed into $Self(P_{i_1})$, $Self(P_{i_2}), \ldots Self(P_{i_n})$ using algorithm 7. Then $Self(P_i) \equiv \cup_{j=1}^{n} Self(P_{i_j})$.

Proof: To prove this theorem, from definition 12 we need to prove the following two parts:
(1) The union of the set of all operations in $Self(P_{i_1})$, $Self(P_{i_2}), \ldots Self(P_{i_n})$ is same as that of $Self(P_i)$, and
(2) for each t_i, the $PreSet(t_i)$ in $Self(P) = PreSet(t_i)$ in $\cup_{j=1}^{n} Self(P_{i_j})$.

Part (1): Consider the algorithm for restrictive partition, algorithm 7. Let us consider the WFMS stub at $A(t_i)$ and assume partition P_i is the input to algorithm 7. According to step 1 of this algorithm, P_i is partitioned using algorithm 1. Without loss of generality, assume there exist one task t_{i_1} such that $t_i \xrightarrow{x} t_{i_1}$ exists in P_i. Let t_{i_j} be a task following t_{i_1} in P_i. We encounter two cases now.

Case (i): There exists a single path from t_{i_1} to t_{i_j}. In this case, t_{i_j} is included only once when P_{i_1} is partitioned later. And therefore, in the $\cup_{j=1}^{n} Self(P_{i_j})$, every task appears only once. As a result, the tasks in $Self(P_i)$ are equivalent to the tasks in $\cup_{j=1}^{n} Self(P_{i_j})$. Hence the union of the set of all operations in $Self(P_{i_1}), Self(P_{i_2}), \ldots Self(P_{i_n})$ is same as that of $Self(P_i)$.

Case (ii): There exist multiple paths from t_{i_1} to t_{i_j}. In this case, t_{i_j} may appear more than once when P_{i_1} is partitioned later. And therefore, in the $\cup_{j=1}^{n} Self(P_{i_j})$, every task t_{i_k} may appear more than once. As a result, the tasks in $Self(P_i)$ are more than the tasks in $\cup_{j=1}^{n} Self(P_{i_j})$. However, the multiple paths are due to the join conditions. As per algorithm 6, each task is executed only once even though there are multiple paths. Therefore, the set of all operations in $Self(P_{i_1}), Self(P_{i_2}), \ldots Self(P_{i_n})$ is same as that of $Self(P_i)$.

In addition to the above two cases, according to algorithm 7 (case 1), sometimes dummy tasks are included to a partition. These tasks are dummy, in the sense that they do not contain any operations. Hence this step does not change the set of operations in the union of all partitions. Therefore, the union of the set of all operations in $Self(P_{i_1}), Self(P_{i_2}), \ldots Self(P_{i_n})$ is same as that of $Self(P_i)$.

Part (2): This can be trivially proved as follows. The first step of the restrictive partitioning algorithm 7 is to execute the partitioning algorithm (algorithm 1). The $PreSet(t_i)$ for each t_i, is not modified in this algorithm. From algorithm 7, the $PreSet(t_i)$ for each task t_i is split into immediate and deferred preconditions. The immediate and deferred tree algorithms merely split the into two, but do not remove any predicates from the $PreSet(t_i)$. As per step (c) of algorithm 6, the immediate expression is evaluated by the WFMS stub at $A(t_j)$, where t_j is the task prior to t_j, and the deferred expression is evaluated at the WFMS stub at $A(t_i)$. The $t_i.signal.complete$ from $A(t_j)$ ensures the successful evaluation of the immediate expression. Therefore, $PreSet(t_i)$ in $Self(P_i) = PreSet(t_i)$ in $\cup_{j=1}^{n} Self(P_{i_j})$. □

9 Related Work

In recent years, several approaches and architectures for decentralized workflow execution have been proposed [2, 27, 13, 6]. EXOTICA/Flowmark on Message Queue Manager [2] describes and implements a distributed and decentralized execution architecture consisting of a process definition node and runtime nodes. Once a process has been defined, this definition is compiled at the definition node. After compilation, the process is divided into several parts and each part is distributed to a runtime node where process instances are executed. The division of the process is based on the users associated with the different nodes and the roles associated with the different activities in the process. Each runtime node has a *node manager* in charge of communicating with the process definition node. Each definition node sends to each involved node's manager only the information pertaining to the activities that will execute at that node. This static information is stored in a *process table*. The manager then starts a *process thread* that is in charge of coordinating the execution of instances using a queue for communicating with other nodes and deciding when an activity is ready for execution.

In METEOR$_2$ [6], the scheduling information is distributed among different task managers. Each task manager is aware of its immediate successors and hence is capable of activating the follow-up task managers once the task it controls terminates. The task manager is designed as CORBA objects with task activation, task invocation, error handling and recovery components. In these approaches, a workflow is pre-partitioned in a central server, thus the partitions are made statically in the central server and distributed to each execution agent, whereas our approach partitions dynamically as the workflow progresses with its execution. As a result, we send the partitions to an agent only when needed, whereas other approaches send the partitions to the execution agents even if they are not executed at all. None of these approaches address the conflict-of-

interest issues while partitioning.

Workflow security issues have been addressed in many studies [6, 3, 4, 24, 11, 16, 12]. However, these studies are geared towards access control issues, but do not address the conflict-of-interest issues. Specifically, [3] proposes an authorization model for workflows, [4] extends it to address separation of duties constraints in workflows, and [24] addresses the issue of delegation. [11, 16, 12] call for the security services needed for web-based workflows, such as authentication, access control, data confidentiality, data integrity and non-repudiation services, to support inter-organizational collaborative enterprises.

In a collaborative and distributed workflow execution, security of mobile agents is achieved through role-based access control [21, 23, 22]. The role based specification model for collaborative systems supports the expression of requirements such as separation of duties, intra-role and inter-role coordination, admission control policies, role activation constraints, and dynamic access control policies, but does not address conflicts-of-interest issues, as in this paper.

Other work in the area of mobile code security where code is executed by untrusted hosts [7, 26, 8, 25] is also relevant to our work. The security concern here is to ensure that sensitive information in the "floating" software is not exploited by malicious hosts to their 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 as in "Onion Routing" approach [10, 17], which is explained below. The motivation of Onion Routing is not to hide the content of messages, but to protect the communications channel against both eavesdropping and traffic analysis, by removing identifying information from the data stream, thereby providing anonymous and private communication. Onion Routing achieves this in the following way: an application, instead of making a (socket) connection directly to a destination machine, makes a socket connection to an Onion Routing Proxy. That Onion Routing Proxy builds an anonymous connection through several other Onion Routers to the destination. Each Onion Router can only identify adjacent Onion Routers along the route. Before sending data over an anonymous connection, the first Onion Router adds a layer of encryption for each Onion Router in the route. As data moves through the anonymous connection, each Onion Router removes one layer of encryption, so it finally arrives as plaintext. As an example, suppose the workflow comprised of three tasks, 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 route the workflow to the right destination dynamically. If the dependency information d in $t_i \xrightarrow{x} t_j$ 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. Moreover, cryptographic technology solutions are computationally expensive. In this paper, we offer a solution that does not employ encryption.

[14, 15] propose a *decentralized label model* for more fine-grained information sharing, while reducing the potential information leakage through uncontrolled propagation among distrusted applications. 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.

10 Conclusions and Future Research

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 for reading and evaluating sensitive dependency objects. 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. We have identified the limitations of the traditional workflow model with respect to expressing the various types of join dependencies, and extended the traditional workflow model. The proposed algorithms to generate self-describing workflows and restrictive partitions take into account this extended workflow model.

Note that although we have portrayed our approach as a solution to resolve the issue of conflicts of interest, one can adopt it for 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.

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] Vijayalakshmi Atluri and Wei-Kuang Huang. An Authorization Model for Workflows. In *Proceedings of the Fifth European Symposium on Research in Computer Security, in Lecture Notes in Computer Science, No.1146, Springer-Verlag*, September 1996.

[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] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Onion Routing for Anonymous and Private Internet Connections. *Communications of the ACM*, 42(2), February 1999.

[11] Myong H. Kang, Joon S. Park, and Judith N. Froscher. Access control mechanisms for inter-organizational workflow. In *Proceedings of the Sixth ACM Symposium on Access control models and technologies*, pages 66–74. ACM Press, 2001.

[12] John A. Miller, Mei Fan, Shengli Wu, Ismailcem B. Arpinar, Amit P. Sheth, and Krys J. Kochut. Security for the meteor workflow management system. Technical Report UGA-CS-LSDIS-TR-99-010, University of Georgia, June 1999.

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

[14] 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.

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

[16] Joon S. Park, Myong H. Kang, and Judith N. Froscher. A secure workflow system for dynamic collaboration. In *Proceedings of the 16th international conference on Information security: Trusted information*, pages 167–181, 2001.

[17] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communication, Special Issue on Copyright and Privacy Protection*, 1998.

[18] 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.

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

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

[21] Anand Tripathi, Tanvir Ahmed, Vineet Kakani, and Shremattie Jaman. Implementing Distributed Workflow Systems from XML Specifications. Technical report, Department of Computer Science, University of Minnesota, May 2000. Available at http://www.cs.umn.edu/Ajanta.

[22] Anand Tripathi, Tanvir Ahmed, and Richa Kumar. Specification of Secure Distributed Collaboration Systems. In *Proceedings to International Symposium on Autonomous Distributed Systems*, Aug 2002.

[23] Anand Tripathi, Tanvir Ahmed, Richa Kumar, and Shremattie Jaman. A Coordination Model for Secure Distributed Collaboration. In *Proceedings of Workshop on Internet Process Coordination and Ubiquitous Computing*. CRC Press, December 2001.

[24] Karin Venter and Martin S Olivier. The Delegation Authorization Model: A model for the dynamic delegation of authorization rights in a secure workflow management system. In *ISSA2002*, Muldersdrift, South Africa, 2002. Published electronically.

[25] Giovanni Vigna. *Mobile Agents and Security*. Springer, Berlin Heidelberg, 1998.

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

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