
YU-CHENG HSIAO AND GWAN-HWAN HWANG
Department of Computer Science and Information Engineering
National Taiwan Normal University, Taipei, Taiwan
swanky.hsiao@gmail.com, ghhwang@csie.ntnu.edu.tw

Secure workflow management systems (WfMSs) are required to support major security features such as authentication, confidentiality, data integrity, and nonrepudiation. The Chinese wall security model (CWSM) was designed to provide access controls that mitigate conflict of interest in commercial organizations, and is especially important for large-scale interenterprise workflow applications. This paper describes how to implement the CWSM in a WfMS. We first demonstrate situations in which an access control model is not sufficient for this if the WfMS does not keep the run-time history of data accesses and company information is mutable, and we then propose an application programming interface (API) to solve this problem, also providing support for the intrinsic dynamic access control mechanism defined in the CWSM (i.e., the dynamic binding of subjects and elements in the company data set). This API can also specify several requirements of the dynamic security policy that arise when applying the CWSM in WfMSs. Finally we discuss how to implement a run-time system to implement CWSM policies specified by this API in a WfMS.

Keywords: Workflow management system (WfMS), Chinese wall security model (CWSM), Role-based access control (RBAC).

1. INTRODUCTION

Workflow management systems (WfMSs) are software systems that support coordination and cooperation among members of an organization who are performing complex business tasks [1,2,3,4,5]. The business tasks are modeled as workflow processes that are automated by a WfMS. The workflow model (also referred to as the workflow-process definition) is the computerized representation of the business process that defines the starting and stopping conditions of the process, the activities in the process, and control and data flows among these activities. An activity is a logic step within a workflow, which includes information about the starting and stopping conditions, the users who can participate, the tools, data, and resources needed to complete the activity, and the constraints on how the activity should be completed. A person who participates in the execution of an activity is called a participant of that activity. Activities are usually organized into a directed graph that defines the order of execution among the activities in the process, where nodes and edges in the graph represent activities and control flow, respectively. A workflow-process instance is the execution state of a workflow process, and the execution of a workflow-process is controlled by the workflow engine.

A WfMS is intrinsically a network-based application. For a WfMS with a single workflow engine, the participants in activities usually communicate with the workflow engine from different locations via a network system. This requires communication
security to be maintained. Furthermore, for a distributed WfMS in which the activities of the workflow process can be executed in different workflow engines \([6,7,8]\), the process instance should be exchanged or transmitted among workflow engines via the network. Thus, a secure network-based WfMS is required to implement major security features such as authentication, confidentiality, data integrity, and nonrepudiation \([9,10]\). Confidentiality involves prohibiting unauthorized disclosure of information such as the process instances or external data of a WfMS during its execution. Therefore, a WfMS usually has an access control mechanism that the system designer can use to specify how to restrict access to authorized users.

This paper presents how to implement the CWSM in a WfMS. The CWSM, which is also called the Brewer and Nash model \([11]\), and is constructed to provide information security or access controls that can change dynamically. This model was designed to provide controls that avoid conflict of interest (COI) in commercial organizations. Data are viewed in this model as consisting of objects that belong to particular companies. Access to particular parts of data is not constrained by their attributes but rather by what data the subject already holds access rights to. Note that the access related to read, write, or read-and-write operation. The company data set is categorized into mutually disjoint COI classes, as shown in Figure 1. For example, banks, oil companies, and airline companies belong to different COI classes, and the CWSM policy prohibits information flows from one company to another that cause COI. Thus, if a subject accesses the information of bank B1, then s/he is not allowed to access information of other companies within the same COI class, such as that of bank B2 or Bi. However, the subject can access information in another COI class, such as objects of oil company O1.

![Figure 1: Examples of company information for the CWSM](image)

The CWSM proposes the following mandatory read and write rules:

- **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.
- **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 that is within a different company data set from the one for which write access is requested.
Many access control mechanisms have been proposed and implemented. Access control is traditionally implemented using an access control matrix (ACM), in which an access request from a subject can be granted if the requested access right is recorded in the matrix [12]. The role-based access control (RBAC) model is regarded as a neutral policy and has been the most popular security model in recent years [13]. Data access is restricted to authorized users in this approach, and it represents a newer alternative approach to mandatory access control and discretionary access control [12]. Roles are created for various operations within an organization, with the permissions to perform certain operations being assigned to specific roles. Subjects are assigned particular roles, through which they acquire the permissions to perform particular operations. RBAC has been applied to access control in WfMSs [14,15,16,17]. Although RBAC has been successful in various applications, we have found it to be difficult for conducting higher level access control such as the CWSM in WfMSs. When the company information used in a workflow process is static, then RBAC can implement the BN read rule. However, RBAC has problems in implementing the BN read rule if the company information is dynamic. There are at least two cases to consider. The first case is where the same workflow definition uses different company information. Although we know the structure of the company information, we need to design different codes to apply RBAC to them. The second case is where the company information may be created or updated dynamically during the execution of the workflow process. In this case the code for implementing RBAC need to bind subject to objects dynamically. A lack of knowledge of the COI class and company data set prior to executing the workflow process makes it impossible to design the code for operating RBAC in advance. In the second situation we need to apply the BN write rule during the execution of the workflow process. Because the application of the BN write rule requires the use of the object access history and the company information to decide if permission for a write operation should be granted, it is impossible to implement the BN write rule purely in RBAC without storing the object access history and the company information in the workflow engine. The main factor making RBAC difficult to implement CWSM is access permission in RBAC being controlled by the binding relationships among subjects, roles, and objects. However, access control in the CWSM depends on the object access history and company information to determine if an access request should be granted. Section 3 provides several examples to illustrate these two situations.

In this paper we show how to implement the CWSM in a WfMS. It is obvious that we should solve the problems that cannot be solved by the popular RBAC model. Our solution is to design an API that allows the system developer to specify the CWSM policy in a WfMS. The proposed API is designed to satisfy the following dynamic security requirements for the CWSM:

1. **Dynamic binding between subjects and companies:** The binding of a subject and a company can be dynamically implemented during the workflow execution. For example, according to the company information shown in Figure 1, if a subject requests access to objects of bank B1 and this access is successful, then s/he is bound to B1. This is the basic requirement that makes the information security access controls of the CWSM able to change dynamically.

2. **Dynamic company information manipulation:** The company information should be able to be created, chosen, or updated dynamically during the workflow execution. As a result, the same workflow definition can use different company information.
3. **Privileged subject management**: The binding of subjects to company information can be controlled during the execution of the workflow process. In some applications, not all data accesses of the participants in a WfMS should be controlled by the CWSM.

4. **Temporal-based enforcement**: The time or duration when the CWSM policy is applied should not be static. In some situations, we may want to set up the CWSM policy to be applied only on part of the activities in a workflow process.

In general, implementing the CWSM should depend on the dynamic behavior of the workflow process. That is, it should be possible to synchronize operations related to the CWSM policy with the progression of the workflow process. This enhances the flexibility of the CWSM because our framework synchronizes access control with the execution of WfMSs. By using the proposed API, the workflow engine creates data access history tables that store dynamic data accesses performed by subjects. Instead of specifying each data access according to the company information, the workflow engine consults the data access history tables and company information to control the data access so as to implement the CWSM security requirement.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 uses several examples to demonstrate the limitation of applying RBAC to implement the CWSM in a WfMS. Section 4 describes general access control models for WfMSs. Section 5 presents the proposed API that is used to specify the CWSM policy in a WfMS. Section 6 discusses how to implement the CWSM policy in a WfMS in the proposed API, and the implementation details are presented in Section 7. Conclusions are drawn in Section 8.

### 2. RELATED WORK

Many access control techniques have been proposed. Traditional access control models such as the static access control of the ACM is insufficient in WfMSs [12]. Olivier et al. proposed an approach for dynamically granting access rights to subjects during the execution of a workflow [18]. Knorr reported dynamic access control matrices for WfMSs that used the Petri net to model WfMSs, with access rights changing with the marking of the Petri net [19].

Thomas and Sandhu proposed modeling access controls from a task-oriented rather than a subject-object perspective [20]. Their approach constantly monitors access permissions, which are activated and deactivated in accordance with emerging context associated with progressing tasks. The approach applies task-based access control (TBAC) to WfMSs and enables the granting and revoking of permissions to be automated and coordinated with the progression of activities in a WfMS. Dong et al. also proposed an access control model based on TBAC that took two basic dynamic factors into account: the state of the authorization processes and the state of the process instances [21].

Multilevel security (MLS) has posed a challenge to the computer security community since the 1960s [22]. In MLS, security levels are assigned to subjects and objects. Higher level users must have access permission to lower level objects, and higher level objects must not leak to lower level subjects or objects. Kang et al. worked on implementing the
MLS model in a WfMS [23,24]. Wietrzyk et al. proposed an approach to security distributed workflow database management system based on MLS [25].

RBAC has been employed to implement access control mechanism in WfMSs. Atluri and Huang proposed a workflow authorization model that allows subjects to gain access to required objects during the execution of a specific task [26]. The authorization can be synchronized with the execution of the workflow process. Bertino et al. presented a language for defining constraints on role assignment and user assignment to tasks in a workflow to allow the separation of duties to be specified in a WfMS [14]. The many studies that have addressed workflow security issues have generally focused on access control and separation-of-duty issues. Park et al. addressed the security services for a secure workflow system to support dynamic collaboration for interorganizational enterprises [27]. Payne et al. proposed a solution that incorporates Napoleon – a multilayered RBAC modeling environment for distributed computing systems [28] – in a WfMS [15]. Huang and Atluri introduced a Web-enabled WfMS called SecureFlow [29], with their work showing that the security specification and enforcement could be placed on top of an existing WfMS to provide security using RBAC. Park and Sandhu described how to use role information on the Web using smart certificates [30]. Their work showed that role information can be used to authorize Web-based transactions between a client and a Web server. Ahn et al. defined a simplified RBAC model to describe the security architecture to be applied to an existing Web-based WfMS [31]. Basin et al. combined unified modeling language (UML) and RBAC to protect process components [32]. They showed how to integrate their security modeling language SecureUML with UML process models. SecureUML is a UML-based language for modeling access control requirements that generalizes RBAC. Park and Hwang proposed an approach that supports RBAC services for collaborative enterprise in peer-to-peer computing environments, where the access control information is interpreted by a middleware among peers [33]. Chou et al. proposed an access control model for WfMSs named WfACL [17]. They focused on dynamically specifying role-subject and role-permission binding. In their paper they used the terms “dynamic role change” and “role association change”. WfACL should be embedded in a workflow to control access when a workflow instance is executing.

Some researchers have studied the use of cryptography to secure WfMSs. The Meteor workflow system utilizes encryption algorithms, digital signatures, and access control, in which workflows are statically prepartitioned in the central server in a distributed CORBA-based system [34,35]. Hwang et al. proposed a WfMS that operates in a purely distributed manner without needing a centralized workflow engine [36]. It is an XML-based document-routing system that implements major required security features such as authentication, confidentiality, data integrity, and nonrepudiation based on cryptographic algorithms in distributed or large-scale network environments. Previous work that has introduced the ACM, MLS, TBAC, RBAC, or cryptography to secure WfMSs has not addressed the goal of the CWSM; that is, avoidance of COI.

One of the proposed access control models is called attribute-based access control (ABAC) where the central idea asserts that access can be determined based on various attributes presented by a subject [37]. UCON is a kind of ABAC model where authorizations are predicates defined on subject and object attributes, while conditions are environmental restrictions represented by system attributes, such as time, location, load, etc [38]. Authorizations and conditions are enforced not only when a subject generates an
access request, but also during the whole ongoing stage of the usage session. As the side-effects of the usage, subject and object attributes can be updated; this is referred to as attribute mutability in UCON. Kuhn et al. summarized access control schemes which employ attributes to determine the permission of accesses into several models including ABAC-basic, ABAC-RBAC hybrid, ABAC-ID, RBAC-A (dynamic roles), RBAC-A (attribute-centric), and RBAC-A (role-centric) [39]. Yao et al. proposed the RCBAC model which extends the RBAC with context constraints [40]. The RCBAC mechanisms dynamically grant and adapt permissions to users based on a set of contextual information collected from the grid environments. Strembeck and Neumann presented an approach that uses special purpose RBAC constraints to base certain access control decisions on context information [41]. A context constraint is defined as a dynamic RBAC constraint that checks the actual values of one or more contextual attributes for predefined conditions. ABAC and context-based RBAC provide dynamic characteristic. If we want to implement CWSM in ABAC or context-based RBAC, the run-time history of data accesses and company information should be represented as attributes or context in the access control system and the access control system should be able to encode BN read and write rules.

There has been little work on implementing the CWSM in WfMSs. Sandhu proposed to employ the lattice model to implement the CWSM [42]. Lattice labels which are actually access arrays and are used to record the access history need to be associated to all the subjects and objects. He thought that the BN model is too restrictive and did not discuss how to enforce the BN write rule in the lattice model. Also, there is no discussion about how to apply the CWSM in WfMS. Atluri et al. extended the CWSM for decentralized workflow execution [43]. They impose access restrictions on sensitive data by restrictive partitioning, which ensures that no two task agents of the same COI class are in one partition. However, their partitioning scheme requires the binding of subjects and companies to be static. The preliminary result of this research is published in [44]. We extended it in this paper including related work, more examples and discussion, and general access control model.

3. LIMITATION IN APPLYING RBAC TO IMPLEMENT THE CWSM

In the section we present several examples to illustrate the limitations and difficulties of applying RBAC to implement the CWSM in WfMSs. The RBAC model is regarded as a neutral policy and has been the most popular security model in recent years. We demonstrate that it is impossible to implement the CWSM without storing the object access history and the company information in the WfMS.

Figure 2 provides the first example, showing two sets of company information (CI1 and CI2) and a portion of a workflow definition of a business process. Four subjects (John, Mary, Ken, and Leo) are involved in this workflow, and A, B, ..., I are activities. The terms “subject” and “participant” are used interchangeably in this paper, since the latter is a formal term in a WfMS and the former is widely used in access control models such as RBAC – together they represent persons or identities that access data and perform tasks. Assume that we need to apply the CWSM based on company information CI1. Leo is the administrator of John, Mary, and Ken, and is responsible for distributing work to these
three persons in activity A. John, Mary, and Ken need to process company data and review the specification of merchandise for one company in CI1. Note that they only perform read operations to objects, and hence only the BN read rule needs to be enforced in advance. Leo assigns three companies C1, C2, and C3 exclusively to John, Mary, and Ken. For example, if Leo assigns company C1 to John, then John is responsible for the data processing of that company and reviewing the specification of merchandise for company C1 in activities B and F, respectively. With the BN read rule, if company C1 is assigned to John, then John is prohibited from accessing the individual objects owned by companies C2 and C3. In order to implement this security requirement in RBAC, we create three roles R_C1, R_C2, and R_C3 that have been granted the permissions needed to access the individual objects of companies C1, C2, and C3, respectively. After executing activity A, we should assign roles to John, Mary, and Ken according to the decisions made by Leo. The assignment of roles can be represented by the pseudo code shown in Figure 3A. We assume that the access control mechanism of a WfMS is capable of performing dynamic role-subject and role-permission binding [17]. ASSIGN_ROLE(Subject_name, Role_name) involves assigning a subject with a specific role, and GRANT_PERMISSION(Set_of_objects, Role_name) grants a role with the permission to perform accesses to a set of objects. Note that the code should be executed after activity A. However, it is necessary to change the code if the same workflow process is to be executed with different company information. For example, if we want to apply the information of another company, such as the CI2 shown in Figure 2, the role assignment must be changed as shown in Figure 3B. Unavoidably, we need to modify the code that control the operation of RBAC.

Figure 2: The first motivation example

This motivation example demonstrates that the RBAC model has to change the role assignment rules when the company information is altered. However, if the company information is generated dynamically during the execution of the workflow process, it is
much more difficult to implement the CWSM in RBAC. We use another motivation example to demonstrate this situation.

```c
/*RBAC operations which should be executed in activity A*/
GRANT_PERMISSION (C1's objects,ROLE R_C1);
GRANT_PERMISSION (C2's objects,ROLE R_C2);
GRANT_PERMISSION (C3's objects,ROLE R_C3);

/*Assign a role to John*/
IF (John is assigned to handle company C1)
ASSIGN_ROLE(John,R_C1);
ELSE IF (John is assigned to handle company C2)
ASSIGN_ROLE(John,R_C2);
ELSE IF (John is assigned to handle company C3)
ASSIGN_ROLE(John,R_C3);

/*RBAC operations which should be executed in activity A*/
GRANT_PERMISSION (D1's objects,ROLE R_D1);
GRANT_PERMISSION (D2's objects,ROLE R_D2);
GRANT_PERMISSION (D3's objects,ROLE R_D3);

/*Assign a role to John*/
IF (John is assigned to handle company D1)
ASSIGN_ROLE(John,R_D1);
ELSE IF (John is assigned to handle company D2)
ASSIGN_ROLE(John,R_D2);
ELSE IF (John is assigned to handle company D3)
ASSIGN_ROLE(John,R_D3);

/*Assign a role to Mary*/
IF (Mary is assigned to handle company D1)
ASSIGN_ROLE(Mary,R_D1);
ELSE IF (Mary is assigned to handle company D2)
ASSIGN_ROLE(Mary,R_D2);
ELSE IF (Mary is assigned to handle company D3)
ASSIGN_ROLE(Mary,R_D3);

/*Assign a role to Ken*/
IF (Ken is assigned to handle company D1)
ASSIGN_ROLE(Ken,R_D1);
ELSE IF (Ken is assigned to handle company D2)
ASSIGN_ROLE(Ken,R_D2);
ELSE IF (Ken is assigned to handle company D3)
ASSIGN_ROLE(Ken,R_D3);
```

(A) (B)

Figure 3: Pseudo code of RBAC

The second motivation example is an extension of the first example. Referring to Figure 4, in activity A’ Leo does not perform work distribution as in activity A of the previous example, instead only preparing the related information of the three companies, which means that the company information is dynamically created in activity A’. The preparation or creation of company information is unpredictable since it may depend on the results of previously executed activities. Activities B’, C’, and D’ are three parallel activities. John, Mary, and Ken choose one of the three companies prepared by Leo in A’ in a first-come-first-served manner. For example, if John made the choice first, he can choose among the available three companies. Also, he would be responsible for reviewing the specification of merchandise in activity F of the company he chose. The example shows a scenario involving commercial security policy, which is a very popular application of the CWSM [45]. For a set of companies that partially compete with each other and a group of consultants, a consultant is not allowed to work for a company if he has insider knowledge of a competitor. The goal of the CWSM is to prevent information flows from competing companies to the same consultant. Figure 5 shows the pseudo code of the RBAC operations that implement the security requirement of the example. The code looks similar to that in Figure 3, which implements the security requirement of the first motivation example. However, the code is divided into four parts and distributed separately to activities A’, B’, C’, and D’, which makes them much more difficult to maintain than the code shown in Figure 3.
However, the premise that makes the code work is that the number of companies prepared in activity $A'$ is fixed so that we can have a fixed pattern for performing role-subject and role-permission binding. For example, the code shown in Figure 5 can only handle information when there are exactly three companies in the company data set. If the number of companies prepared in activity $A'$ is decided dynamically, a more flexible way is needed to specify access control in RBAC since we cannot use a fixed number of roles to set up the required access control. Thus, if RBAC in the system does not support the dynamic creation of roles, it is impossible to implement this type of security requirement.

```c
/*RBAC operations which should be executed after activity $A''$*/
GRANT_PERMISSION (objects of company one, R_COMP1);
GRANT_PERMISSION (objects of company two, R_COMP2);
GRANT_PERMISSION (objects of company three, R_COMP3);

/*RBAC operations which should be executed in activity $B''$*/
IF (John chose to handle company one) ASSIGN_ROLE(John,R_COMP1);
ELSE IF (John chose to handle company two) ASSIGN_ROLE(John,R_COMP2);
ELSE IF (John chose to handle company three) ASSIGN_ROLE(John,R_COMP3);

/*RBAC operations which should be executed in activity $C''$*/
IF (Mary chose to handle company one) ASSIGN_ROLE(Mary,R_COMP1);
ELSE IF (Mary chose to handle company two) ASSIGN_ROLE(Mary,R_COMP2);
ELSE IF (Mary chose to handle company three) ASSIGN_ROLE(Mary,R_COMP3);

/*RBAC operations which should be executed in activity $D''$*/
IF (Ken chose to handle company one) ASSIGN_ROLE(Ken,R_COMP1);
ELSE IF (Ken chose to handle company two) ASSIGN_ROLE(Ken,R_COMP2);
ELSE IF (Ken chose to handle company three) ASSIGN_ROLE(Ken,R_COMP3);
```

Figure 4: The second motivation example

Figure 5: Pseudo code of RBAC
The above two motivation examples show that RBAC has problems when the CWSM policy is dynamic. For example, the first one shows that the company information can change and the second one shows that entities in the company data set can only be decided during the run-time of the WfMS. The two examples only need to apply the BN read rule. The BN write rule prevents information leakage by Trojan Horses. For example, referring to the company information in Figure 1, suppose John has read access to objects of bank B1 and oil company O1, and Mary has read access to objects of bank B2 and oil company O1. If John is allowed write access to objects of O1, a Trojan-Horse-infected subject executing with John’s privileges can transfer information from objects of B1 to those of O1, which can be read by Mary, who then can read information about banks B1 and B2. Apparently, run-time history is also necessary in applying BN write rules.

It is obvious that if a WfMS wants to comply with the BN read and write rules for data accesses, it must maintain the run-time history of data accesses and the company information. Thus, the requirements of the BN read and write rule cannot be implemented purely by applying RBAC, context-based RBAC, ABAC, or UCON without referring to the history of data accesses and the company information. Even if the WfMS is able to access the history of data accesses and the company information, its access control system should determine if a request should be granted according to BN read and write rules. If we apply the RBAC to make access control, we should insert code in some activities which can dynamically change the binding relationship between subjects, roles, and objects according to the BN read and write rules. If we embed code which encode the BN read and write rules to bind the required relationship, this tight coupling between the execution code in activities and its security policy increases the cost of implementing and maintaining the WfMS.

4. GENERAL ACCESS CONTROL MODEL IN WFMS

Before we present how to implement the CWSM in a WfMS, we first illustrate the general access control model of WfMSs. Referring to Figure 6, the participant of the activity usually communicates with the executed activity via a user interface. In an engine-based WfMS, the execution of activities and the user interface are controlled by the workflow engine. Some of the data that the participant needs to access are stored in the process instance of the workflow that the activity belongs to. However, not all the data objects referenced by the participants can be stored in the process instance. For example, some large paper documents are intrinsically difficult or not necessary to be stored in the process instance. The workflow engine accesses these data using the access controller. Usually not all the data can be accessed by a participant. The access controller retrieves data from or updates data in a process instance according to the security policy specified by the workflow designer. Note that we do not define the type of security policy here as it could be any type, such as the ACM, RBAC, TBAC, or the CWSM.
We can classify the data stored in the process instance into several groups:

- **Static intraprocess data object**: Static data objects, such as strings, images, or files, will be shown to the participants during the execution of the activities in a workflow process. They exist when the process is started.

- **Dynamic intraprocess data object**: This type of data object is generated during the execution of an activity. Usually the participant needs to embed messages in the process instance after the execution of an activity terminates.

- **Static and dynamic external data objects**: An external data object is static if it is created before the workflow process execution; otherwise it is a dynamic external data object.

According to how a WfMS shows the data objects to the participant, we can distinguish two data access models in the execution of the workflow process. The **passive data access model** is where the workflow engine shows the objects only when the execution code in the activity access a specific data object. Similarly, in the **active data access model**, the workflow engine displays all the data objects that the participant can access when an activity has started.

### 5. API TO SPECIFY THE SECURITY POLICY FOR THE CWSM IN WfMS

As discussed in Section 3, specifying a CWSM policy in RBAC is complicated by the need to set up sophisticated role assignments based on the company information and subjects that are required to be involved in the security framework. A convenient way to solve this problem would be to control the read/write access to objects issued by all the subjects involved in a WfMS according to all the available company information automatically. However, this method does not work in a WfMS. In the motivation example shown in Figure 2, Leo is responsible for the work distribution in activity A. It is obvious that Leo should be able to access objects of all companies to perform his task in this business process. If the CWSM of company information CI1 is applied to Leo, then he will not be able to perform his task in activity A. It is obvious that Leo needs to refer to data
objects of all three companies to distribute work related to the three companies to John, Mary, and Ken. The examples shown in Section 3 provide the motivation to design an API to specify security policy based on the CWSM for WfMSs. First of all, we should be able to bind the subjects to certain company information, which is a basic operation in the CWSM. We should then be able to control when the CWSM associated with the binding starts or ceases. Also, applying the CWSM can be synchronized with the progress or the dynamic behavior of the WfMS. We call this API the “Chinese Wall Security Policy Specification API for WfMS” (CWSPS API).

We use the Backus Naur Form [46] to show the syntax of the CWSPS API (see Figure 7A). Note that all the nonterminal symbols in the figure are underscored.

There are several types of commands in the CWSPS API:

- **LoadCompanyInformation** (line 5): Company information that will be used should first be loaded into the workflow engine. Since company information can be represented by a tree data structure, we propose storing the company information in an XML document since such a document intrinsically has a tree data structure [47]. An example of an XML document that represents company information is shown in Figure 7B. Line 1 in Figure 7C shows an example of a LoadCompanyInformation statement. Note that variable CI1 now represents an entity of the company information.

- **Subject and company information binding** (line 6): This statement binds a group of subjects to a set of company information. The binding is assigned to a variable that is subsequently used to start or cease the associated CWSM policy; see line 3 in Figure 7C.

- **Subject and company information ignore** (line 8): We assume that all the accesses to objects should be controlled by the workflow engine. However, if the designer wants some subjects not to be controlled by the CWSM, s/he can set these subjects to be able to perform read and write operations to objects contained in certain sets of company information without limitation.

- **EnforceCWSM** (line 10): This starts a CWSM policy according to a binding defined in a Subject and company information binding or Subject and company information ignore statement.

- **CeaseCWSM** (line 11): This stops the execution of a CWSM policy according to a binding defined in a Subject and company information binding or Subject and company information ignore statement.

- **TouchR** (line 12): This command has two parameters, **SubjectName** and **CompanyName**, and sets the status of individual objects of company **CompanyName** as having been read by subject **SubjectName**.

- **TouchRW** (line 13): This command is very similar to **TouchR**, setting the status of individual objects of company **CompanyName** as having been read and written by subject **SubjectName**.

- **CheckR** (line 14): This command has two parameters, **SubjectName** and **CompanyName**, and is used to check if subject **SubjectName** should be granted permission to read objects of company **CompanyName**.

- **CheckRW** (line 15): This is used to check if subject **SubjectName** should be granted permission to read and write objects of company **CompanyName**.
1. LoadCompanyInformation | Subject and company information binding |
2. Subject and company information ignore | EnforceCWSM | CeaseCWSM |
3. LoadCompanyInformation → CompanyInformationName = LoadCompanyInformation(URL_Address)
4. Subject and company information binding →
5. Var = CWSM (CompanyInformation (CompanyInformationName), Subject (SubjectName, SubjectName))
6. Subject and company information ignore →
7. Var = CWSMIgnore (CompanyInformation (CompanyInformationName), Subject (SubjectName, SubjectName));
8. EnforceCWSM
9. CeaseCWSM
10. TouchR
11. TouchRW
12. CheckR
13. CheckRW
14. CompanyInformationName → StringWithoutWhitespace
15. SubjectName → StringWithoutWhitespace
16. Var → StringWithoutWhitespace
17. CompanyName → StringWithoutWhitespace
18. URL_Address → StringWithoutWhitespace
19. <!--CWSPS API statements executed in activity A--> 
   1. CI = LoadCompanyInformation ("http://www.csie.ntnu.edu.tw/~iclab/CI1.xml");
   2. CI2 = LoadCompanyInformation ("http://www.csie.ntnu.edu.tw/~iclab/CI2.xml");
   3. CI = CWSM (CompanyInformation (CI1, CI2), Subject (John, Mary, Ken));
   4. Enforce (CI);
   5. /*CWSPS API statements executed in activity Z*/ 
   6. Cease (ignore);
   7. Cease (binding);
The CWSPS API statements are used to set up the CWSM policy of a workflow process. We assume that the scripts written in the CWSPS API are embedded in the execution code of activities, so that access control based on the CWSM can be synchronized with the progress of the workflow process. This means that these scripts may be executed during the execution of workflow processes. The statements “b1=CWSM(CompanyInformation(CI1), Subject(John, Mary, Ken)); Enforce (b1);” specify enforcing the BN read and write rules to subjects John, Mary, and Ken according to company information CI1 as shown in Figure 2. We can easily change the associated company information to CI2 using the statements “b2=CWSM(CompanyInformation(CI2), Subject(John, Mary, Ken)); Enforce (b2);”. Also, we can specify to apply multiple sets of company information to the same group of subjects as shown in Figure 7C. The CWSPS API code in Figure 7D show how to specify the security policy for the first motivation example shown in Section 3. The statement in line 1 first retrieves the company information used in the workflow process. We assume that the routine “CompanyInformationProcessedByWfProcess()” generates the information of a company. Second, lines 2 and 3 use the command “CWSMIgnore()” to instruct the system to grant permission for any access requests of objects that are included in company information CI to Leo. Leo then performs work distribution in line 4. Lines 5 and 6 specify the data accesses of three subjects to be controlled by the company information CI. Finally, lines 7, 8, and 9 set up the statuses of individual objects in the company information according to the work distribution performed by Leo.

The CWSPS API code in Figure 7E show how to specify the security policy for the second motivation example described in Section 3. First, in line 1 Leo prepares the company information for several companies, and lines 2 to 6 specify the appropriate security policy between subjects Leo, John, Mary, and Ken, and company information CI. The above statements are executed in activity A’. The code that need to be executed in activities B’, C’, and D’ are identical. Line 7 first locks a semaphore S to ensure that lines 8 and 9 are a critical section. Line 8 selects a company for the subsequent processing, lines 9 grants the subject access rights for objects of the chosen company, and line 10 unlocks the semaphore. Finally, the actual company data process can be performed. Note that the correctness of the code does not depend on the number of companies in the dynamically created company information.

We now describe how the CWSPS API satisfies the four dynamic security requirements mentioned in Section 1. First, the TouchR and TouchRW statements can be used for
requirement 1. When the workflow process decides to bind a subject to an element in the company data set, the execution of \texttt{TouchR} or \texttt{TouchRW} achieves the binding. Requirement 2 can be satisfied by using the \texttt{LoadCompanyInformation} statement. The workflow process can prepare the company information in some activities and then execute this statement to load the information of a company to start CWSM-based access control. It is obvious that the workflow process can choose among several sets of company information before it starts access control. Requirement 3 can be met by the \texttt{Subject and company information binding} and \texttt{Subject and company information ignore} statements. As long as the two statements can be embedded in the execution code of the activities, we can dynamically set up the access of objects contained in the company information by the specified subjects to be controlled or ignored. Finally, requirement 4 is satisfied by \texttt{EnforceCWSM} and \texttt{CeaseCWSM} statements, since it can control when to start or cease the CWSM-based access control specified by the programmer. The CWSPS API statements are generally designed to be embedded in the execution code of the workflow process. Thus, we can manipulate CWSM-based access control according to the dynamic behavior of the workflow process.

6. RUN-TIME SYSTEM FOR THE CWSPS API AND IMPLEMENTATION

In this section we present the proposed architecture of the run-time system that can implement the security policy specified in the CWSPS API. Figure 8 shows the proposed architecture. The subject accesses objects via the workflow engine. When the workflow engine receives a request from the subject to access an object, it checks the company information as well as the history tables according to the BN read and write rules to decide if the request should be granted. The history tables may need to be updated after objects are accessed by subjects. In general, if the required access control is not related to the CWSM policy, we can employ RBAC to control data accesses during the execution of the workflow process. However, if the accesses of any data object need to be controlled according to the BN read and write rules, we first establish the company information including the COI class and company data set. These data objects are then allocated to the companies in the company data set that own them. Finally, during the run-time of the workflow process, the workflow engine can decide if a request from a subject to access some object should be granted according to the BN read and write rules.

![Figure 8: Architecture for supporting the execution of the CWSPS API in a WfMS](image-url)
Whilst the company information is built during the run-time of the workflow process, it must be ready before we start enforcing the CWSM policy. Each data object that can be accessed during the workflow execution should be attached to at least one company information if its access control is to follow the BN read and write rules. We first classify companies into different COI classes. We identify which company each static intraprocess and external data object belongs to, and designate it as an object of that company. Objects that are created during the execution of the workflow process need to be added to the company information that owns them.

Each item of used company information has a corresponding history table, which is a two-dimensional array created during the execution of the workflow process. Actually, a history table can be defined as a projection function, $HT$, that maps a subject and a company into the corresponding access history and is designated as $HT: S \times C \rightarrow Q$, where $S$ is the set of subjects, $C$ is the set of companies, and $Q = \{R, RW, \emptyset, I\}$. $HT(s,c)$ is defined as follows:

- $HT(s,c)=R$ if subject $s$ has read the objects of company $c$.
- $HT(s,c)=RW$ if subject $s$ has read and written the objects of company $c$.
- $HT(s,c)=\emptyset$ if subject $s$ has neither read nor written the objects of company $c$.
- $HT(s,c)=I$ if we do not have to control the access of subject $s$ to objects of company $c$.

Whenever the system executes a `LoadCompanyInformation` command, the access controller creates history tables for the loaded company information. In the execution of the `Subject and company information binding` command, the access controller records the binding relationship between subjects and company information. The `EnforceCWSM` command will add subjects to history tables. In the case where a subject $s$ is supposed to be bound to a specific company information, the `EnforceCWSM` command will add a row for $s$ to the corresponding history table of the information of that company, with elements equal to $\emptyset$. The `CeaseCWSM` command will delete the rows of subjects from history tables. The processing of the `Subject and company information ignore` command is very similar to that of the `Subject and company information binding` command. However, we initialize the elements of history tables to be $I$ in the subsequent `EnforceCWSM` command.

For the statements “\(b1=\text{CWSM}(\text{CompanyInformation}(CI1,CI2), \text{Subject}(John, Mary, Ken))\); `Enforce (b);’”, subjects John, Mary and Ken are set to apply the CWSM to two sets of company information CI1 and CI2. Since CI1 has three companies (C1, C2, and C3) with the COI class “Bank”, we create a history table as shown in Figure 9A. Similarly, we create other history tables for the binding between CI2 and subject groups John, Mary, and Ken (see Figure 9B). The two commands “\(b2=\text{CWSMIgnore}(\text{CompanyInformation}(CI1), \text{Subject}(Leo))\); `Enforce (b2);’” will add a row for Leo to the history table of CI1, as shown in Figure 9C. Figure 9D shows the situation where John, Mary, and Ken have read and written to the objects of companies C1, C2, and C3, respectively.
The history table clearly indicates whether a subject has ever accessed objects of a company in the company data set. The workflow engine is responsible for maintaining the coherence and the correctness of the history table. Before each operation (read or write) of objects that belong to a company in the company data set, the workflow engine needs to look up an appropriate history table to check if it is allowed. According to the BN read and write rules, the engine should deny this access if the subject had ever accessed the objects of another company in the same company data set. The “Check_Read” and “Check_Write” algorithms in Figure 10 allow the workflow engine to verify if the access violates the BN read and write rules, respectively, by implementing the CheckR and CheckRW commands. Also, the workflow engine may need to update some history tables after accessing an object. The “Touch_Read” and “Touch_ReadWrite” algorithms correspond to the commands TouchR or TouchRW, respectively.

**Algorithm Check_Read(s,o):** Check if a subject $s$ has the right to read an object $o$ owned by a company.

**Input:** A subject $s$ and an object $o$.

**Output:** Returns True if $s$ has the right to read $o$.

Let $c$ be a company that both owns object $o$ and is in company information CI.

**IF** (The corresponding history table of CI is not created) **THEN**

RETURN False.

**ELSE**

Let $HT$ be the corresponding history table of CI.

**IF** ($HT(s,c)=="I","R","RW") **THEN**

RETURN True.

**END IF**

**IF** ($HT(s,c')="R","RW") **THEN**

RETURN False.

**END IF**

**END FOR**

RETURN True.
Algorithm Check_Write(s,o): Check if a subject s has the right to write to an object o owned by a company.
Input: A subject s and an object o.
Output: Returns True if s has the right to write to o.
Let c be a company that both owns o and is in company information CI.
IF (The corresponding history table of CI is not created) THEN
    RETURN False.
ELSE
    Let HT be the corresponding history table of CI.
END IF
IF (HT(s,c)=="I") THEN
    RETURN True.
END IF
IF (Check_READ(s,o) returns False) THEN
    RETURN False.
END IF
Let Γ be the set of companies that belongs to all the COI classes except that of c.
FOR each element c’ in Γ
    IF (HT(s,c’)=="R" or "RW") THEN
        RETURN False.
    END IF
END FOR
RETURN True.

Algorithm Touch_Read(s,o): Set subject s to have previously read an object o owned by a company.
Input: A subject s and an object o.
Output: Returns True if s has successfully updated the history table, otherwise it returns False.
Let c be a company that both owns o and is in company information CI. Let HT be the corresponding
history table of CI.
IF (Check_READ(s,o) returns True) THEN
    Update HT(s,c) to “R”.
    Return True.
ELSE
    RETURN False.
END IF

Algorithm Touch_ReadWrite(s,o): Set subject s to have previously read and written to an object o owned by
a company.
Input: A subject s and an object o.
Output: Returns True if s has successfully updated the history table, otherwise it returns False.
Let c be a company that both owns o and is in company information CI. Let HT be the corresponding
history table of CI.
IF (Check_Write(s,o) returns True) THEN
    Update HT(s,c) to “RW”.
    Return True.
ELSE
    RETURN False.
END IF

Figure 10: Algorithms for the CWSM

We can employ the algorithms presented in Figure 10 to implement the passive and active data access model described in Section 4. The “Perform_Read” and “Perform_Write” algorithms shown in Figure 11 perform passive read and write operations on data objects. The “List_Accessable_Objects” algorithm is the skeleton to perform active data access for all the objects.

Algorithm Perform_Read(s,o): Read an object o owned by a company by subject s.
Input: A subject s and an object o.
Output: Returns the value of object $o$ if $s$ has successfully read object $o$, otherwise it returns False.

Let $c$ be a company that both owns object $o$ and is in company information CI. Let HT be the corresponding history table of CI.

IF (Check_Read($s$, $o$) returns True) THEN
    Update HT($s$, $c$) to “R”.
    Return the value of object $o$.
ELSE
    RETURN False.
END IF

Algorithm Perform_Write($s$, $o$, $v$):
Write $v$ to an object $o$ owned by a company by subject $s$.

Input: A subject $s$, an object $o$, and a value $v$ to be written to $o$.
Output: Returns True if $s$ has successfully written to object $o$, otherwise it returns False.

Let $c$ be a company that both owns object $o$ and is in company information CI. Let HT be the corresponding history table of CI.

IF (Check_Write($s$, $o$) returns True) THEN
    Write $v$ to $o$.
    Update HT($s$, $c$) to “RW”.
    Return True.
ELSE
    RETURN False.
END IF

Algorithm List_Accessable_Objects($s$):
List all the objects that can be accessed by subject $s$.

Input: A subject $s$.
Output: Returns the names of objects and their permissions.

FOR each object $o$ in the system
    IF (Check_Write($s$, $o$) returns True) THEN
        Show that subject $s$ has read and write permission for object $o$ and display $o$.
    ELSE
        IF (Check_Read($s$, $o$) returns True) THEN
            Show that subject $s$ has read permission for object $o$ and display $o$.
        ELSE
            Show that subject $s$ has no access permission for object $o$.
        END IF
    END IF
END FOR

Figure 11: Algorithms for performing passive and active data access in the CWSPS API

7. AN IMPLEMENTATION EXAMPLE

To demonstrate the feasibility of the proposed run-time system for the CWSPS API, we implemented the run-time system and tested it in two Java-based WfMSs [48,49], with all the execution code implemented in the Java programming language and each CWSPS API command implemented as a Java method. The class diagram defined in UML is shown in Figure A1 in Appendix A. The source code of the corresponding Java programs can be downloaded at http://www.csie.ntnu.edu.tw/~ghhwang/CWSPSL_Java.zip. We call this software package CWSPSL_Java. Figure 12 shows an example of invoking the CWSPSL_Java package in Java.

```java
// create company information
CompanyInformation oil = new CompanyInformation("CI_1");
CoiClass bankClass = new CoiClass("Bank");
CoiClass oilClass = new CoiClass("Oil");
Company bank1 = new Company(bankClass, "B1");
```
Company bank2 = new Company(bankClass, "B2");
Company oil1 = new Company(oilClass, "O1");
DataObject b1_Obj = new DataObject(bank1, "B1_Data");
c11.addDataObject(b1_Obj);
DataObject b2_Obj = new DataObject(bank2, "B2_Data");
c11.addDataObject(b2_Obj);
DataObject o1_Obj = new DataObject(oil1, "O1_Data");
c11.addDataObject(o1_Obj);
// enforce
String ci1File = "ci1.xml";
String ci2File = "ci2.xml";
CompanyInformation ci1 = ChineseWall.loadCompanyInformation(ci1File);
CompanyInformation ci2 = ChineseWall.loadCompanyInformation(ci2File);
Set<CompanyInformation> cis = new HashSet<CompanyInformation>();
cis.add(ci1);
cis.add(ci2);
Set<Subject> subjects = new HashSet<Subject>();
Subject john = new Subject("John");
Subject mary = new Subject("Mary");
subjects.add(john);
subjects.add(mary);
CwsmBinding bind1 = ChineseWall.cwsm(cis, subjects);
ChineseWall.enforce(bind1);
Set<Subject> ignoreSubjects = new HashSet<Subject>();
ignoreSubjects.add(john);
CwsmBinding bind2 = ChineseWall.ignore(bind1, null, ignoreSubjects);
ChineseWall.enforce(bind2);

Figure 12: An example of a Java program that invokes the CWSPS API

Our preliminary implementation revealed that the CWSPS API and the run-time system proposed in this paper can dramatically reduce the effort of specifying CWSM-related security policy. It also makes the debugging process much easier since the CWSPS API code are always concise and generic. It is obvious that the code which encode the BN read and write rules are more complex than CWSPS API code and the former code are usually more difficult to debug when we mix them with others.

8. CONCLUSION AND FUTURE WORK

In this paper we have described how to implement the CWSM in a WfMS. The presented examples demonstrate that RBAC with dynamic role-subject and role-permission binding can be applied to implement the CWSM in certain situations. However, we also show that RBAC cannot handle general cases of the CWSM – the BN read rule can be simulated in RBAC with some limitations, whereas the BN write rule cannot be implemented in RBAC. We propose an API that the system developer can use to specify the CWSM. Access control based on the CWSM can be integrated with the dynamic behavior of the workflow process. In addition to supporting the intrinsic dynamic access control mechanism defined in the CWSM (i.e., the dynamic binding of subjects and elements in the company data set), this API can specify several requirements of the dynamic security policy that arise when we want to apply the CWSM in WfMSs. The implementation and experiment demonstrate the feasibility of the proposed scheme.
REFERENCES


40. Hanbing Yao, Heping Hu, Baohua Huang, and Ruixuan Li, Dynamic Role and Context-Based Access Control for Grid Applications, Proceeding PDCAT '05 Proceedings of the Sixth International Conference on Parallel and Distributed Computing, 2005.


Subject
name: String
belongCompany: Company
roles: Set<Role>

Role
name: String
permissions: Set<Permission>
+getName(): String
+getPermissions(): Set<Permission>
+addPermission(Permission)
+removePermission(Permission)

Permission
resource: Resource
operation: Operation
getResource(): Resource
getOperation(): Operation

Operation
+READ
+READ_WRITE

Resource
getAccessRoles(): Set<Role>
addAccessRole(Role): boolean
removeAccessRole(Role): boolean

Role
name: String
permissions: Set<Permission>
+getName(): String
+getPermissions(): Set<Permission>
+addPermission(Permission)
+removePermission(Permission)

CompanyInformation
name: String
coiClasses: Set<CoiClass>
+addDataObject(DataObject)
+getCoiClass(): Set<CoiClass>

CoiClass
name: String
belongCI: CompanyInformation
companies: Set<Company>
+belongCompanyInformation(): CompanyInformation
+getCompanies(): Set<Company>
+getDataObjects(Company): Set<DataObject>
+getAllDataObjects(): Set<DataObject>

Company
name: String
belongCI: CompanyInformation
dataObjects: Set<DataObject>
+belongCompanyInformation(): CompanyInformation
+getCompanies(): Set<Company>
+getDataObjects(Company): Set<DataObject>
+getAllDataObjects(): Set<DataObject>

DataObject
belongCI: CompanyInformation
belongCOI: CoiClass
belongCompany: Company
data: String
+getData(): String
+setData(String)
+belongCompanyInformation(): CompanyInformation
+belongCoiClass(): CoiClass
+belongCompany(): Company

CwsmBinging
subjects: Set<Subject>
companies: Set<Company>
historyTable: Map<HistoryTableKey, HistoryTableRecord>
ignoreSubjects: Set<Subject>
ignoreCompanies: Set<Company>
+getHistoryTableRecordBySubject(Subject): List<HistoryTableRecord>
+getHistoryTableRecordByCompany(Company): List<HistoryTableRecord>
+updateHistoryTableRecord(Subject, Company, Operation)
+getCompanyInformations(): Set<CompanyInformation>
+getSubjects(): Set<Subject>
+getCompanies(): Set<Company>
+getIgnoreSubjects(): Set<Subject>
+getIgnoreCompanies(): Set<Company>

ChineseWall
+checkRead(CwsmBinding, Subject, DataObject): boolean
+performRead(CwsmBinding, Subject, DataObject)
+checkWrite(CwsmBinding, Subject, DataObject): boolean
+performWrite(CwsmBinding, Subject, DataObject)
+loadCompanyInformation(String url): CompanyInformation
+cease(CwsmBinding)
+ignore(CwsmBinding, Set<Company>, Set<Subject>): CwsmBinding
+enforce(CwsmBinding)
+touchRead(CwsmBinding, Subject, DataObject)
+touchWrite(CwsmBinding, Subject, DataObject)

RBAC
+assignRole(Subject, Role)
+grantPermission(Set<Resource>, Role, Operation)

Figure A1: Class diagram of the CWSPS API implemented in Java