Napoleon: A Recipe for Workflow

C. Payne, D. Thomsen, J. Bogle and R. O'Brien
Secure Computing Corporation
2675 Long Lake Road
Roseville, MN 55113

Abstract

This paper argues that Napoleon, a flexible, role-based access control (RBAC) modeling environment, is also a practical solution for enforcing business process control, or workflow, policies. Napoleon provides two important benefits for workflow: simplified policy management and support for heterogeneous, distributed systems. We discuss our strategy for modeling workflow in Napoleon, and we present an architecture that incorporates Napoleon into a workflow management system.

1. Introduction

Last year at this conference we introduced Napoleon, a multi-layered, role-based access control (RBAC) modeling environment for distributed computing systems [10]. Napoleon's primary objective is to simplify policy management for the system administrators of distributed computing systems. It satisfies this objective by shifting the burden of policy management from the system administrator alone to all of the principals involved in the system's development, including application designers, system integrators and the like. Napoleon translates the resulting policy for each enforcement mechanism in the system. Napoleon is a practical solution for RBAC policy management.

This year we argue that Napoleon is also a practical solution for business process control, or workflow, policy management.[1] Napoleon addresses two challenges posed to workflow technology developers: simplify policy management [1] and support distributed computing systems [3]. Napoleon's layered model simplifies policy management by dividing the burden among all principals in the system's development, and Napoleon supports distributed computing systems by providing policy translators for the various enforcement mechanisms in the distributed system.

Modeling workflow in Napoleon is simple, because the underlying concepts of workflow are consistent with the Napoleon model. However, implementing workflow is more complicated. RBAC policies are primarily class-based, but workflow policies are very much instance-based. We discuss these issues and propose a solution that incorporates Napoleon into a workflow management system.

First we review the Napoleon model and software tool. Napoleon has evolved significantly since last year. In particular, the model is more general and more flexible.

2. Revisiting Napoleon

> Pastry is like mathematics. Everything is logical. If you know the basic building blocks, you can make anything.
>
> Jacques Torres
> Dessert Circus[11]

Napoleon is the common name for a family of desserts that are created by alternating layers of pastry with sweet, creamy filling and then finishing with a glaze of icing or a dusting of confectioners' sugar (see Figure 1 (a)).

Napoleon is also an acronym for the Network Application POLicy EnvirONment, a role-based access control (RBAC) modeling environment [10]. The environment consists of a policy model and a software tool for defining and managing the model. The software tool is implemented in Java with a model-view-controller architecture.

Like the dessert, the Napoleon policy model is multi-layered (see Figure 1 (b)). Each layer defines a set of roles that become policy building blocks for all higher layers. The bottom policy layer defines the most primitive access control policy. This policy layer is typically application-specific and is defined in terms of the access control mechanisms that manage the application's resources. The second through penultimate layers use the roles defined at lower layers to create even more abstract roles that simplify policy

[1] This investigation was performed under a Phase I SBIR from the National Institute of Standards and Technology (NIST) Contract No. 50-DKNB-8-90107. Phase II has been awarded and will begin in the fall of 1999.

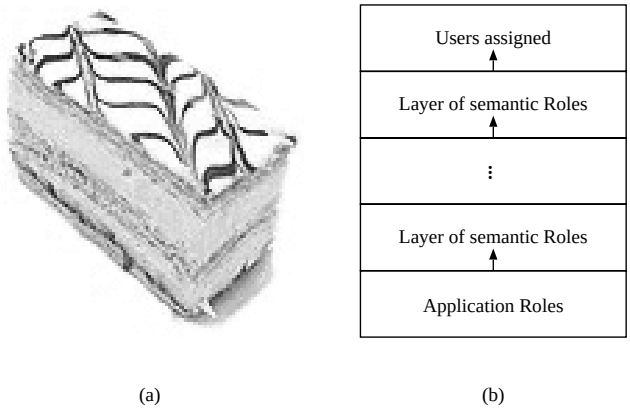

Figure 1. Two Napoleons

management. Roles defined in the top layer are assigned to users. Like the dessert, there can be an arbitrary number of layers, and new layers can be introduced as required.

Unlike the dessert however, each Napoleon policy layer is constructed by a different "chef." Application designers define the bottom layer because they understand best what their resources are and how access to these resources should be constrained. Several chefs may contribute to a single layer, e.g., there may be several applications represented in the bottom layer. System administrators define the top layer because they know who their users are. The chefs for intermediate layers vary with each model. An application suite designer may group the roles of participating applications into roles for the suite. A system integrator may create more abstract roles based on the suite roles. It is important to note that unlike the dessert, the layers in a Napoleon policy model may not be strictly one above the other. A particular layer may build on roles defined in any layer below it, not just the layer immediately below it. For example, the local system administrator is not restricted to roles defined in the penultimate layer. Roles assigned to users can be culled from any layer as needed.

The model described above, with its arbitrary number of layers, is a significant improvement over the original model [10]. The original model contained only two layers: one for the application designer and one for the system administrator. It did not consider the myriad of other principals that should define policy. The current model evolved from our attempts to include some of these principals and from our exploration of workflow support.

The current model continues the metaphor of a key to simplify policy management. A key corresponds to a role. Within each layer, keys are collected into key chains for easier handling. Keys cannot be exported directly to higher layers, but they can be incorporated into a key chain with only one key. Key chains can also contain other key chains, which supports role hierarchies. One of the innovations of Napoleon is that it associates constraints with each key chain. The constraints place additional restrictions on the use of the key chain. For example, a key chain may allow access to patient medical records, but constraints may prevent the holder of the key chain from accessing any records for which the holder is not the primary care physician. Figure 2 illustrates how keys and key chains are used to build semantic layers in the Napoleon model.

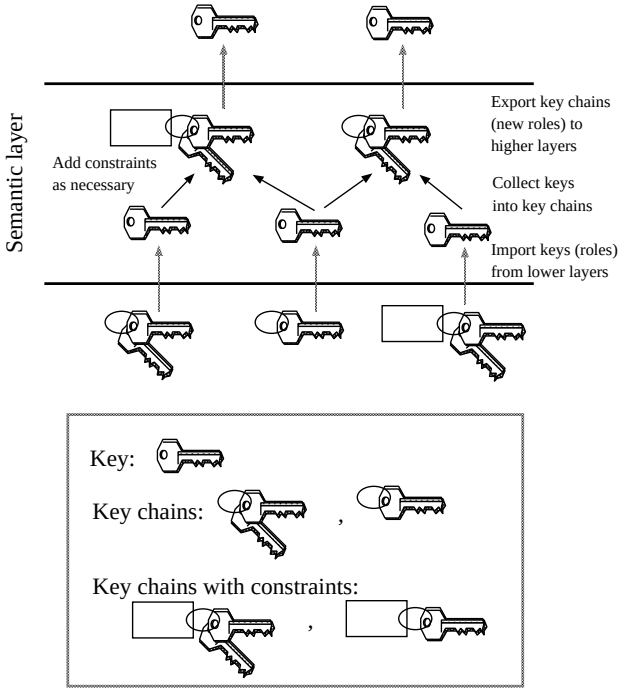

Figure 2. Building semantic layers with keys and key chains

Napoleon's software tool provides a graphic user interface (GUI), or viewer, for each layer of the model. While the middle layers of the model are identical structurally, they differ semantically depending on the chef, so a different viewer is possible in each case. The tool manages the export and import of keys between layers and directs the policy translators to convert the policy rules of the Napoleon model into the enforcement languages of the underlying policy enforcement mechanisms. The tool is very modular: new viewers and policy translators can be added easily.

2.1. Example

Consider a simple example of a hospital data system that is composed of two applications: a CORBA application used by the medical staff to record and share patient information and a COM billing application. The hospital purchases these applications from a third party integrator.

The system's RBAC policy is modeled in Napoleon as three layers, which are illustrated in Figure 3. In the bottom layer, the designers of the CORBA application and the COM application define their application policies independently. For CORBA and COM-based applications, the Napoleon software tool gathers automatically a list of supported operations, or methods, from the application's interface definition language (IDL) files. The application designer uses Napoleon's GUI to group these methods into convenient sets called handles and then to assign handles to keys. A key designates that the holder has permission to execute the associated methods. Since CORBA and COM are object-based, controlling access to an object's methods is sufficient for controlling access to the object itself.

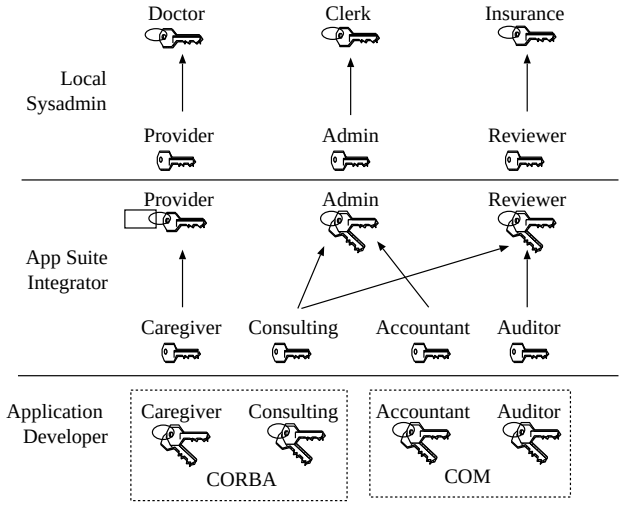

Figure 3. An example Napoleon model

To define the application security policy, the application designer uses the Napoleon GUI to collect keys into key chains and marks the key chains for export to higher model layers. By marking key chains for export, the application developer creates policy building blocks for other layers to build upon. It is similar to creating a software interface. Anything not explicitly included in the interface is not available for use outside the layer.

For our simple example, the CORBA-based, patient information application designer exports two key chains: a CAREGIVER key chain for creating and modifying patient records and a CONSULTING key chain for only viewing patient records. The COM-based billing application designer also exports two key chains: an ACCOUNTANT key chain for generating billing data and an AUDITOR key chain for only viewing billing data. These four key chains represent application-specific roles that are available as building blocks for higher layer policies.

In the middle layer, an application suite integrator imports the four key chains from the application layer. Once a key chain is exported, it is considered an atomic entity, so it is considered a key by all higher layers. The application suite integrator is charged with defining a policy that spans all applications in the suite. In this example, the application suite builds three key chains for export: the ADMIN key chain that contains the CONSULTING key and the ACCOUNTANT key, a PROVIDER key chain that contains the CAREGIVER key, and a REVIEWER key chain that contains the CONSULTING key and the AUDITOR key. The PROVIDER key chain includes constraints that the holder must be a primary care provider for the patient whose records are being accessed.

At the top layer, the three key chains exported from the middle layer (ADMIN, PROVIDER and REVIEWER) are available as simple keys. The four key chains exported from the bottom layer (CAREGIVER, CONSULTING, ACCOUNTANT and AUDITOR) are also available in the event that ADMIN, PROVIDER and REVIEWER are not sufficient, but they are not immediately visible.

While the hospital is tied to a regional information network, it employs a small staff that must wear many hats. The system administrator uses Napoleon to create three key chains to assign to users: the DOCTOR key chain contains only the PROVIDER key, the INSURANCE key chain contains only the REVIEWER key, and the CLERK key chain contains only the ADMIN key. The constraints applied to any keys contained in a key chain apply to the key chain also. For example, a user in the DOCTOR role can only modify patient records for which the user is the primary care physician.

Once the hospital's security policy is defined, the system administrator directs Napoleon to translate the policy for the CORBA and COM object managers. These object managers enforce the policy for their respective objects. In other words, as users attempt to access patient records or billing data, the object managers ensure that the users have the appropriate role and that stated constraints are satisfied.

We will return to this example later in the paper to consider workflow.

3. Workflow

A workflow is "the computerized facilitation or automation of a business process, in whole or part." [5] Workflow

technology is a promising solution for protecting business assets, because it controls not only who has access to what but when that access occurs.

Figure 4 illustrates a simple workflow for processing employee expense reimbursements. We represent the workflow as a directed graph with one entry. Each node in the graph is a workflow activity, or *step*, and the edges determine the order in which steps must occur. Associated with each step are the object(s) that will be accessed (e.g., "check request"), the operation(s) that will be performed ("prepare") and the performer ("EMPLOYEE"). In this simple workflow, an employee must first prepare an check request form and have that form approved by a manager. Then a company accountant drafts the check, and finally the treasurer approves (signs) it. The most obvious constraint is that the steps must occur in order, but other constraints are possible. We may require that the approving manager be the manager for the employee. We may require also that no two functions be performed by the same individual, which is called separation of duties [9].

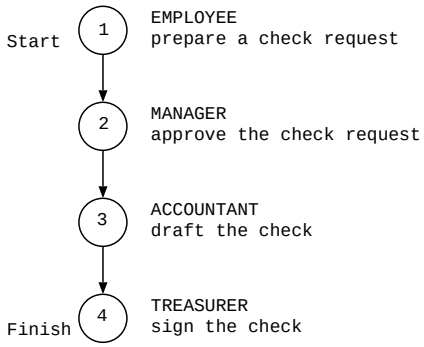

Figure 4. Simple workflow

Riddle [8] identifies the fundamental concepts of workflow and describes the relationships between them (see Figure 5).

- A *step* is a unit of work. It may require several resources to complete. Associated with the step are those resources and the role required to perform it.
- A *work product* is an artifact created or modified by steps. Steps use and produce work products.
- A *role* represents the accesses that are required to perform a step.
- A *workflow condition* is a predicate that must be satisfied during step performance. It is often expressed as entry and exit conditions on the step, that is, the step can begin when and can end when the conditions are true.

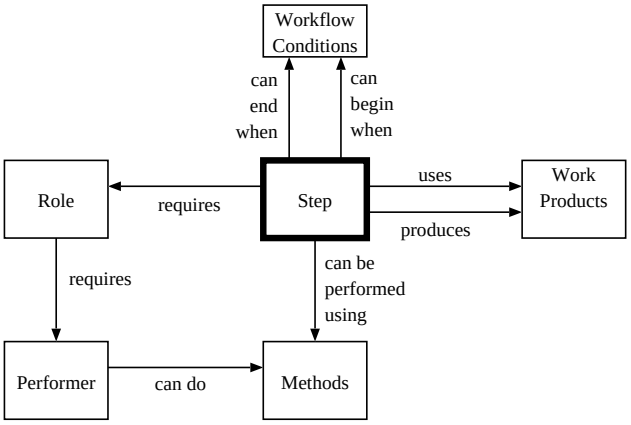

Figure 5. Fundamental concepts of workflow

- A *performer* is a person or tool with the skills necessary to complete the step. A role may require special skills and therefore a specific performer.
- A *method* is an approach for carrying out a step. A step can be performed using one of several methods. The performer can do these methods.

Several of these concepts, such as roles, methods and performers, are also fundamental concepts for RBAC. Even *work products* is familiar; it is just a different name for the resources to be accessed. Only steps and workflow conditions are really new.

Riddle relates concepts a bit differently than the RBAC community, but the differences are superficial. Figure 6 illustrates Riddle's concepts using a role-based perspective, rather than the step-based view of Figure 5. From this perspective, we see that steps are like sub-roles. They define a group of accesses that are specific to a task. Workflow conditions determine when the sub-roles are active. A role, then, is a collection of steps and their associated workflow conditions.

Workflows are enforced by a workflow management system (WMS). The user interacts with the WMS to gain access to resources controlled by the workflow. Automated workflow technology has evolved significantly since it was introduced thirty years ago for office automation systems. Early workflow systems did not acknowledge the variety of ways that humans perform a task. So researchers focused on better modeling techniques, and today workflow research is more interdisciplinary: a combination of computer science and social science. The WMS must encompass non-computer activities such as meetings, handle unexpected contingencies, and allow new workflows to be constructed from existing workflows [3]. Workflow process models must be

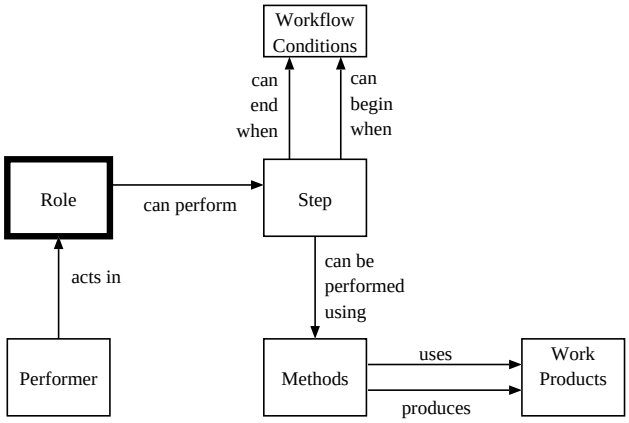

Figure 6. A role-based perspective of workflow

reconciled with the rich variety of activities and behaviors that comprise "real work" [1]. In short, workflow management is a complex activity, and we want to leverage existing technology as much as possible.

4. Napoleon and workflow

Abbott *et al* [1], Feinstein *et al* [4] and Bertino *et al* [2] note that workflow management can be simplified considerably by adopting an RBAC model. However, many role-based models fail to include the role authorization constraints that are required for workflow [2]. As Napoleon includes role constraints, it is a good candidate for workflow policy management.

Our design strategy for incorporating Napoleon into a WMS is to let each tool do what it does best. Napoleon is a policy management tool. While it may be tempting to extend Napoleon with workflow management features, the complexity of workflow management would overwhelm it. Instead, we let Napoleon serve as the policy management engine for a WMS. We use Napoleon to specify and enforce certain aspects of the workflow policy. The divisions of labor between Napoleon and the WMS are described below.

4.1. Specifying workflow in Napoleon

We begin by assuming that the workflow is defined in the WMS and imported into Napoleon. The workflow is imported as a collection of steps. Note that the workflow conditions associated with each step are not imported. We could model these conditions in Napoleon, but we will explain in Section 5 why we choose not to do so.

Our approach is to model workflow as a new layer in a Napoleon model. The new layer looks structurally like the other layers, that is, it has keys and key chains with associated constraints. The difference is in how we build and interpret it. We call the new layer the *workflow layer*, and we introduce a new "chef," the workflow administrator, to construct it.

The workflow administrator begins by assessing the keys that are available for the workflow. The workflow will require certain operations to be performed. If those operations are not represented in the available keys, the workflow administrator must create new keys. Once the necessary keys are imported, the workflow administrator collects the keys required for each step into a key chain that represents the step. The collection of key chains defined in this layer map one to one to the collection of steps in the workflow. The workflow administrator marks each step for export to the next layer, where they are assigned to the roles that will perform them. Several steps may be performed by the same role.

4.1.1. An example with workflow

To illustrate this process, let us return to the hospital scenario from Section 2.1. Suppose the system administrator, who also happens to be the workflow administrator, wants to specify the simple workflow illustrated in Figure 7. This workflow states that whenever a DOCTOR updates a patient's medical record with treatment information, the CLERK must prepare a bill for the treatment. The bill must then be reviewed by the INSURANCE representative, who may authorize partial payment. Finally, the CLERK bills the patient for the remaining balance. This workflow ensures that all bills are reviewed by the insurance representative before they are mailed to the patients, and it ensures that no insurance payment is authorized without a bill.

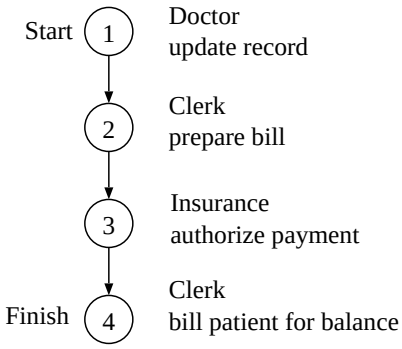

Figure 7. A simple workflow

Figure 8 illustrates how the new workflow layer is mod-

eled in Napoleon. The bottom and second layers are constructed as before. Then the workflow administrator (who may be the system adminstrator) imports the keys (PROVIDER, ADMIN and REVIEWER) necessary to perform the workflow from the second layer. We assume these keys are sufficient, but the workflow administrator could revisit the lower layers and construct new keys if appropriate.

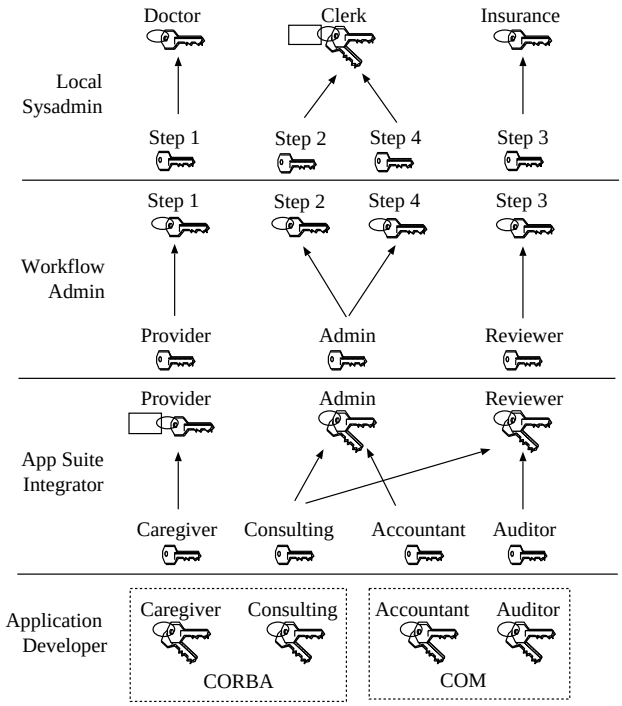

Figure 8. Modeling the workflow in Napoleon

These keys are collected according to the steps that require them. Step 1 requires only the PROVIDER key. Both step 2 and step 4 require the ADMIN key, so the ADMIN key appears on two separate key chains. If different operations are required between the two steps, we could introduce constraints on one or both of the key chains. Finally, step 3 requires only the REVIEWER key. The workflow administrator marks these four steps for export to the system administrator level, where they are assigned to the roles (DOCTOR, CLERK and INSURANCE) that will perform them. In the case of a role that can perform multiple steps (for example, CLERK), constraints are used to determine the appropriate step.

The main difference between a Napoleon model without workflow and a Napoleon model with workflow is that the latter divides roles into sub-roles by task. A Napoleon model simply describes sets of sets, so the division is natural. However, as we will discuss next, there are huge differences in how these models are enforced.

4.2. Enforcing workflow in Napoleon

Napoleon is designed to provide central policy management with distributed policy enforcement. Once the policy is defined, it is "pushed out" to the various enforcement mechanisms in the distributed system. If the policy changes, the new version is pushed out. Napoleon makes no access decisions itself.

Workflow management, on the other hand, requires some central policy enforcement. First, there can be many instances of a workflow active simultaneously. The accesses permitted a specific user may vary depending on the instance. Each access request must be bound to the appropriate instance, and that binding must occur in the WMS.

Second, for each workflow instance only one step (the *current step*) is active at any time. From an access control perspective, the permissions associated with the current step are granted only when the step begins and are revoked immediately after the step concludes. Each instance of a workflow may have a different current step at any point in time. The WMS must track the current step for each workflow instance in order to determine appropriate accesses.

Our initial investigation focused on ways to enforce workflow entirely within the local enforcement mechanisms. To satisfy workflow's central enforcement needs, we envisioned a *workflow object* would track the current step for each instance of a workflow. Napoleon would create the workflow object and bind it to the resources it controls. For each access request, the local enforcement mechanism would examine the corresponding workflow object and verify that the request is approved for the current step. If the request is approved, the local policy ("pushed out" as usual by Napoleon) would be enforced for that resource. The local enforcement mechanism would update the workflow object's indicator of *current step* as required.

There are several disadvantages with this approach. First, Napoleon must be modified considerably to create and distribute workflow objects. Second, each access request requires an additional permission check to the workflow object, which may be expensive. Third, we must trust the enforcement mechanisms to update the current step correctly. An enforcement mechanism could circumvent the workflow policy with malicious updates. Fourth, this approach would duplicate much of the workflow management processing already handled by the existing WMS. Clearly this approach is very invasive, so we refocused our efforts on a solution that leaves Napoleon and the local enforcement mechanisms relatively unchanged.

5. A Napoleon-based workflow management architecture

Olivier [7] notes that workflow policy enforcement can be partitioned into three layers, from lowest to highest: controlling access to resources, controlling access to steps and application-specific enforcement. A useful split occurs in the middle, or step, layer. Steps are a natural primitive for workflow designers. A WMS is specialized to create steps, determine their proper order and control execution of workflow instances according to that order. These operations are unique to workflow technology. However, access for a particular role to the resources associated with a particular step can be controlled by mechanisms that are commonly available in non-workflow domains.

Our solution exploits these partitions by assigning the step layer and the application-specific layer to the WMS and by assigning the resource layer to Napoleon. Workflows, their steps and workflow conditions are specified within the WMS. The steps are then exported to Napoleon, where resources and roles are bound to them. During workflow execution, the WMS manages workflow instances and directs Napoleon to grant and revoke access, as appropriate, to specific steps. Workflow conditions are enforced by the WMS because they determine when the access grantings and revocations should occur.

A high-level design of our solution is illustrated in Figure 9. This design illustrates two modes: policy specification mode and workflow execution mode. Operations for policy specification mode are noted in *italics*, while operations for workflow execution mode are noted in ordinary text. A classical workflow management system will isolate these modes into two modules: a *specification module*, which enables administrators to specify the workflow, and an *execution module*, which assists in coordinating and performing the procedures and activities [3]. Traditionally the specification module is used only in pre-execution; however, researchers are recognizing the need for the two modules to co-evolve to handle dynamic change and exception handling.

The best way to explain the architecture is with a simple scenario for creating and executing a workflow.

5.1. Start with the workflow class definition.

The workflow designer begins by specifying an access control policy that will apply to all instances of the workflow. The designer creates the workflow and its steps using the specification tools in the WMS. This information is then exported to Napoleon, where the binding of resources and roles to steps (as described in Section 4.1) occurs. Napoleon has already gathered a list of available object classes from the IDL files of its object managers. This list is also provided to the WMS for creating workflow instances (see Section 5.2 below).

When this process is complete, the designer has created an access control policy for a particular *class* of workflow. This policy names the roles required, it identifies the steps that each role may take and the class of resources that can be accessed at each step. However, the policy is incomplete. It does not have enough information to control a workflow instance. For example, it does not name individual objects. The objects that may be accessed will depend on the current step of a workflow instance. Therefore, Napoleon holds onto the policy for now; that is, it does not "push out" the policy for the enforcement engines.

5.2. Create a workflow instance.

At this point, Napoleon is loaded with a set of access control policies for workflow classes. A workflow instance gets created when some event occurs to trigger it. For example, a user requests a check reimbursement form, or a notification appears in a user's inbox. When such an event occurs, the WMS determines the appropriate workflow for the event and creates a new instance of that workflow. The workflow instance is stored locally at the WMS. The instance names the specific objects that may be accessed and the specific users that may access them.

When a workflow instance is created in the WMS, it must also be created in Napoleon. The WMS provides Napoleon with the necessary information to instantiate the appropriate workflow's class access control policy, which means providing constraints such as "if object is named `foo.txt`" that will be added to the instance copy in Napoleon. The instance policy names (via constraints) the specific objects that can be accessed. If all specific objects are not known when the instance is created, the WMS may provide additional constraints for that instance later.

In summary, the workflow instance definition in Napoleon looks like the class definition except that it also contains the constraints that name specific objects.

5.3. Execute the workflow instance.

The execution phase highlights the simplicity of this solution. The WMS controls the execution of the workflow instance. It determines the proper sequence of steps (e.g., what branches are executed), and it knows which steps are active. It decides when a step should start (become active) and when it is completed (and thus become inactive). The WMS does what it implies: it manages the workflow. However, it relies on Napoleon to manage the access control policy.

As the workflow executes, the WMS directs Napoleon to grant access to the active steps and revoke access to inactive

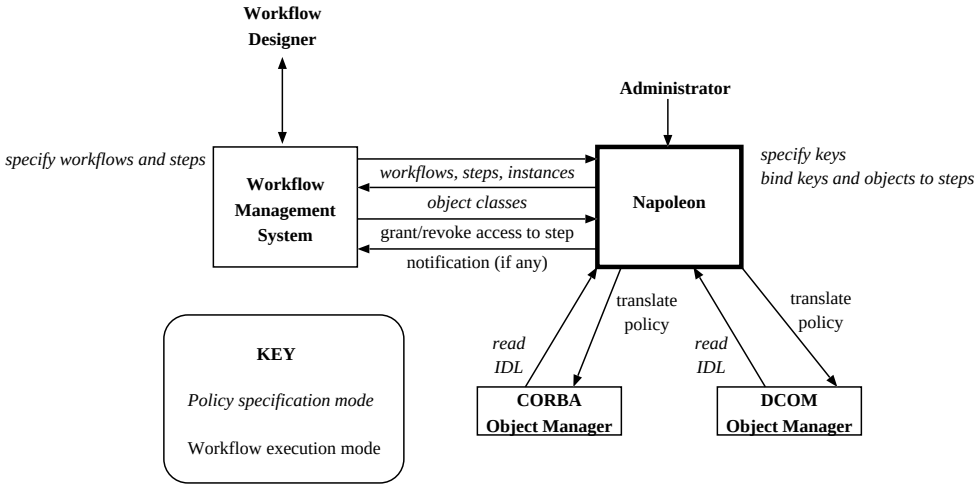

Figure 9. A Napoleon-based workflow management architecture

steps. No policy is translated for the object managers unless directed by the WMS. For example, suppose that step 1 of workflow instance P is active. Once step 1 is complete, the WMS will direct Napoleon as follows:

> Revoke access to step 1 in instance P, then grant access to step 2 in instance P.

Note that Napoleon runs in tandem with the WMS. With regard to policy translation, the only change in Napoleon's behavior is that it now "pushes out" the policy a step at a time rather than all at once.

6. Conclusions

We have described a workflow management architecture that incorporates Napoleon for workflow policy management. The architecture exploits the natural partitions in workflow policy management by assigning workflow-specific tasks to the WMS and workflow-generic tasks to Napoleon. This approach lets each tool do what it does best.

Napoleon offers many benefits to workflow management, including simplified policy management and support for heterogeneous, distributed computing systems. Napoleon's flexible model lets workflow be introduced at any layer.

The support for distributed systems lets a workflow's control extend beyond the local system or local network. A business's divisions may be far-flung across the Internet, so workflows may span several divisions or even several companies (supplier, distributor, etc.). Also, a workflow may need to control resources under the purview of legacy enforcement mechanisms as well as resources managed by newer standards like CORBA. In fact, the WMS does not have to know how the resources under its control are managed. Napoleon acts as a "universal adapter" between the WMS and the policy enforcement mechanisms.

7. Future work

An outstanding issue for Napoleon is how constraints should be implemented for CORBA and COM/DCOM objects. Neither system handles instance policies very well, although we have a sample approach for enforcing constraints in CORBA. The Object Management Group has looked at the issue [6], and we will look more closely at their findings and at related work by the Workflow Management Coalition (WfMC) [12]. We will examine the issue of constraints more fully in the next phase of this effort.

We will also consider other ways in which Napoleon may benefit workflow management, such as with support for hierarchical workflows.

Finally, we will identify a suitable WMS to prototype our architecture.

References

[1] K. Abbott and S. Sarin. Experiences with workflow management: Issues for the next generation. In *Computer Supported Cooperative Work (CSCW)*. ACM, 1994.

[2] E. Bertino, E. Ferrari, and V. Atluri. A flexible model supporting the specification and enforcement of role-based authorizations in workflow management systems. In *ACM Workshop on Role-Based Access Control*. ACM, 1997.

[3] C. Ellis and G. Nutt. Workflow: The process spectrum. In *Workshop on Workflow and Process Automation in Information Systems*. National Science Foundation, May 1996.

[4] H. Feinstein, R. Sandhu, C. Youman, and E. Coyne. Final report small business innovation research (sbir) role-based access control phase 1. Technical Report Department of Commerce Contract number 50-DKNA-4-00122, NIST, May 1995.

[5] D. Hollingsworth. Workflow reference model v. 1.1. Technical Report TC00-1003, Workflow Management Coalition, January 1995.

[6] Object Management Group. Workflow management facility. Technical report, Object Management Group, July 1998.

[7] M. Olivier, R. van de Riet, and E. Gudes. Specifying application-level security in workflow systems. In *9th International Workshop on Database and Expert Systems Architectures (DEXA '98)*. IEEE Computer Society, August 1998.

[8] W. Riddle. Fundamental process modeling concepts. In *Workshop on Workflow and Process Automation in Information Systems*. National Science Foundation, May 1996.

[9] R. Sandhu. Separation of duties in computerized information systems. In *Database Security IV: Status and Prospects*, pages 179–189. North Holland, 1991.

[10] D. Thomsen, D. O'Brien, and J. Bogle. Role based access control framework for network enterprises. In *14th Annual Computer Security Applications Conference*, December 1998.

[11] J. Torres. http://www.jacquestorres.com/ bookinformation / book1 / jtbook.htm.

[12] Workflow Management Coalition. Workflow security considerations — white paper. Technical Report WFMC-TC-1019, Workflow Management Coalition, February 1998.