# Instance-level access control for business-to-business electronic commerce

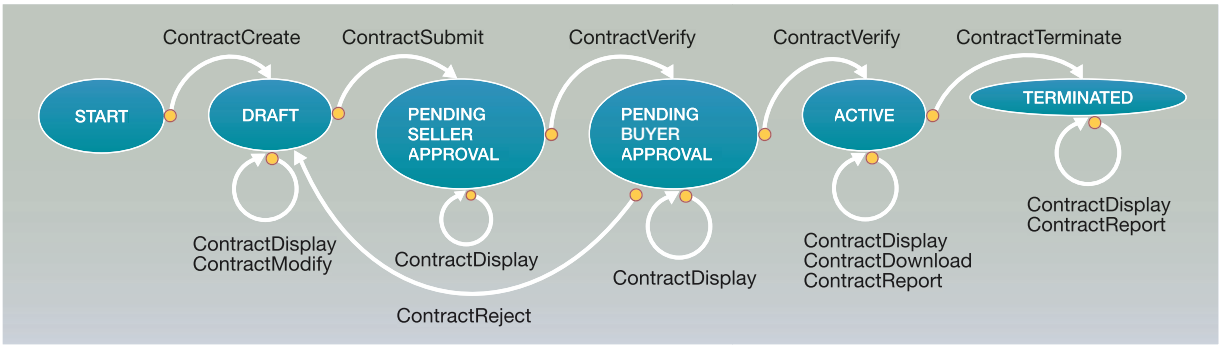
by R. Goodwin
S. F. Goh
F. Y. Wu

The emergence of e-marketplace Web sites that contain proprietary information from multiple organizations requires the creation of new access control schemes that provide fine-grained access control while reducing both administrative and run-time overhead. It is also desirable to have clear, concise, and easily configurable definitions of access control policies that are aligned with business processes, and to have these policies enforced consistently throughout an e-commerce system. In this paper, we describe a policy-based access control scheme, and its implementation, that allows access to individual instances of resources to be specified in a concise and computationally efficient manner. We model business relationships between users and business objects and use implicit grouping of users and resources. These concepts allow policies to refer efficiently to aggregates of resources and users and to document the intention of an authorization policy. Our access control scheme is implemented as an application-level access control mechanism within IBM's WebSphere® Commerce Suite, Marketplace Edition. We use this implementation to provide examples and give performance data. For future work, we discuss how our policy-based, resource-level access control scheme might be enhanced to augment language-level access control schemes, such as the Java™ 2 Platform, Enterprise Edition (J2EE™) security model.

The promise of electronic commerce is that it can improve economic efficiency by increasing the pool of potential buyers and sellers and by reducing transaction costs. Electronic marketplaces, called e-marketplaces, enable electronic commerce by serving as centralized hubs where buyers and sellers can exchange information about products and services and conduct business transactions. In order to enable this function, e-marketplaces must gather, store, and distribute proprietary information from a multitude of organizations. Access to e-marketplace functions and information must be properly controlled to ensure integrity of the process; this is essential if an e-marketplace is to attract and retain participants.

Typically, an e-marketplace defines a set of business processes and services that it provides, signs up businesses, and gives them access to an appropriate subset of these processes and services. The access control policies of the e-marketplace owner define what actions these businesses can perform in the e-marketplace. These businesses in turn grant access to their employees, based on their own access control policies. In granting access, the e-marketplace owner and the individual businesses need to consider not only the business processes and the steps within each process, but also the instances of business objects used in each step.

Figure 1 Sample contract creation process. Nodes are process states, and transitions are process steps. For simplicity, we have not shown all possible transitions and have not included buying against a contract.




To illustrate the requirements for authorization within an e-marketplace, in this paper we use the simplified process for recording a pricing contract between two organizations available within IBM's WebSphere* Commerce Suite, Marketplace Edition software (WCS MPE). Pricing contracts represent agreements in which one business agrees to sell certain products to another business at agreed-upon prices or discounts over a prescribed period of time. A contract can include minimum and maximum quantities of goods and often represents volume discounts for goods to be purchased over a period of time. Recording a contract allows users in the buying organization to see contract prices for items in the catalog and allows them to place orders against the contract, fulfilling the buyer's obligation under the contract. Pricing contracts result from negotiations between the buying and the selling organizations. WCS MPE provides several mechanisms for negotiating pricing contracts, including request for quote (RFQ) and reverse auction business processes. Terms can also be negotiated through e-mail messages, faxes, and phone calls. Any of these processes can lead to a contract that needs to be recorded for use in the marketplace.

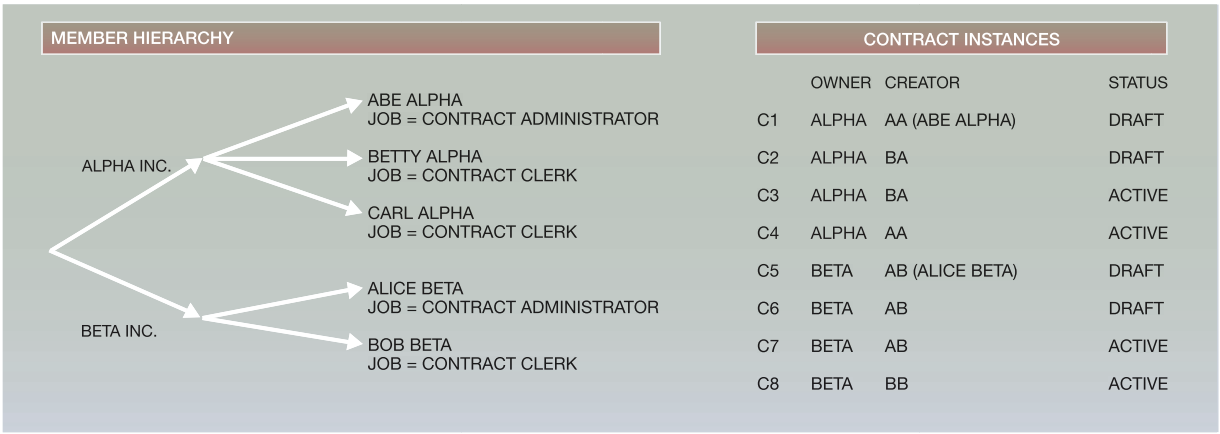
Figure 1 shows a simplified contract recording process. The initiator of the contracting process first drafts a contract, either by entering it directly or by using the results of a negotiation process. The contract is approved by the selling organization, and then sent to the buying organization for approval. Once both sides approve, the contract becomes active. The contract is terminated automatically at the end of the contract period.

To enable this process, the e-marketplace owner will authorize certain businesses to initiate and participate in the contracting process. These businesses in turn authorize certain of their employees to initiate and participate in the contracting process (see Figure 2). In granting this access, the e-marketplace does not grant an organization the right to perform contract actions on *any* contract, only those contracts in which the organization is a participant. Furthermore, an organization might want to limit which contracts an individual can modify. For example, some organizations might limit a contract clerk to the contracts that they initiated, while others allow their contract clerks to modify any contract that the organization owns. Within an organization, policies might vary by geography. For example, contract clerks located at the business headquarters might be able to modify any contract, whereas those located at the branch offices are limited to the contracts they initiated. In addition to authorization policy, organizations might also want to customize the business process workflow. Although authorization is required to enable a workflow, workflow and its interaction with authorization is beyond the scope of this paper. (See Bussler[1] for a discussion on workflow and authorization.)

For purposes of this paper, we consider the following sample access control policies:

- Contract clerks and contract administrators can create contracts.
- Contract clerks can view only contracts they created.
- Contract administrators can view any contract initiated by their organization.

Figure 2 Sample membership hierarchy with two organizations: Alpha Inc. and Beta Inc., five users, and eight sample contracts showing owner, creator, and status. Other details are omitted.




- Contract clerks can modify only contracts they created that are in the draft state.
- Contract administrators can modify any contract their organization created that is in the draft state.

## Design objectives

We have tried to create an authorization design with the following characteristics:

*Expressive*. The authorization language must be expressive enough to be able to handle a wide variety of business policies used by the participants in the e-marketplace.

*Comprehensible*. The authorization language should express the authorization policy in terms of the business processes, and business objects they operate on, and make sense to a business analyst in order to minimize the technical knowledge required to administer the system.

*Compact*. E-marketplaces can involve thousands of organizations, hundreds of thousands of products, and millions of transactions. The size of the authorization policy should be relative to the number of business processes and types of business objects, rather than the number of instances of each object in the system.

*Efficient*. Transaction volume on a successful marketplace, and the need for quick response times, requires that authorization checks place only a minimum overhead on each transaction. If authorization checks cannot be done efficiently, then externalized schemes must be replaced with hard-coded application logic.

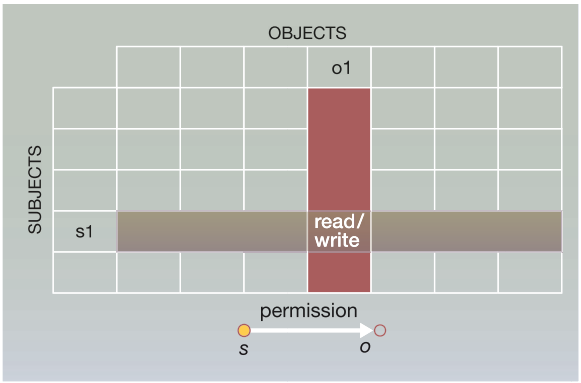
In some cases, these aims conflict. For example, a language that is sufficiently expressive, such as first-order predicate logic, might not be sufficiently comprehensible to a business analyst and might not have an efficient run-time implementation, or it might allow policies that have no efficient run-time implementation.

In the remainder of this paper, we review related work in the evolution of authorization schemes. We identify some limitations of previous approaches and outline our approach. We then describe authorization policy implementation in WCS MPE and use this implementation to demonstrate some performance results. We conclude with some comments on future work and the evolution of authorization for Java**-based applications.

## Background and related work

A good review of prior work on security policies and models can be found in Summers.[2] When multiuser operating systems were first developed in the 1970s, early access control models were created to deal with potential conflicts between users. The access matrix model[3] is a simple way to represent explicitly the operations that a user is authorized to perform on a resource object. As shown in Figure 3, each row of

Figure 3 An access matrix maps subjects to objects, granting permissions. Permissions can also be represented as directed arcs from subject to object, labeled with the permission.




the access matrix corresponds to a subject (user, or process running on behalf of a user) and each column corresponds to an object. Each cell of the matrix is filled with a list of the permissions that the subject has over the object. The permissions are in terms of operating system primitives, such as read, write, and modify. Each column in the matrix represents the access control list for the object of that column. The access matrix model is widely used, flexible, and easy to understand. For our example, Figure 4 shows the required permissions, where each permission arrow corresponds to an entry in the access matrix. One problem with the access matrix is its explicit enumeration of rights for each subject-object pair, which creates a heavy maintenance burden. This model is also not capable of expressing the intent of the assignment. For example, a policy to allow every subject access to read an object would imply that any new subject added should have read access. However, a column in which everyone currently has read permission cannot be assumed to mean that the object is readable for a new subject. A related problem is how to assign permissions for a new object.

Protection of objects can also be represented by a directed graph, as in Figure 3. In this figure, vertex $s$ represents the subject, vertex $o$ represents the object, and the labeled directed edge from $s$ to $o$ represents the set of permissions that subject $s$ has over object $o$. The permissions in the set must be members of a fixed set of permissions. In the take-grant model, this set of permissions can include "take" and "grant" permissions, which enable transfer of permissions by one user to another. The number of permission decisions is of complexity $O(|s| \times |o|)$, where $|s|$ is the number of subjects and $|o|$ is the number of objects. This corresponds to the number of possible labeled arcs in Figure 4.

Recently, role-based access control (RBAC)[4] has gained acceptance for management of computing resources because it reduces the administrative burden as compared with access control lists and other early forms of access control. In RBAC, a role is a named job function within an organization that describes the authority and responsibility conferred on a user. RBAC requires access rights to be assigned to roles, rather than individual users, and users obtain rights by virtue of being assigned appropriate roles, as illustrated in Figure 5. A role is traditionally defined in the access control literature as an association between a group of users and a group of permissions. (For example, see Barkley and Cincotta.[5]) RBAC can also be extended to include role hierarchies and constraints between roles. Using a role hierarchy, the Contract Administrator role would be superior to the Contract Clerk role and would inherit all the privileges of the Contract Clerk role. Using role constraints, it is possible to prevent an employee who is a Contract Clerk but not a Contract Approver from approving contracts that the employee created.[4]

RBAC eases the administration of access controls because the roles defined for an organization tend to be relatively static entities, whereas the users who occupy those roles change frequently. Similarly, the set of permissions associated with a role are expected to be stable, whereas resources and access to those resources may be dynamic. RBAC also allows separation of the personnel function of assigning subjects to roles from the business process administration function of defining which privileges a role has. The complexity of permission decisions is $O(|s| \times |r|) + O(|r| \times |o|)$, which can be significantly less than for an access matrix, if the number of roles is much smaller than the number of users. Further reductions in complexity can be achieved by grouping objects by their type. Figure 6 shows an RBAC scheme where roles are granted permissions on types of objects rather than individual objects. Because each object implicitly maps to the correct type, the complexity of permission decisions is reduced to $O(|s| \times |r|) + O(|r| \times |t|)$.[5] However, as we can see from Figure 7, there are difficulties in applying this approach to our example. A different role is required for each user,

Figure 4 The intended permissions for our example. We can represent all the intended permissions using an access matrix.

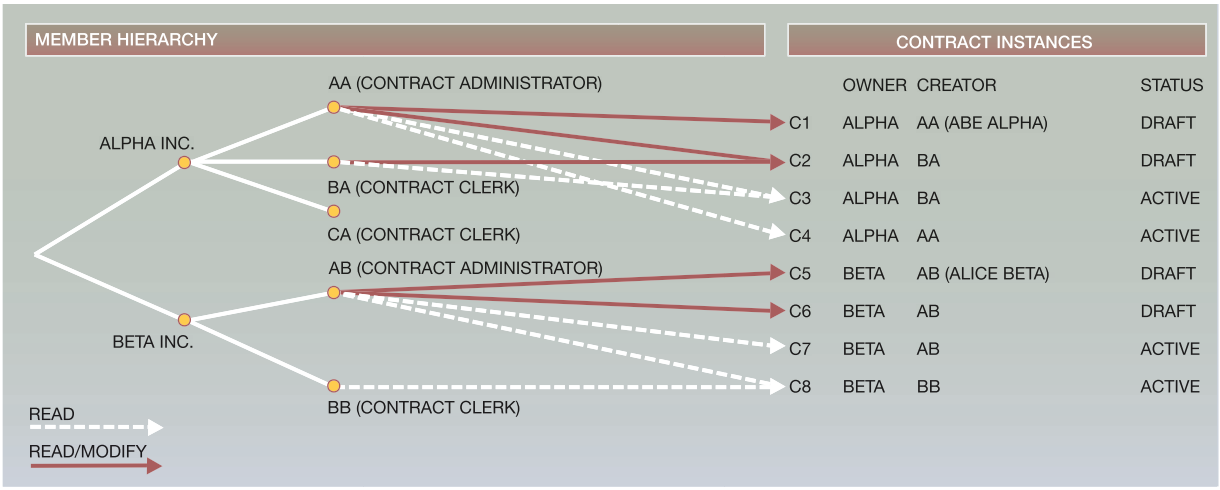
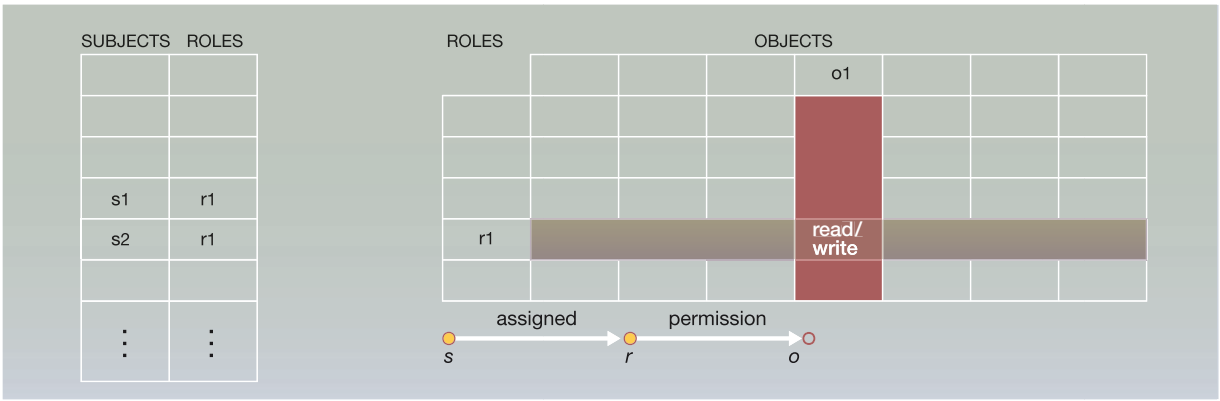



Figure 5 Role-based access control maps users to roles and roles to permissions on objects.




so using roles would increase the administrative burden in this case.

Implicit mapping of roles to privileges on objects is extended in the work on role templates, where privileges are granted to a role for all objects of a given type that satisfy an expression.[6] For example, we could define the role AlphaContractAdministrator = (select, contract, (contract.seller = "Alpha")), where "select" is the action (SQL [Structured Query Language] select), "contract" is the type, and the remainder is the restricted-privilege expression that the contract must satisfy in order for users with the AlphaContractAdministrator role to read the contract. Role templates extend this to allow parameterized definitions such as ContractAdministrator(⟨organization⟩) = (select, contract, (contract.seller = ⟨organization⟩)). This template could then be used to assign to a contract administrator from Alpha the role Contract Administrator("Alpha"). Using this scheme, we would create two templates for contract administrators, the one already given and one for ContractAdministratorModify(⟨organization⟩) = (modify, contract, ((contract.seller = ⟨organization⟩) and (contract.status = "draft"))) to allow modification of the organization's contracts in the draft state. There would be one instance of each template for

Figure 6 To ease administration overhead, permissions can be assigned by object type, rather than by object.

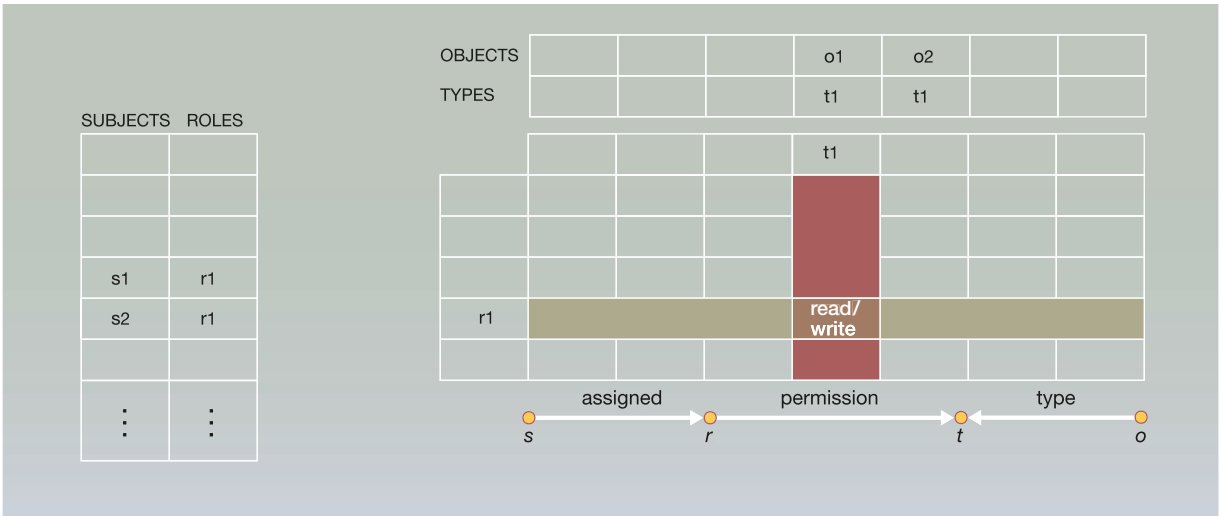
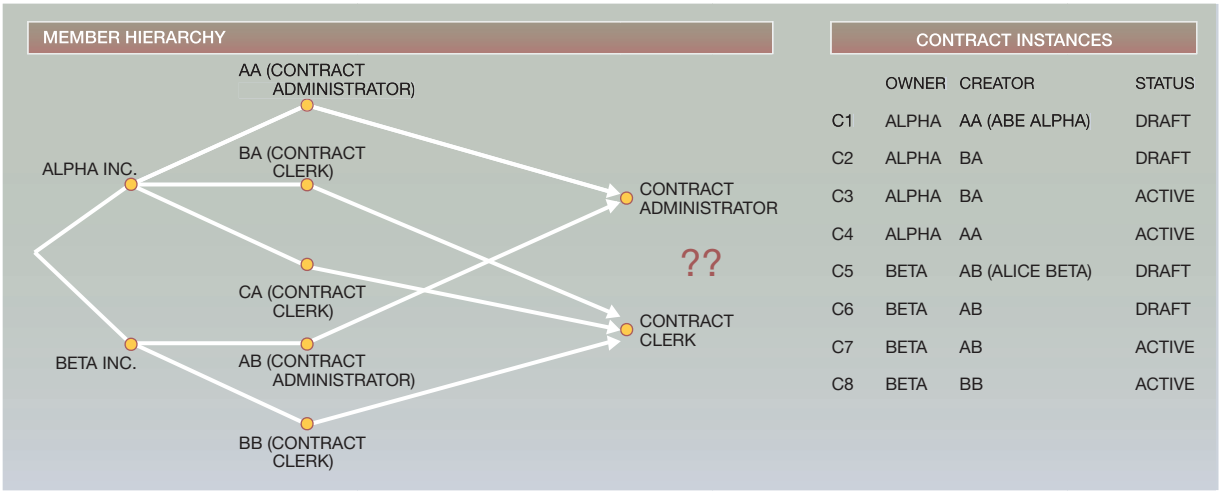



Figure 7 Contract administrator and contract clerk roles are insufficient, because different subjects with the same role should have different permissions. In this case, we need one role for each user.




each organization. For contract clerks, we would need another two templates, with both the organization and the user as parameters:

- ContractClerk(⟨organization⟩,⟨user⟩) = (select, contract, ((contract.seller = ⟨organization⟩) and (contract.creator = ⟨user⟩)))
- ContractClerkModify(⟨organization⟩,⟨user⟩) = (update, contract, ((contract.seller = ⟨organization⟩) and (contract.creator = ⟨user⟩) and (contract.status = "draft")))

We would need to create one instance of each of the contract administrator templates per organization and one of each of the contract clerk templates per contract clerk. We would then assign each user to the correct pair of role template instances. The

number of role instances grows with the number of organizations and the number of contract clerks.

The organization modeling and management (OMM) authorization scheme further extends implicit mapping by using expressions to define virtual relationships between objects.[7] Using virtual relationships, we could define AdministratorForContract = ((*owner*.jobFunction = ContractAdministrator) and (*owner*.organization = *contract*.seller)) where *owner* is the range of the relationship and *contract* is the domain. We could then grant the Contract Read privilege to anyone satisfying this virtual relationship with the contract in question. At run time, the existence of a virtual relationship is checked using the information in the OMM data store. We could also define: AdministratorForModifiableContract = ((*owner*.jobFunction = ContractAdministrator) and (*owner*.organization = *contract*.seller) and (*contract*.status = "draft")) and grant users with this relationship permission to modify the contract. However, in the OMM system, virtual relationships are used to control access to process steps, and not to objects. As a result, we would need only the one relationship for contract administrators. Likewise, a single relationship could be used for contract clerks.

An early application of RBAC to the Web is described by Barkley et al.[8] The RBAC/Web system provides the benefits of RBAC in a package easily integrated with common Web servers, supplementing the Web server's own authentication function with role-based authorization. However, it does not offer the level of fine-grained access control described in this paper.

Na and Cheon[9] describe an RBAC design in which roles can be temporarily delegated from one user to another. Granting delegation can be predicated on an exception condition, such as an emergency. Lin and Brown[10] describe extending Intel's Common Data Security Architecture to enable user-defined trust policy enforcement. A key feature is the ability to define customizable policies in the form of logic rules that further restrict the actions of users. The RBAC system presented in this paper is platform-independent and uses relatively simple policies that can likewise be easily configured in a graphical user interface.

A highly adaptable model by which RBAC can be used to administer RBAC is described by Sandhu et al.[11] This model extends RBAC by adding the concepts of administrative roles and administrative permissions, which are dedicated to the management of roles. It is shown that the model can represent and formalize complex management and delegation rules that an organization may have in place. However, the requirements that have driven the design of the WCS MPE access control system did not justify adding this degree of complexity solely to manage the granting and revocation of roles.

As we show in the next section, our design builds upon and extends the previous work just described. Although it exploits the reduction in administrative burden made possible by RBAC and its recent derivatives, the design is strongly driven by both the functional and performance requirements of the customers of this commercial product.

## Design

In this section, we outline our design and implementation for application-level authorization in WCS MPE. Our primary goal was to enhance role-based access control to meet the challenges of business-to-business electronic commerce applications. A significant shortcoming of RBAC is the inability to distinguish which instances of a resource an individual role holder can access. As stated earlier, although contract clerks can execute commands to modify contracts in the draft state, they may only be allowed to modify contracts they created. Frequently RBAC systems address this issue by hard-coding this part of the authorization policy as part of the business logic. Our aim is to externalize all authorization decisions, using a set of authorization policies, while still maintaining the performance of hard-coded authorization policies. In the rest of this section, we describe the concepts we use to implement our authorization policies.

**Implicit grouping.** A limitation of RBAC is that assignment of subjects to roles is the only method for aggregating subjects to assign permissions. While roles are typically equated with job function, and the permissions needed to carry out a particular job in a given business may be consistent across all users holding the same position, it may not be the case across organizations or between countries. For example, if contract administrators for Alpha Inc. and Beta Inc. require different permissions, we need to create two roles. If we created the AlphaContractAdmin and BetaContractAdmin roles, their names would imply the purpose of those roles, but the system would not enforce their intended use. Furthermore, if Alpha Inc. is a multinational company, accounting and

legal practices may require that contract administrators in different countries be given different permissions. We may need to create the AlphaUSContractAdmin, the AlphaCanadaContractAdmin, and so on. Again, the role names would imply which subjects an administrator should assign to those roles, but the system would not have captured our intentions.

**In addition to grouping by type, we allow objects to be grouped by other attributes, such as their state.**

To address the problem of efficient subject aggregation, we use implicit grouping of subjects and use these groups to map subjects to permissions on groups of objects. An implicit group is defined by a set of constraints and any subject satisfying the constraints is a member of the group. For example, we would define AlphaUSContractAdmin = [(organization = "Alpha Inc.") and (country = "US") and (job = "ContractAdministrator")]. Note that the constraints refer only to attributes of the subject (organization, country, and job) and constants, such as "US." This restriction allows group membership for the subject to be determined efficiently from the attributes of the subject and the definition of the group. We also allow explicit assignment of users to groups. This is useful for defining groups such as "Trusted Partners" when there is no implicit definition of "trusted."

The use of implicit groups for specifying authorization policies eases administration overhead in two ways. First, if a contract administrator moves from one country to another, then updating the user profile automatically moves the user to the correct group, without requiring an administrator to intervene and know whether a user's country is significant for assignment of permissions. Second, if circumstances change and permissions become dependent on another attribute, such as language, then defining new groups and assigning them permissions is sufficient. The alternative with RBAC is to define new roles and then reassign all people with the old roles the correct new ones.

Implicit grouping also applies to objects. Assigning permissions by type implicitly groups objects by their type (Figure 6). Because types in an object-oriented system can be hierarchical, an object can have multiple types and be a member of multiple type groups. In addition to grouping by type, we allow objects to be grouped by other attributes, such as their state. To implement our contract example, we would define modifiable contracts as contracts with state = "draft."

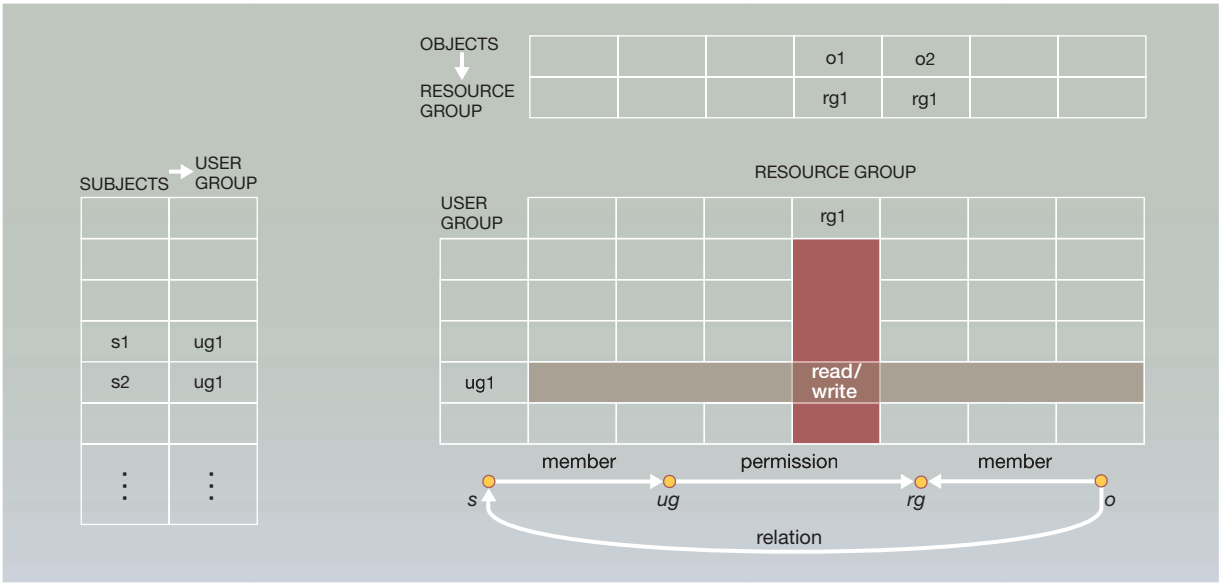
Implicit grouping is identical to role templates[6] when the role templates are used solely for grouping of subjects. However, when the concept of role templates is used to restrict the range of objects over which the role has permissions, it can suffer from the need to proliferate instances of the templates, as described earlier. The use of relationships allows control over object instances based on specific attributes without the need to instantiate all relevant combinations of roles with object attributes.

**Relationships.** An object-oriented design of business objects would maintain meaningful relationships or associations between objects and between objects and users. For our contract example, a reasonable design would represent the one-to-many creator relationship between users and contracts, the one-to-many ownership relation between organizations and contracts, and the many-to-many assignment between users and jobs. The implementation of such a design would necessarily include methods that could be used to determine if a relationship held between a given object and a given user. For example, a getCreator( ) or an isCreator(⟨user⟩) method on a contract object could be used to determine if a user was the creator of a contract. Because these methods would be implemented as part of the application, we would expect that they would be implemented to run efficiently.

It is also important to note that the set of relationships maintained for each type of object can be different. For example, a request for quote (RFQ) object that is targeted at a specific list of people would have to maintain this list and the "recipient" relationship would be defined for RFQs, but not for contracts. Also, if there were no need to record the creator of the RFQ, then there would be no creator relationship defined for RFQs.

In our design, we try to take advantage of the relationships already present in the business object

Figure 8 Policies assign permissions to groups of users to perform an action on a group of resources, if a given relationship exists. Group definitions implicitly map users and objects to groups. Using relationships improves the specificity of rules, while maintaining intention and efficiency.




model, and the associated method implementations. We want to exploit relationships that are already maintained for purposes other than authorization. Figure 8 shows the graphical representation of the RBAC model, extended to take advantage of these relationships. By making the creator relationship on a contract available for specifying authorization policies, we do not place any additional burden on the application in terms of the information it must store, but we do increase the expressiveness of the authorization language. Using the creator relationship, we can determine if a contract clerk can access a given contract. To implement such a policy at run time, we only require that an instance of an object provides an interface that we can use to determine if a user fulfills a particular relationship.

**Ownership.** We could also group resources by their owner and use owner-based resource groups to grant access. Instead, we choose to treat ownership as a fundamental characteristic of every object and take advantage of the membership hierarchy to scope policies. For each resource that can be controlled, we require an owner, which can either be a user or an organization. Ownership is used to determine which access control policies apply to a given object. The policies defined for the owner of an object determine who can access it. Ownership is also transitive. Organizations own their employee's entries and as a result indirectly own the business objects that their employees own. Similarly, the marketplace owns the organization entries and indirectly owns all the business objects in the marketplace. Authorization policies defined at any level in the hierarchy therefore apply to all objects owned by the entity for which the policy is defined and all objects owned by its descendants in the hierarchy. In this way, market-wide policies are defined at the root of the hierarchy and apply to all objects in the marketplace.

To simplify administration and improve run-time efficiency, we allow policies that grant permission but not policies that revoke permissions. By default, no one is allowed to do anything. Only if a policy grants permission is an action allowed. Having only policies that grant permission eliminates the need for conflict resolution, because no policy can logically contradict another. Comprehensibility of policies is also improved because the effect of a policy cannot be changed by the addition or removal of another policy. This also improves run-time efficiency; as soon as a policy is found that grants permission, no other policies need be considered.

Policies are themselves first-class objects in our system, and access to policies is controlled through authorization policies.[12] So, while the policies for an organization define who can access the organizations' resources, the organization administrator cannot change these policies, unless there is a policy that allows this. For most organizations in an e-marketplace, we do not include such a policy. The real advantage of having policies scoped by organization is that it allows the e-marketplace to load different sets of policies for each type of organization and to customize these policies for an individual organization where needed. More sophisticated and trusted organization administrators can be granted access to modify policies for their organizations, but the site administrator can limit the scope of the policies they could change and thereby limit damage they could do to resources owned by their own organization.

**Explicitly callable.** Finally, we make the authorization code directly callable from the application. In many operating systems and language-based authorization systems, calls to perform an action automatically invoke the authorization code. While this is desirable to prevent unauthorized access, it is not sufficient. It should be possible to determine whether something would be allowed without actually having to try to perform it. This functionality is particularly useful for creating user interfaces in which the user is shown only the menu items, buttons, and hyperlinks that he or she is permitted to use. By calling the authorization code to determine if authorization would be granted, the user interface can selectively enable and disable functionality. If the authorization policies are changed, the user interface automatically adapts to provide the correct functionality to each user, eliminating the need to recode the user interface.

**Policies.** In our design, authorization policies are represented as a four-tuple:

[user group, actions, resource group, relationship]

The first and third policy elements must be the names of existing user group and resource group objects, respectively. The actions must correspond to one or more predefined actions, although this element can also take a wild-card value that matches all actions. The relationship must be valid for the resource group. A set of relationships is defined for each resource type. The relationship in a policy should match a relationship defined for some objects in the resource group; for example, if a resource group includes contracts, then a policy could include a "Creator" relationship, because contracts have creators. The relationship is optional, and a policy without a relationship means that the policy does not require a user to have a specific relationship with the object.

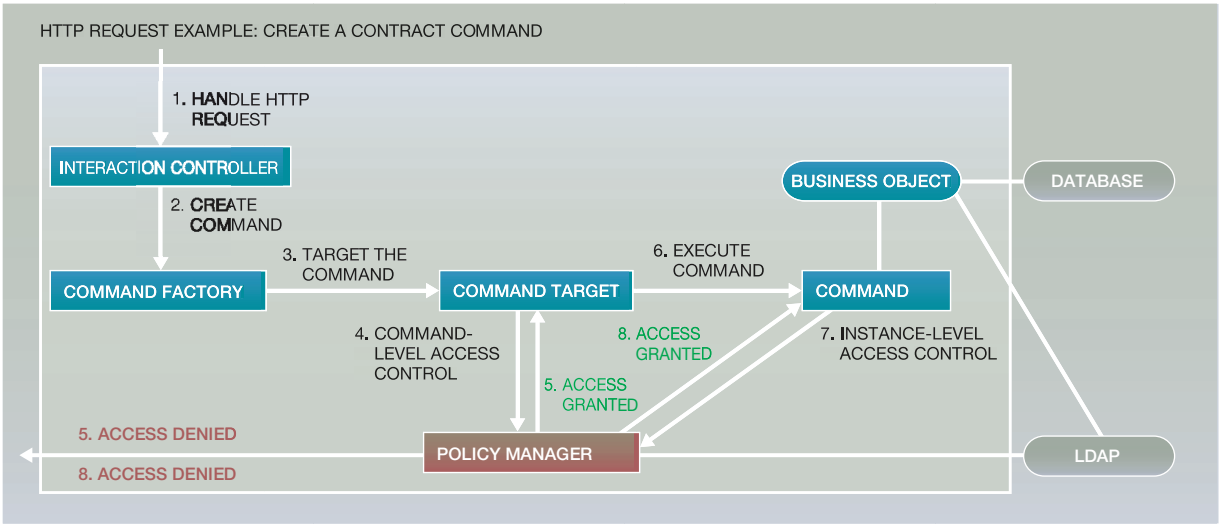
The policy can be interpreted as granting access to anyone in the user group to perform the given actions on any resource in the resource group, provided he or she has the given relationship with that object. The policy only applies to objects owned by the owner of the policy.

**Role assignment.** In RBAC, a system administrator is responsible for assignment of roles to users in order to grant access. The set of roles that can be assigned is the set defined for the system. For an e-marketplace that involves large numbers of organizations and users, we want to distribute user management to administrators within each organization. However, we do not want the organization administrators to be able to assign *any* role. Instead, we limit them to assigning only the roles that their organization has been assigned, and to assigning these roles only to members of their own organization. In this way, the site administrators assign roles to an organization, and the administrators within each organization can then assign these roles to selected individuals in the organization. Of course, role assignment authorization itself is controlled through authorization policies. To maintain integrity, when a site administrator revokes a role from an organization, it must also be revoked from all the users in the organization.

## Authorization in WebSphere Commerce Suite, Marketplace Edition

WebSphere Commerce Suite, Marketplace Edition (WCS MPE) is IBM's middleware for constructing business-to-business electronic marketplace Web sites. It is an outgrowth of the WebSphere Commerce Suite, 4.1 (WCS) software product that is designed for implementing retail e-commerce Web sites. In implementing WCS MPE, we needed to enhance the access control design of WCS to support the more complex policies that are characteristic of business-to-business e-marketplaces, where there are multiple sellers and transactions involve more than just a shopping cart and a credit card. The implementation of the MPE version of WCS also coincided with the transition from C++ to the Java language, which afforded us some freedom in implementing a new access control scheme. However, existing function-

Figure 9 Sample HTTP (HyperText Transfer Protocol) request flow. Each request is associated with a session, and each session can be associated with a user via an authorization step. The Policy Manager is invoked to check user authorization for the command (step 4) and to check authorization for the business object (step 7).




ality for order and payment processing, written in C++ and reused for the MPE edition, continues to use the rudimentary C++ access control inherited from WCS.

WCS MPE is implemented using WebSphere Application Server 3.02, an e-business Java-based application deployment environment. Within this environment, we use the San Francisco Command framework to implement business processes. This framework consists of interaction controllers (ICs) that manage interaction with HyperText Transfer Protocol (HTTP) -based requests and commands that implement steps in a business process.[13] Within a transaction, multiple commands can be chained and commands can invoke other commands to implement substeps in a process.

Figure 9 shows a typical interaction when the MPE server handles an HTTP request. The interaction controller receives the request and calls the command factory to select an implementation of the appropriate command. The command is then given the parameters from the request and invoked via a command target. The use of the command factory allows the functioning of the e-marketplace to be customized by having the e-marketplace administrator configure the command factory to select the appropriate command implementations. The use of a command target allows command processing to be moved between servers for load balancing to improve performance. The command target performs the first authorization check to see if the user has authorization to perform the execution action on the requested command. This check corresponds to a system-level check that determines if the user has permission to execute the command at all. Since the commands are owned by the marketplace, only marketplace-level policies are checked. In this way, the system-wide policies control access to the business functions. Referring to the policies in Table 1, we can determine that for contract commands, only the last two policies could apply, because these are the only two with resource groups that include command objects. If the user has the ContractClerk or the ContractAdministrator role, then one of these policies would grant access.

Within each command, the required business objects are loaded from the database, or in the case of user profiles, from the LDAP (Lightweight Directory Access Protocol) server. The parameters passed to the command indicate which instance(s) of each business object the command should operate on. The resource-level authorization check is performed within each command. For contract commands, the check would determine if the user could perform the given action (command) on the given contract. Be-

Table 1 Contract-related policies. For implicit groups, we include the group definition with its first use. Organization-level policies are loaded at organization-creation time for each organization. The ⟨organization⟩ variables are replaced by the organization identifier.

| | User Group | Action | Resource Group | Relationship |
|---|---|---|---|---|
| **Organization Resource Policies** | ContractAdministrator := (job = Contract/Administrator) and (organization = ⟨organization⟩) | contractRead | Contract := (objectType = contract) | |
| | ContractAdministrator | contractModify | ModifiableContract := (objectType = contract) and (status = draft) | |
| **Market-wide Resource Policies** | ContractClerk := (job = ContractClerk) | contractRead | Contract | Creator |
| | ContractClerk | contractModify | ModifiableContract | Creator |
| **Market-wide Command Policies** | ContractClerk | execute | ContractCommand := (objectType = ContractReadCmd) or (objectType = ContractModifyCmd) | |
| | ContractAdministrator | execute | ContractCommand | |

cause contracts are owned by the organization, we first check the organization policies. The first two policies in Table 1 allow contract administrators to access all contracts owned by their organizations. Organization policies are typically used to grant a user with an administrative job access to an entire class of objects owned by an organization. In such cases, there is no requirement for a direct relationship between the user and the object. For contract clerks, who require a direct relationship to the contracts they can access, we use a market-level policy that includes the creator relationship. Here we are taking advantage of the fact that the system does not allow a clerk to create a contract for any organization but his or her own. If we wanted to enforce a policy that clerks could access only a contract that they had created and that was owned by their organization, we could make this an organization-level policy.

**Policy manager.** Authorization is performed by the PolicyManager instance, which manages authorization policies and carries out authorization checks when invoked. The PolicyManager class is instantiated as a singleton and provides an isAllowed(User, Action, Object) method for determining if a user is allowed to perform a given action on a given object. When invoked, the isAllowed method first looks up the object's owner and retrieves the owner's authorization policies, embodied in policy objects. For each policy, the PolicyManager instance invokes the policy's isAllowed(User, Action, Object) method to determine if the policy grants access. The implementation of the policy object checks to see if the conditions of its four-tuple policy are satisfied. It checks to see if the user is a member of the user group, if the object is a member of the resource group, if the actions match, and for a policy that specifies a relationship, if the user fulfills the relationship with the object. If none of the owner's policies grants access, then the policies of the owner's parent are retrieved and tested. This process is repeated until the root of the membership tree is reached. If no policy grants access, then the PolicyManager instance returns false. (This description of the algorithm, while conceptually correct, does not reflect the actual implementation, which takes advantage of numerous optimizations to achieve the required performance.)

To check to see if an access is authorized, the PolicyManager instance needs to determine the owner of the resource and may need to determine if a user has a particular relationship with the resource. To support the authorization check, we require that business objects implement the Protectable interface. This interface serves as a marker to indicate that au-

thorization is needed for the given resource and provides a getOwner( ) ⇨ owner method to determine the owner and a fulfills(User, relationship) ⇨ Boolean method used to determine if a relationship exists. The efficiency with which authorization can be checked is critical to the overall performance of the system. To provide efficient checking, policies are cached in a nested hash table that is keyed on the action, then on the owner. This arrangement allows the PolicyManager instance to quickly select the set of policies that refer to the given action for a particular owner, and ignore all others. Within the policy object, the checking of the conditions of the policy are ordered so as to perform the most efficient, most restrictive checks first to quickly eliminate policies that do not apply. These and similar optimizations helped to provide a two order-of-magnitude speedup over our initial naive implementation.

WCS MPE was released with a sample e-marketplace for buying and selling shipbuilding materials. This sample comes with a default set of authorization policies that encode reasonable behavior for a default set of roles. Figure 10 shows a breakdown of the policies by level and type. In all, there are 154 different policies at the marketplace level, 33 of which are for assigning permission to execute commands. The remaining 121 are resource-level access policies that make use of relationships. In these policies, there is a fair bit of repetition when a role holder with a relationship is granted access to execute multiple actions. In future versions, we intend to support grouping of actions. This would allow us to reduce the number of resource-level policies by a factor between 3 and 5.

Figure 10 Breakdown of policies by level and type. One organization policy is included in the default set, because these policies were for small organizations for which a single administrator might perform all administrative functions.

| | |
|---|---|
| Market Level | |
| Policies: | 154 |
| Command Policies: | 33 |
| Resource Policies: | 121 |
| User Groups: | 16 |
| Jobs: | 8 |
| Resource Groups: | 34 |
| Resource Types: | 265 |
| Organization Level | |
| Policies: | 1 per organization |
| Resource Policies: | 1 per organization |
| User Groups: | 1 |
| Resource Groups: | 1 |

## Performance results

To measure the overhead that our authorization scheme adds to transaction processing, we performed a series of tests, both with and without authorization checks. The tests without authorization checks were performed using a modified version of the PolicyManager that does not load policies and has an isAllowed method that always returns "true."

To achieve good steady-state performance, the system caches policies and group membership information for logged-on users. Policies are loaded only once, when the system starts. In Table 2, we see that policy loading adds about 0.6 seconds overhead for this small example with three organizations. Since by default each organization has one policy, the total number of policies grows linearly with the number of organizations. The overhead per policy is less than time/(|org_policies| × |orgs| + |market_policies|) = 0.6 seconds/(1 × 3 + 154) = 4 milliseconds, and we have successfully deployed systems with 30 000 organizations. From Table 3, we also see that the caching of group memberships for the logged-on user adds about 8 percent overhead to the logon process. This overhead is due primarily to testing to see whether a user is a member of an implicitly defined user group. With the current implementation of implicit user grouping, even if the user and the group definition are in memory, a call must still be made to the LDAP server to determine group membership. We intend to remove this bottleneck in future versions and perform membership tests in memory.

## Future work

Our immediate plans are to enhance our authentication scheme for inclusion in the base infrastructure of the forthcoming release of WebSphere Commerce Business Edition. This version will be based on the Enterprise JavaBeans** technology and is written completely in the Java language. We also recognize that adding features such as role hierarchies and constraints between roles and enriching the language for using relationships would improve the usefulness of our implementation.[4]

In future work, we hope to more closely integrate with the Java 2 authorization model and augment it with appropriate portions of our policy-based authorization scheme. The Java 2 Platform, Enterprise

Table 2 Comparison of the steady-state overhead for performing authorization checks. Each command was run 1000 times, both with and without authorization checks. The times are mean response times from the point of view of a Web browser.

| Command | With Authorization Mean(s) | Without Authorization Mean(s) | Authorization Overhead (%) |
|---|---|---|---|
| UserDisplay | 0.242 | 0.240 | 0.8 |
| GroupDisplay | 0.166 | 0.165 | 0.6 |

Table 3 Comparison of initialization times with and without authorization

| Command | With Authorization Mean(s) | Without Authorization Mean(s) | Authorization Overhead (%) |
|---|---|---|---|
| System refresh | 0.87 | 0.24 | 72 |
| User login | 0.75 | 0.69 | 8 |

Edition (J2EE**) incorporates a role-based security model into the Java language.[14] J2EE separates the design of the security model of an application from the deployment in an operational environment. Using J2EE, the application developer defines the roles that are relevant to the application, as well as the permissions granted to each role. The definition of the security requirements are externalized in a document called a deployment descriptor. At deployment time, the deployer maps the security roles in the deployment descriptor to the user groups defined in the operational J2EE environment. J2EE access control is presently based only on the type of data being accessed. It is predicted that J2EE will address instance-level access control in future versions.[15]

## Conclusions

In this paper, we have described an application-level authorization scheme that uses implicit grouping and exploits relationships between users and resources to compactly express access control policies. Successful deployment of the implementation in WebSphere Commerce Suite, Marketplace Edition 4.1 has demonstrated the expressiveness of the authorization language, and our performance results indicate that it imposes minimal run-time overhead. The externalization of the authorization policies, and the adaptive user interface that enables only those functions actually available to the user, has resulted in a system that is easier to configure and customize, and therefore less costly to deploy.

## Acknowledgments

We wish to thank the entire WCS MPE development team and especially Dan Eaton for his help in gathering performance data.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

## Cited references and note

1. C. Bussler, "Policy Resolution in Workflow Management Systems," *Digital Technical Journal* **6**, No. 4 (Fall 1994).
2. R. C. Summers, *Secure Computing*, McGraw-Hill, New York (1997).
3. B. W. Lampson, "Protection," *Proceedings*, *5th Annual Princeton Conference on Information Sciences and Systems*, Princeton, NJ (March 25–26, 1971), pp. 437–443. Reprinted in *Operating Systems Review* **8**, No. 1, 18–24 (January 1974).
4. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *IEEE Computer* **29**, No. 2, 38–47 (February 1996).
5. J. Barkley and A. Cincotta, "Managing Role/Permission Relationships Using Object Access Types," *Proceedings*, *3rd ACM Workshop on Role-Based Access Control*, Fairfax, VA (October 22–23, 1998), available at http://hissa.ncsl.nist.gov/rbac/rgperms/rgperms.htm.
6. L. Giuri and P. Iglio, "Role Templates for Content-Based Access Control," *Proceedings*, *2nd ACM Workshop on Role-Based Access Control*, Fairfax, VA (October 28–29, 1999).
7. E. C. Cheng, "An Object-Oriented Organizational Model to Support Dynamic Role-Based Access Control in Electronic Commerce," *Decision Support Systems* **29**, 357–369 (2000).
8. J. F. Barkley, A. V. Cincotta, D. F. Ferraiolo, S. Favrilla, and D. R. Kuhn, "Role-Based Access Control for the World Wide Web," *Proceedings*, *20th National Information Systems Secur-*

*ity Conference*, Baltimore, MD (October 1997); can be downloaded from http://hissa.ncsl.nist.gov/rbac/rbacweb/paper.ps.
9. S. Na and S. Cheon, "Role Delegation in Role-Based Access Control," *Proceedings*, *5th ACM Workshop on Role-Based Access Control*, Berlin, Germany (July 26–28, 2000), pp. 39–44.
10. A. Lin and R. Brown, "The Application of Security Policy to Role-Based Access Control and the Common Data Security Architecture," *Computer Communications* **23**, 1584–1593 (2000).
11. R. S. Sandhu, V. Bhamidipati, and Q. Munawer, "The ARBAC97 Model for Role-Based Administration of Roles," *ACM Transactions on Information System Security* **2**, No. 1, 105–135 (February 1999).
12. This approach also does not require special treatment of administrative roles and the associated implementation machinery. See Reference 11 for an RBAC model that includes separate administrative roles.
13. B. S. Rubin, A. R. Christ, and K. A. Bohrer, "Java and the IBM San Francisco Project," *IBM Systems Journal* **37**, No. 3, 365–371 (1998).
14. S. Bodoff, D. Green, E. Jendrock, M. Pawlan, and B. Stearns, *The J2EE Tutorial*, can be downloaded from http://java.sun.com/j2ee/tutorial/download.html/.
15. B. Shannon, *Java 2 Platform, Enterprise Edition, Specification, v1.3*, Proposed Final Draft 3 (March 30, 2001); available at http://java.sun.com/j2ee/j2ee-1_3-pfd3-spec.pdf.

*Accepted for publication January 22, 2002.*

**Richard Goodwin** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: rgoodwin@us.ibm.com).* Dr. Goodwin received Ph.D. and M.S. degrees in computer science from Carnegie Mellon University in 1994 and 1996, respectively. He also holds a B.A.Sc. degree from the University of Waterloo, Ontario, Canada. Dr. Goodwin is currently manager of the intelligent e-marketplace research group at the Watson Research Center. For the past four years his work has focused on infrastructure for electronic commerce, including authentication and authorization. His research interests also include intelligent agents for planning, scheduling, and decision support.

**SweeFen Goh** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: sweefen@us.ibm.com).* Ms. Goh is a software engineer working in the e-commerce department at the Watson Research Center. She received her B.S. and M.S. degrees in computer science, with a minor in mathematics, from Kent State University, Ohio. Her current research interests are in the areas of mobile commerce and distributed authorization.

**Frederick Y. Wu** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: fywu@us.ibm.com).* Dr. Wu received the S.B. and Ph.D. degrees from MIT in 1973 and 1976, respectively. He is a research staff member in the Intelligent e-Marketplaces department at the Watson Research Center. For the past four years he has been active in the design and development of electronic commerce systems and technology components. Dr. Wu's recent experience includes the architecture and implementation of several large-scale marketplaces for IBM clients. His research interests are in the areas of access control systems and decision support tools for e-commerce participants.