8

Capability-based primitives for access control in object-oriented systems

John Hale
School of Electrical Engineering and Computer Science
Washington State University, Pullman, Washington 99164, USA
hale@eecs.wsu.edu

Jody Threet and Sujeet Shenoi
Department of Computer Science
University of Tulsa, Tulsa, Oklahoma 74104, USA
{threet, sujeet}@euler.mcs.utulsa.edu

Abstract

Access control is the cornerstone of information security and integrity, but the semantic diversity of object models makes it difficult to provide a common foundation for access control in object-oriented systems. This paper presents a primitive capability-based access control architecture that can model a variety of authorization policies. The architecture described is integrated at the meta-object level of the Meta-Object Operating System Environment, providing a common foundation for access control in heterogeneous object models.

Keywords

Access control, distributed object systems, capabilities, meta-object models

1 INTRODUCTION

Access control is critical to the security and integrity of distributed systems. While object-oriented technology has become a touchstone for developing distributed systems, the full potential of access control mechanisms for objects

Database Security XI T.Y. Lin and Shelly Qian (Eds)

has yet to be realized. Most advances in object-based access control have been confined to database security (see, e.g., Dittrich *et al.*, 1989; Keefe *et al.*, 1989; Jajodia and Kogan, 1990; Bruggemann, 1992; Thomas and Sandhu, 1993; Bertino *et al.*, 1994; Fernandez *et al.*, 1994; Jonscher and Dittrich, 1995).

Several factors have limited the incorporation of access control mechanisms in distributed object technology. Object models are heterogeneous with tremendous semantic diversity. Authorization policies, greatly influenced by the specific object models they are designed to protect, cannot be applied to other object models. Moreover, most access control mechanisms are brittle, incapable of supporting multiple policies.

Capabilities, which are unforgeable tokens that endow their possessors with privileges (Fabry, 1974; Karger, 1984, 1988), can implement a variety of authorization policies. The flexibility of capabilities suggests their use as a meta-model for access control of objects in distributed environments.

The Meta-Object Operating System Environment (MOOSE) (Hale *et al.*, 1997) provides a framework for developing verifiably secure heterogeneous distributed objects with a mix of formal methods and object technology. The MOOSE Meta-Object Model (MOM) decomposes object-oriented behavior into a few core mechanisms. The security mechanisms integrated into MOM can express a variety of authorization models for object-oriented systems.

This paper describes a capability-based access control architecture for meta-objects as a common foundation for security in heterogeneous distributed object systems. The paper begins with an overview of access control in object-oriented systems. The Meta-Object Model (MOM) and the access control architecture are described along with their roles in capturing heterogeneous access control models. The paper concludes with a discussion of related work.

2 ACCESS CONTROL OF OBJECTS

Object-oriented systems are composed of classes, instances, attributes (instance variables) and methods. These components support encapsulation, modularity and re-use through message-passing, inheritance and aggregation.

The goal of access control is to protect resources (objects) from unauthorized access by users (subjects). An *authorization state* is a function $State : (Subject \times Object \times Privilege) \Rightarrow Boolean$, where the subject's privilege is the access type. Often, the authorization state is represented as a list of tuples, e.g., $< Subject, \texttt{Object}, access\_type >$, declaring that `Subject` has the *access_type* privilege for `Object`. An access control model defines domains for subjects, privileges and objects. Also, it defines implicit authorization rules and commands to take systems from one authorization state to another.

While access control is fundamental to information security, authorization models in object-oriented database management systems (OODBMSs) (Keefe *et al.*, 1989; Rabitti *et al.*, 1991) lack a common perspective compared with those for relational database management systems. Many OODBMSs do not

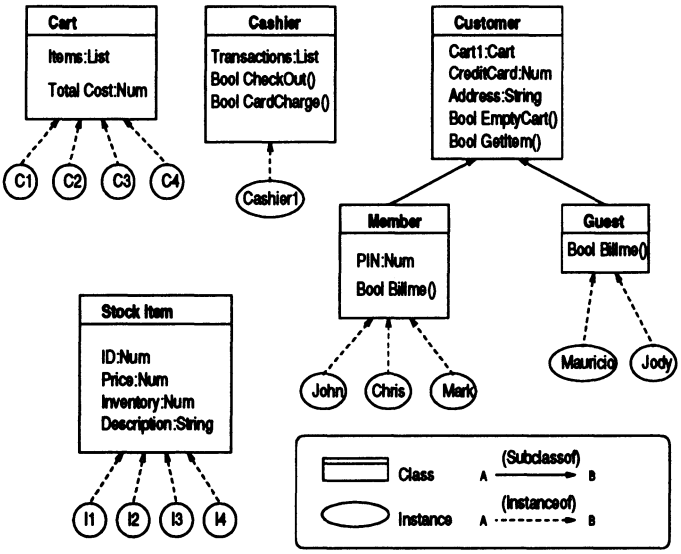

Figure 1 Object–oriented electronic commerce model.

provide any type of access control. The semantic diversity of object models is partly to blame, e.g., not all object models support multiple inheritance. The presence of competing authorization models also introduces problems. Each model resolves protection granularity, access types and implicit authorization flow in its own way. The dilemma is illustrated using a simple example.

Consider the object-oriented electronic commerce model in Figure 1. The protection granularity specifies the finest units to be protected. Classes, objects, attributes and methods are all viable atomic units for protection.

Objects could be chosen as the finest unit of protection. The resulting "all or nothing" access to objects is efficient (only one authorization set is needed per object), but inflexible. For instance, customers must be able to invoke `Cashier::CheckOut()` to pay for items. Using objects as atomic protection units implies that Customer objects would have access to Cashier objects and the potentially sensitive lists of Cashier transactions. The inflexibility of this approach forces most object models to respect individual methods and attributes as atomic units of protection, e.g., permitting customer access to `Cashier::CheckOut()`, but denying access to `Cashier::Transactions`.

Access types are another major issue in object-oriented systems. Database authorization models use *read* and *write* permissions. The privilege of modifying rights can introduce access types (e.g., *grant* and *revoke*). Often, this is a power implicitly held by object owners. Providing *grant* and *revoke* types means that the abilities to grant or revoke can themselves be access types (e.g., *grant-grant*). A general implementation of this requires a self-referential access type.

Suppose that cashiers need to read customer addresses, but should not modify them. Authorizing $<$ `Cashier1, Jody :: Address`, *read* $>$ only gives `Cashier1` read access to `Jody::Address`. If attributes can be read or writ-

ten directly (without using a local accessor method), access types *read* and *write* are desirable. However, if the model relies on local accessor methods to preserve encapsulation, then separate methods should exist for reading and writing attributes and effective read/write control can be manifested via the *execute* privilege on those accessor methods (Gal-Oz *et al.*, 1993).

Implicit authorization is a convenient way of propagating permissions and protections. The idea is that permissions/protections can be given to an entire class of subjects/objects using one authorization rule – this is most natural because the *class* concept is fundamental to object models. An instance can inherit its authorization status from its class just like it inherits methods and attributes. The rule < `Customer`, `StockItem` :: `Price`, *read* > gives all customers access to the prices of all items in stock.

Generating *exceptions* to authorization rules is possible using *negative authorizations* and *strong/weak authorizations*. A negative authorization explicitly denies access to an object (< `Chris`, `Cashier` :: `CheckOut`(), ¬*execute* > prohibits `Chris` from checking out). Combining positive and negative authorizations can lead to conflicts. Thus, authorizations may be derived by labeling rules as *weak* or *strong*. Weak rules cannot override strong ones. While the presence of such rules makes it difficult to derive an authorization rule for a particular event, the resulting authorization models are highly expressive.

Messages are the principal medium of communication in object-oriented systems. Jajodia and Kogan (1990) were the first to propose that messages be used as the focus of access control mechanisms in object systems. They introduced a *message filter* for deciding whether or not to accept a message on behalf of an object based on its source, content and destination. Message filters can reside within each object, providing ubiquitous access control. This decentralized authorization technique is superior to and more natural than centralized access control schemes for distributed object systems.

Our access control approach decomposes object systems into their most primitive components. Since message-passing is central to all meta-object models, the message filter can help unify access control in heterogeneous distributed object systems. The following section explores access control for meta-object models and presents a flexible authorization architecture that is easily integrated with existing object models.

3 META-OBJECT ACCESS CONTROL

This section describes a primitive foundation for access control in object-oriented systems. It is designed to be integrated at the meta-object level to permit a unified treatment of meta-classes, classes and objects (Stefik and Bobrow, 1985). The model is flexible enough to support multiple access control policies in distributed computing environments.

Meta-object models provide a common theoretical underpinning for object systems. They present a unified view of features such as classes, subclasses

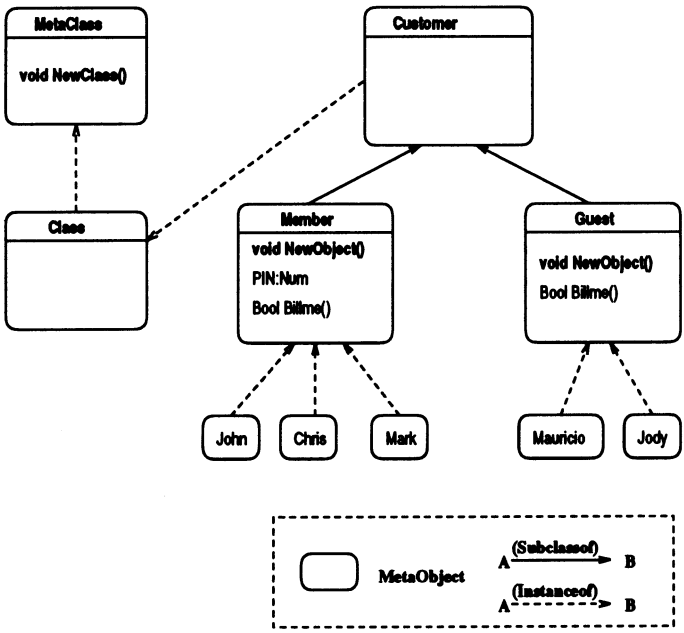

Figure 2 Meta-object electronic commerce model.

and inheritance using the more primitive notions of meta-objects and delegation. Meta-object models synthesize method invocation, message-passing and aggregation, the basic features of all object systems.

Figure 2 shows a meta-object representation of the electronic commerce model. Note that all classes and objects have been replaced by meta-objects. Meta-objects that can spawn other meta-objects (using method `NewObject()`) model classes. Meta-objects capable of spawning "class" meta-objects are called meta-classes. `MetaClass` and `Class` are both meta-classes.

This model uses capabilities (Fabry, 1974; Karger, 1984, 1988) for method-based access control of meta-objects. A capability is an unforgeable token that a subject uses as a ticket for object access. A ticket, which is also associated with an access type, can be held by a subject or for a subject by a trusted third party. The ticket is inspected by an object or by the trusted third party before access is granted. Alternatively, capabilities can be viewed as *locks* and *keys*; this view implies that objects must hold matching tokens. Each subject has keys that give access to objects with matching locks.

Capabilities typically control run-time privilege distribution between processes and subprocedures. They can be used to implement various authorization models, including identity and group-based Discretionary Access Control (DAC), Role-Based Access Control (RBAC) and Mandatory Access Control (MAC). This flexibility makes capabilities ideal for meta-level access control and conducive to supporting multipolicy functionality.

An authorization model must resolve (s, o, a) as true or false for every subject (s), object (o) and access type (a) given an authorization state. An au-

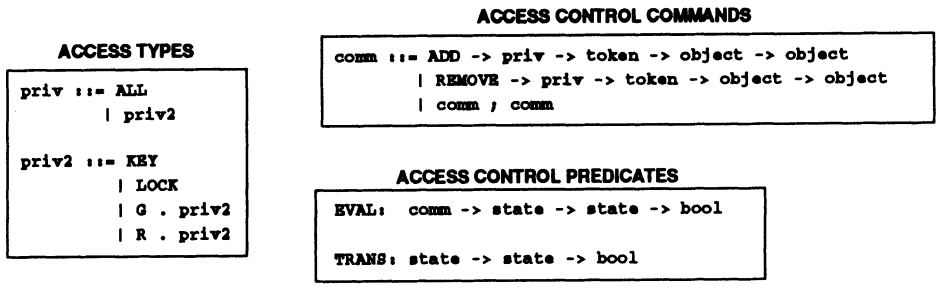

Figure 3 Access control definitions.

thorization state is defined by *State* : *Object* → *Privilege* → *Token* → *Bool* where *Token* serves as a representative for the subject.

The recursive definition of access types (Figure 3) implements general *grant* and *revoke* privileges. The definition permits access types such as `G. G. LOCK` (*grant grant* lock) and `G. R. G. KEY` (*grant revoke grant* key). These types can be thought of as pertaining to token lists. Note that every type other than `KEY` behaves as a lock, e.g., `G. R. KEY` is a type of lock that guards the `R. KEY` list. Any subject with `KEY` token matching a token associated with `G. R. KEY` in an object can add (*grant*) tokens to the `R. KEY` list.

The `ALL` privilege in Figure 3 confers privileges of every type to a subject. A subject with the `ALL` privilege can add or remove any type of authorization. The only limitation is that the subject must hold the actual token as a key. Rule 1 in Figure 4 formalizes the semantics of the `ALL` access type.

The command set given in Figure 3 enables alteration of the authorization state. It comprises commands for adding and removing authorization tuples. Subjects can only add or remove tokens that they hold as keys. This means that even when a subject has a *grant* or *revoke* permission on some access control list, the only tokens it is able to add or remove are those that it holds as keys. The third command allows atomic `ADD/REMOVE` command sequences.

Predicates `EVAL` and `TRANS` are defined on authorization states. `EVAL` is true when a command can take one state to another. It is used to give semantics to the command set. `TRANS` is true if a transition between states is possible. The relationship between `EVAL` and `TRANS` is formalized by Rule 2.

Rule 3 gives semantics to the `ADD` command. It states that a subject must have *grant* privilege over an access type in an object to add a token of that type to the object. The constraint that subjects can only add tokens held by them as keys is formalized. E.g., authorization tuples $< \mathtt{o}, \mathtt{G.R.LOCK}, \mathtt{a} >$, $< \mathtt{s}, \mathtt{KEY}, \mathtt{a} >$ and $< \mathtt{s}, \mathtt{KEY}, \mathtt{b} >$ allow the command `ADD R.LOCK b o s`.

The semantics for the `REMOVE` command (Rule 4) specify when it is legal for a subject to remove authorization tuples. Using the previous example, an additional authorization tuple $< \mathtt{o}, \mathtt{R.R.LOCK}, \mathtt{b} >$ would let `s` remove `R.LOCK` permissions from `o`. Again, `s` must hold the affected token as a key.

Rule 5 introduces command sequences to the system. It formalizes the transitive nature of commands on authorization states.

Rule 1 : $\forall p : priv,\ o : obj,\ t : token,\ s : state.\quad s\ o\ \mathbf{ALL}\ t \Rightarrow s\ o\ p\ t$

Rule 2 : $\forall s_1, s_2 . \exists c : comm.\quad \mathbf{EVAL}\ c\ s_1\ s_2 \Rightarrow \mathbf{TRANS}\ s_1\ s_2$

Rule 3 : $\forall p, s_1, s_2, o_1, o_2, t.$

$s_1\ o_1\ \mathbf{KEY}\ t \wedge s_1\ o_2\ \mathbf{G.p}\ t \Rightarrow\ (s_2\ o_2\ p\ t \wedge (\forall o', p', t'.\ o' \neq o_2 \vee p' \neq p$

$\vee\ t' \neq t \Rightarrow s_1\ o'\ p'\ t' = s_2\ o'\ p'\ t') \Rightarrow\ \mathbf{EVAL}\ (\mathbf{ADD}\ p\ t\ o_2\ o_1)\ s_1\ s_2)$

Rule 4 : $\forall p, s_1, s_2, o_1, o_2, t.$

$s_1\ o_1\ \mathbf{KEY}\ t \wedge s_1\ o_2\ \mathbf{R.p}\ t \Rightarrow\ \neg(s_2\ o_2\ p\ t \wedge (\forall o', p', t'.\ o' \neq o_2 \vee p' \neq p$

$\vee\ t' \neq t \Rightarrow s_1\ o'\ p'\ t' = s_2\ o'\ p'\ t') \Rightarrow\ \mathbf{EVAL}\ (\mathbf{REMOVE}\ p\ t\ o_2\ o_1)\ s_1\ s_2)$

Rule 5 : $\mathbf{EVAL}\ (c_1)\ s_1\ s_2 \wedge\ \mathbf{EVAL}\ (c_2)\ s_2\ s_3\ \Rightarrow \mathbf{EVAL}\ (c_1; c_2)\ s_1\ s_3$

Figure 4. Access control rules.

4 AUTHORIZATION IN THE META-OBJECT MODEL

The Meta-Object Model (MOM) is a core model for the design and analysis of distributed object systems. MOM is augmented with mechanisms supporting the implementation and analysis of a spectrum of authorization models for object-oriented systems. This section describes MOM and the integration of capability-based access control primitives in MOM.

MOM is constructed with the Robust Object Calculus (ROC), a process calculus for distributed objects (Hale *et al.*, 1997). ROC's features, such as encapsulation and tuple-based communication, have facilitated the formal design of MOM. MOM is influenced by ACTORS (Agha, 1986) and LOOPS (Stefik and Bobrow, 1985). It models core object behavior, including persistence, method invocation, asynchronous message-passing, delegation and aggregation. Virtually any object model can be built from MOM.

Access control policies are implemented in MOM systems with object access control lists (OACLs) and message filters. These mechanisms implement flexible and ubiquitous access control for objects and methods. Objects with OACLs and message filters can provide authorization services to domains of subobjects when efficiency concerns outweigh security requirements.

4.1 MOM Objects

MOM objects are collections of tightly encapsulated components (processes). Each MOM object uses a navigational identifier (nid) set that defines how it is addressed. MOM components that share a nid are part of the same object.

A MOM system has a hierarchical structure of object domains, specifying objects that contain objects. Each MOM system has one *root* object that is not contained by any other object. Each object is named by a local identifier (lid) unique to its domain. Nids are constructed from (lid, direction) pair sequences. The domain of an object is its parent object.

The local identifier of an object (say Obj_1) is prepended to the global identifier (gid) of its parent (say **root**) to define a unique gid for the object, e.g., [[**out**, Obj_1#]#, [[**out**, **root**#]#, **null**#]#]. Objects contain several cooperating MOM components: an *object registry*, a *message handler* and *methods*. The message handler and the object registry form the basis for an object's identity and control communications. Furthermore, objects can house a message filter and an *object access control list* (OACL), which contains authorizations for access to the object's components. The message filter resides in the message handler and authorizes each message using the OACL.

An object registry maintains a record for each component within an object. Specifically, it keeps track of message handlers, method interfaces and active method invocations. Component lids and types are stored within registry records. As an object is being deleted, it must refer to its registry to gracefully delete each of its components. Furthermore, to create an object inside a parent, the registry is checked to see that the new lid is unique. Only then is the object created and registered with the parent object.

4.2 MOM Message-Passing

Messages embody asynchronous communication in MOM. They carry requests, acknowledgements and replies. A message handler processes incoming MOM messages and marshals object requests. It controls the distribution of requests and replies for MOM components. An incoming message can be received as a local request or delegated to another object. A message is delegated by consuming it and then re-creating it in an adjacent domain.

4.3 MOM Methods

The method architecture incorporates three types of processes: *method interfaces*, *method arbiters* and *method bodies*. Method interface components accept invocation requests propagated by message handlers, and control activation and synchronization for individual methods. A unique method interface exists for each method in an object. A method interface spawns a method arbiter and method body upon receiving a method invocation request. The method body performs the actual computation, while the method arbiter negotiates communications between the method body and the outside world.

MOM includes *mutable* and *immutable* methods. A mutable method invocation spawns a persistent process with state that can be accessed many times. Immutable methods, on the other hand, behave like traditional methods, terminating and returning a value upon completion. While mutable methods model instance variables, it is a MOM standard to create (immutable) accessor methods in objects for instance variables.

Requests to an immutable method spawn a new method arbiter and body to support concurrent method invocation. However, a mutable method interface does not permit co-existing invocations. Requests to an active mutable method are forwarded to the method arbiter. Requests for a mutable method are forwarded to its arbiter which then forwards them to the method body. The behavior of the method body can be affected by previous requests; this models state information.

Immutable methods behave like methods in conventional object-oriented programming languages. Each request spawns a new method body supporting concurrent method invocation. The completion of an invocation results in a reply from the method body to the method arbiter. This reply can be propagated back to the initiating object (modeling a traditional method invocation) or it can go elsewhere as indicated by the request.

Component creation and deletion are handled by special methods resident in each object. Objects invoke these methods like they would any other method. For example, a foreign object might issue a request to another object to create a subobject. However, this request might be refused if a conflict exists in the object registry (e.g., a subobject of the same name exists) or even as a matter of policy (e.g., only ancestors can delete object components).

4.4 MOM Security

OACLs and message filters implement access control in MOM systems. Each object can contain an OACL and a message filter. An OACL is a list of authorization tuples of the form < **component**, **access_type**, **token** >. Message filters use OACLs to authorize incoming messages. A message includes information regarding the type of access and a set of tokens provided by the message originator that are used as keys in the authorization process.

Method-based access control is specified by tuples with method names in the component field. Authorization commands, e.g., `G. KEY`, can also be issued in messages to destination objects. Since filtration occurs at the destination and at objects along the message route, the delegation of messages between objects must be authorized. This feature protects entire objects.

A message filter examines a message and refers to its OACL to determine authorization. If the message contains a key matching a lock held by the intended component recipient, the message is authorized. This scheme is adequate for object-oriented programming languages and is well-suited to object-oriented databases and distributed object systems. Figure 5 shows MOM object components with the OACL and message filter.

MOM objects affect access control by introducing intervening filters. Suppose object s has legitimate access to object o because it has a key matching one of o's locks. Object s could be denied access to o if an intervening object i is on the message path between s and o. This can occur because messages

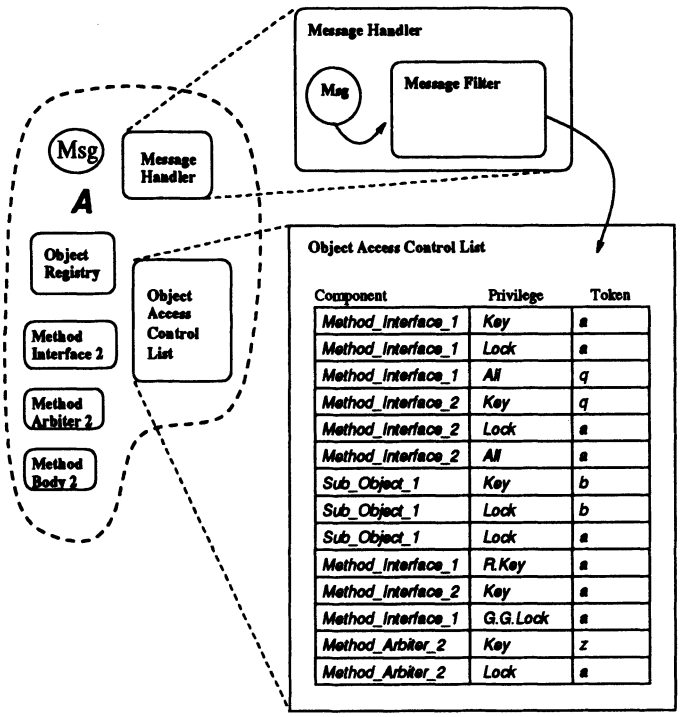

Figure 5 OACL and message filter components.

must be authorized to pass through each domain between source and destination. If s is not allowed to send messages to i, the message will be returned at that point, effectively denying access to o. Intervening objects complicate the authorization architecture, but facilitate specialization of authorizations.

The root object plays an important role by creating meta-objects and initializing the authorization state. Classes are meta-objects with special methods that construct instance objects of a certain type. Authorizations can be inherited by instances or subclasses via token propagation and runtime delegation. These techniques manifest implicit authorization flow. User-controlled objects manifest explicit authorization at runtime by invoking methods containing authorization commands.

Figure 6 shows the MOM version of the electronic commerce model. It is complemented by a partial view of the virtual global OACL that shows the authorization state after the bootstrapping process (Figure 7).

No customer should access the information of another customer. This is enforced by ensuring that no **Customer** instance has a key to a lock in the OACL of the root object (which contains all **Customer** instances). Figure 6 shows the result of **John** trying to access **Jody**. Access to **Jody** and its subobjects is denied because the filter in **root** intervenes. The filter checks the OACL in **root** and discovers that **John** does not have permission to send messages to (or through) **Jody**.

It is important that customers read, but not modify, stock item prices.

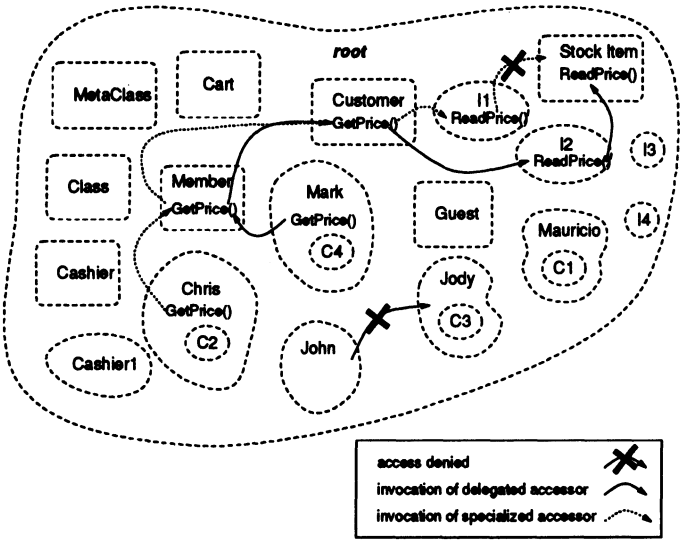

Figure 6 Electronic commerce model (MOM representation).

Therefore, tuple < `ReadPrice()`, `LOCK`, `read_price` > is placed in the OACL for the meta-object (class) `Stock Item`. Customer access to prices through `Customer::ReadPrice()` requires the OACL for meta-object (class) `Customer` to contain < `ReadPrice()`, `KEY`, `read_price` >. The authorizations to add these tuples are given by the root object in the bootstrapping process.

Delegations must be authorized like method invocations. Meta-objects propagate delegation authorizations to instances, e.g., `Stock Item` must propagate < `ReadPrice()`, `LOCK`, `read_price` > to each of its instances and/or subclasses. Propagation is performed in the constructor for `Stock Item`.

When a `Customer` reads a price, it delegates to `Customer::ReadPrice()` which has the key to stock item prices. Figure 6 shows the chain when `Mark::GetPrice()` is invoked for `I2`. At the other end of this invocation request, `Customer::ReadPrice()` calls an accessor function local to `I2`, which is then delegated to an accessor method in `Stock Item`.

Authorizations for method-based access can be specialized like methods. The authorization to delegate is held by the delegating object (instance or subclass), facilitating its specialization. For example, the accessor method in `I1` could choose not to delegate to its parent class, but instead define its own behavior and/or authorization set. This is shown in Figure 6 where `Chris` attempts to get the price of `I1`, but is denied because the OACL for `I1` does not contain the authorization tuple < `ReadPrice()`, `LOCK`, `read_price` > for delegation to the parent class (Figure 6).

Efficiency is a concern whenever message filters are used. The proposed architecture permits objects to provide authorization services for entire domains. This obviates the use of message filters and OACLs in each object in MOM, resulting in a flexible and potentially lightweight access control system.

Object	Component	Privilege	Token
Stock Item	*MsgHandler*	*LOCK*	*cashier*
I2	*MsgHandler*	*LOCK*	*cashier*
I3	*MsgHandler*	*LOCK*	*cashier*
I4	*MsgHandler*	*LOCK*	*cashier*
Stock Item	*ReadPrice()*	*LOCK*	*read_price*
I2	*ReadPrice()*	*LOCK*	*read_price*
I3	*ReadPrice()*	*LOCK*	*read_price*
I4	*ReadPrice()*	*LOCK*	*read_price*
root	*Customer*	*LOCK*	*cashier*
root	*Stock Item*	*LOCK*	*cashier*
root	*Stock Item*	*LOCK*	*customer*
Customer	*MsgHandler*	*KEY*	*customer*
John	*MsgHandler*	*KEY*	*customer*
Customer	*ReadPrice()*	*KEY*	*read_price*
John	*ReadPrice()*	*KEY*	*read_price*
Customer	*MsgHandler*	*LOCK*	*cashier*
John	*MsgHandler*	*LOCK*	*cashier*

Figure 7 Global view of OACLs.

5 RELATED WORK

This work is motivated by the sophisticated meta-object models of LOOPS (Stefik and Bobrow, 1985) and ACTORS (Agha, 1986). The access control mechanisms in MOM support multipolicy access control in heterogeneous object systems. The meta-level authorization scheme is geared for distributed objects and relies on three concepts: capabilities, message filters and method-based access control. Integrating these features in a meta-object model produces a rich framework for expressing a variety of authorization models for object-oriented systems.

Research in database security has also influenced this work (Dittrich *et al.*, 1989; Thuraisingham, 1989; Thomas and Sandhu, 1993; Fernandez *et al.*, 1994; Demurjian *et al.*, 1995). The ORION/ITASCA system adopts discretionary access control for objects, embracing notions of explicit/implicit, positive/negative and weak/strong authorizations (Rabitti *et al.*, 1991). An extension by Bertino *et al.* (1994) supports additional access types and type dependency modeling, clarifies subject group semantics, and considers object versions and the potential for distributed authorization control. Our model can be extended to positive/negative and weak/strong authorizations by modifying components to handle the new types. An important advantage of our model is that semantic-based forms of implicit authorization emerge in any system designed using MOM.

Gal-Oz *et al.* (1993) introduced method–based access control for objects. By using methods as a basis for access control, first-order access types (e.g., *read* and *write*) are reduced to a single *execute* type. This scheme complements the

MOM architecture nicely because MOM stipulates that any form of access always occurs through a method invocation (accessor methods are used to read and write instance variables). This facilitates policy specifications that are emergent from meta-level access control primitives and MOM.

Multipolicy access control is an important area of research (Bell, 1994; Bertino *et al.*, 1996). Multipolicy mechanisms and models enable users to protect each object according to a different policy. The architecture described by Bertino *et al.* (1996) employs flexible access control mechanisms and mediators (Wiederhold, 1992). Mediators shape access control mechanisms to enforce user-specified authorization policies. Our work in multipolicy systems seeks a common ground for access control mechanisms that can support the interoperation of disparate authorization policies.

Argos (Jonscher and Dittrich, 1995) shares the goal of developing a unified view of heterogeneous access control models in open distributed environments. It achieves this goal by incorporating features of various identity-based authorization models. It models implicit authorization flow and introduces domains to generate classes of behavior and protection, exploiting the richness of the object-oriented paradigm to create a flexible system. Our approach differs by decomposing object behavior into primitive mechanisms. Capabilities are also more general – they can be identities, groups, labels, roles or tasks.

The Distributed Computing Environment (DCE) uses a decentralized authorization service in its security architecture (Rosenberry *et al.*, 1993). DCE's authorization service associates access control lists with servers, files and records, specifying legal operations for each user. The authorization service works in concert with DCE's authentication service. Privilege attributes are embedded in tickets provided to subjects by the authentication server at login. ACL managers that reside on each server authorize access requests. DCE supports a variety of authorization models by allowing developer customization of ACL managers. DCE does not deal directly with object-oriented and multipolicy access control issues. However, it provides a framework for ubiquitous, yet practical, access control in distributed systems.

6 CONCLUSIONS

The meta-level authorization service architecture presented in this paper integrates primitive capability-based access control mechanisms within a meta-object model for maximal support of multiple policies in heterogeneous object systems. The meta-object model engages message filters and method-based access control, although access control for objects is also possible. Access control in this model can be ubiquitous, where each object is responsible for its own authorization policy. It can also be lightweight, where objects provide authorization services for entire object domains. Implemented in the Meta-Object Model (MOM), this authorization service architecture provides a common foundation for the secure interoperation of heterogeneous distributed objects.

Acknowledgement This research was supported by MPO Grants MDA904-94-C-6117 and MDA904-96-1-0115 and OCAST Grant AR2-002.

REFERENCES

Agha, G.A. (1986) *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts.

Bell, D. (1994) Modeling the multipolicy machine. *Proceedings of the New Security Paradigms Workshop*, 2–9.

Bertino, E., Jajodia, S. and Samarati, P. (1996) Supporting multiple access control policies in database systems. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 94–109.

Bertino, E., Origgi, F. and Samarati, P. (1994) A new authorization model for object-oriented databases, in *Database Security, VIII: Status and Prospects* (eds. J. Biskup *et al.*), Elsevier, Amsterdam, 199–222.

Bruggemann, H.H. (1992) Rights in an object-oriented environment, in *Database Security, V: Status and Prospects* (eds. C. Landwehr and S. Jajodia), Elsevier, Amsterdam, 99–115.

Demurjian, S., Daggett, T., Ting, T.C. and Hu, M. (1995) URBS enforcement mechanisms for object-oriented systems, in *Database Security, IX: Status and Prospects* (eds. D. Spooner *et al.*), Chapman and Hall, London, 79–94.

Dittrich, K., Hartig, M. and Pfefferle, H. (1989) Discretionary access control in structurally object-oriented databases, in *Database Security, II: Status and Prospects* (ed. C. Landwehr), Elsevier, Amsterdam, 105–121.

Fabry, R. (1974) Capability-based addressing. *Communications of the ACM*, **17(7)**, 403–412.

Fernandez, E.B., Wu, J. and Fernandez, M.H. (1994) User group structures in object-oriented database authorization, in *Database Security, VIII: Status and Prospects* (eds. J. Biskup *et al.*), Elsevier, Amsterdam, 57–76.

Gal-Oz, N., Gudes, E. and Fernandez, E.B. (1993) A model of methods access authorization in object-oriented databases. *Proceedings of the 19th International Conference on Very Large Databases*, 52–61.

Hale, J., Threet, J. and Shenoi, S. (1997) A framework for high assurance security of distributed objects, in *Database Security, X: Status and Prospects* (eds. P. Samarati and R. Sandhu), Chapman and Hall, London, 99–115.

Jajodia, S. and Kogan, B. (1990) Integrating an object-oriented data model with multilevel security. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 76–85.

Jonscher, D. and Dittrich, K.R. (1995) Argos – A configurable access control system for interoperable environments, in *Database Security, IX: Status and Prospects* (eds. D. Spooner *et al.*), Chapman and Hall, London, 43–60.

Karger, P. (1984) An augmented capability architecture to support lattice security. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 2–12.

Karger, P. (1988) Implementing commercial data integrity with secure capabilities. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 130–139.

Keefe, T.F., Tsai, W.T. and Thuraisingham, M.B. (1989) Soda: A secure object-oriented database system. *Computers & Security*, **8(6)**, 517–533.

Rabitti, F., Bertino, E., Kim, W. and Woelk, D. (1991) A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, **16(1)**, 88–133.

Rosenberry, W., Kenney, D. and Fisher, G. (1993) *Understanding DCE*. O'Reilly and Associates, Inc., Sebastopal, California.

Stefik, M. and Bobrow, D.G. (1985) Object-oriented programming: Themes and variations. *AI Magazine*, **6(4)**, 40–62.

Thomas, R.K. and Sandhu, R. (1993) Discretionary access control in object-oriented databases: Issues and research directions. *Proceedings of the Sixteenth National Computer Security Conference*, 63–74.

Thuraisingham, M.B. (1989) Mandatory security in object-oriented database systems. *ACM SIGPLAN Notices*, **24(10)**, 203–210.

Wiederhold, G. (1992) Mediators in the architecture of future information systems: A new approach. *IEEE Computer*, **25(3)**, 38–49.

7 BIOGRAPHY

John Hale is an assistant professor in the School of Electrical Engineering and Computer Science at Washington State University. His research interests are database security, heterogeneous distributed systems, object-oriented systems and computer graphics. Dr. Hale received his B.S., M.S. and Ph.D. degrees in computer science from the University of Tulsa in 1990, 1992 and 1997, respectively.

Jody Threet is vice president and founding partner of Sleek Software, Inc., Austin, Texas. He received his B.S. in mathematics and his M.S. and Ph.D. degrees in computer science from the University of Tulsa in 1990, 1993 and 1997, respectively. Dr. Threet's research areas are distributed systems, intelligent tutoring systems and formal methods.

Sujeet Shenoi is an associate professor of computer science at the University of Tulsa. He received his B.Tech. degree from the Indian Institute of Technology, Bombay in 1981, and his M.S. (Ch.E.), M.S. (CS) and Ph.D. degrees from Kansas State University in 1984, 1987 and 1989, respectively. Dr. Shenoi's primary research interests are in database systems, artificial intelligence and intelligent control.