7 PROGRAMMABLE SECURITY FOR OBJECT-ORIENTED SYSTEMS

John Hale, Mauricio Papa, and Sujeet Shenoi

Abstract: This paper focuses on "programmable security" for object-oriented systems and languages. A primitive distributed object model is used to capture the essence of object behavior and access control schemes. This model can be used to construct virtually any distributed object language or system while supporting a spectrum of decentralized authorization models.

7.1 INTRODUCTION

High assurance security is crucial to deploying distributed object systems in mission-critical applications. Unfortunately, most software architectures and tools require designers to implement security from scratch. Such *ad hoc* solutions often fail in open distributed environments (Jonscher and Dittrich, 1995).

One solution is to integrate primitive security mechanisms and constructs for "programming" and "verifying" security within distributed object languages and architectures. This approach is similar to the incorporation of primitive data types, type constructors and type-checking in traditional programming languages. Like strong typing, providing and checking security at the language level will significantly improve code reliability.

This paper illustrates programmable security for object systems using security mechanisms embedded in a primitive distributed object model. The essence of object behavior and a flexible ticket-based access control scheme are incorporated in the primitive model. All object and security functionalities are captured and articulated using meta objects. This facilitates the construction of virtually any distributed object language or system – even C++, Java and CORBA – while supporting a spectrum of decentralized authorization models.

S. Jajodia (ed.), *Database Security XII*

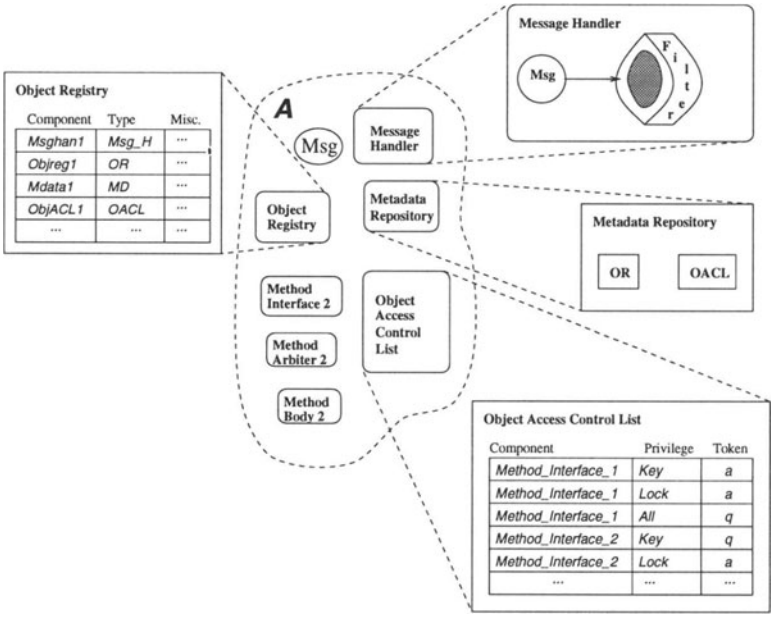

Figure 1. MOM object components.

7.2 META OBJECT MODEL

The Meta Object Model (MOM), as presented by Hale *et al.* (1998), is a core distributed object model designed for building object systems and languages.

7.2.1 MOM Objects

MOM objects (Figure 1) comprise: (i) a message handler, (ii) three information repositories: object registry, metadata repository and object access control list (OACL), and (iii) object contents (methods and subobjects).

Message handlers delegate and process messages, at times interacting with method interfaces (for method invocation requests) and method arbiters (for method replies). Message handlers constrain the set of messages an object will accept from its immediate environment (domain). The addressing scheme for messages is based on MOM identifiers (local ids: *lids* and global ids: *gids*). The MOM authorization model employs tickets for access control. Message handlers contain message filters that provide access control by further constraining the set of accepted messages based on the embedded tickets and the local authorization state. The message filter in the message handler of an object authorizes messages by referring to the OACL for the local authorization state (set of prevailing locks) of the object.

Each MOM object also maintains an object registry with bookkeeping information (*lid*, component type, etc.) about each object component. Object registries are mainly used to avoid conflicts when creating and deleting objects.

Metadata repositories provide meta objects with templates for creating objects, manifesting the emergent object-oriented behavior of classes and metaclasses.

7.2.2 Message-Passing and Methods

MOM messages are processes that persist until they are consumed. They carry method invocation or authorization requests, acknowledgements and replies.

Message handlers accept or reject messages and marshal object requests. They also control the distribution of requests and replies. An incoming message can be received as a local request or it can be delegated to another object in an adjacent domain.

MOM's method architecture has three components: (i) method interfaces, (ii) method arbiters and (iii) method bodies. Each method uses a distinct method interface to accept method invocation requests and manifest synchronization constraints. A method interface spawns a method arbiter and method body upon acceptance of a method invocation. Method arbiters negotiate communication between method bodies and their environments.

Methods can be mutable or immutable. Mutable methods model instance variables by creating processes with state that can be accessed multiple times. Immutable methods are stateless methods that terminate and return values. They often serve as interfaces to instance variables (accessor methods).

Each immutable method invocation produces a distinct method arbiter. On the other hand, mutable method interfaces prohibit the creation of new method arbiters until the previous arbiters have terminated. Requests to an active mutable method are forwarded to the active arbiter.

Immutable method interfaces create new method arbiters and bodies for each request. An arbiter passes arguments to its method body and waits for requests from the method body and/or replies from methods invoked by the body; it formulates a reply message at the termination of the invocation.

Component creation and deletion are handled by special methods that refer to object registries to prevent naming conflicts and maintain consistency. Metadata repositories are queried for the structures of new objects.

7.2.3 Security in MOM

Message filters (Jajodia and Kogan, 1990) in MOM message handlers accept messages by comparing embedded tickets with the local authorization states of objects. The local authorization state of an object defines a set of ticket-based permissions that are recorded in its OACL. These two mechanisms permit the implementation of a variety of authorization models for secure object systems.

OACLs are local information repositories with tuples defining object authorizations. The tuple <`Comp`,`Priv`,`Tok`> specifies that component `Comp` associates privilege `Priv` with token `Tok`. E.g., <`Interface1`,`lock`,`a`> states that `Interface1` has a `lock` associated with token `a`, i.e., `Interface1` is accessible by all messages holding `a` tickets. Component tickets are also held in OACLs. The tuple <`Arbiter2`,`key`,`b`> states that `b` is a ticket held by `Arbiter2`.

ACCESS TYPES

```
priv ::= ALL
       | priv'

priv' ::= KEY
        | LOCK
        | GRANT . priv'
        | REVOKE . priv'
```

ACCESS CONTROL COMMANDS

```
comm ::= ADD priv token object object
       | REMOVE priv token object object
       | comm ; comm
```

ACCESS CONTROL PREDICATES

```
EVAL:  comm -> state -> state -> bool

TRANS: state -> state -> bool
```

Figure 7.1 Figure 2. Access control definitions.

7.2.3.1 Ticket-Based Security. Tickets are unforgeable tokens visible only to trusted processes (message filters and OACLs). Untrusted processes (messages and methods) may carry and pass tickets. Methods can request that new tickets be created, but they cannot forge old ones.

Ticket distribution and revocation are thorny issues (Gilgor *et al.*, 1987). A naive distribution policy allows owners to distribute tickets. Another might permit ticket holders to distribute tickets (under certain circumstances). Ticket revocation is more complicated, particularly for capabilities where revocation must be partial, selective and transitive.

MOM provides mechanisms for adding and removing tickets from OACLs, but these cannot ensure that the critical properties of revocation and distribution are respected. Constraints may be built on top of MOM to enforce distribution/revocation policies (or implement access control models). This is accomplished using structured tickets and reconfiguring message filters.

7.2.3.2 Authorization Model. MOM uses a ticket-based scheme for its authorization model. The authorization model resolves (s,o,a) tuples as TRUE or FALSE for subjects s, objects o and access types a. An authorization state is defined by $State : Object \rightarrow Privilege \rightarrow Token \rightarrow Bool$ where $Token$ is an atomic ticket representing the subject. MOM tickets (or keys) are unforgeable tokens that are embedded in messages. The dual of a key is a lock, which is associated with a token and a privilege type. Locks define the authorization state of an object by specifying ticket-based permissions on object components.

Figure 2 defines MOM's model of *grant* and *revoke* privileges. The model permits access types such as GRANT.LOCK and GRANT.REVOKE.KEY. Every type other than KEY behaves as a lock. E.g., GRANT.REVOKE.KEY is a lock that applies to REVOKE.KEY privileges. A subject with a key matching the GRANT.REVOKE.KEY lock held by an object can add (*grant*) tokens to REVOKE.KEY list.

The ALL privilege confers all privileges to a subject. A subject holding a key matching a lock ALL held by a component has complete access to that

Rule 1 : $\forall p : priv,\ o : obj,\ t : token,\ s : state;\ s\ o\ \texttt{ALL}\ t \Rightarrow s\ o\ p\ t$

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

Rule 3 : $\forall p, s_1, s_2, o_1, o_2, t;\ s_1\ o_1\ \texttt{KEY}\ t \wedge s_1\ o_2\ \texttt{GRANT.p}\ t \Rightarrow$
$(s_2\ o_2\ p\ t \wedge (\forall o', p', t'.\ o' \neq o_2 \vee p' \neq p$
$\vee\ t' \neq t \Rightarrow s_1\ o'\ p'\ t' = s_2\ o'\ p'\ t') \Rightarrow \texttt{EVAL}\ (\texttt{ADD}\ p\ t\ o_2\ o_1)\ s_1\ s_2)$

Rule 4 : $\forall p, s_1, s_2, o_1, o_2, t;\ s_1\ o_1\ \texttt{KEY}\ t \wedge s_1\ o_2\ \texttt{REVOKE.p}\ t \Rightarrow$
$\neg(s_2\ o_2\ p\ t \wedge (\forall o', p', t'.\ o' \neq o_2 \vee p' \neq p$
$\vee\ t' \neq t \Rightarrow s_1\ o'\ p'\ t' = s_2\ o'\ p'\ t') \Rightarrow \texttt{EVAL}\ (\texttt{REMOVE}\ p\ t\ o_2\ o_1)\ s_1\ s_2)$

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

Figure 3. Authorization semantics.

component. The authorization state of an object can be modified by adding or removing ticket–privilege associations in its OACL.

The command set in Figure 2 permits dynamic and explicit authorization state modification. Commands can be embedded in messages as authorization requests. Subjects can add or remove ticket-privilege associations in objects for the tokens they hold as keys. Command sequences are permitted in messages.

Figure 3 provides the authorization model semantics. Rule 1 defines the ALL access type. Rule 2 formalizes the predicates EVAL and TRANS in Figure 2. Note that EVAL returns TRUE when a command will take one state to another while TRANS returns TRUE if a transition between states is possible. Rule 3 defines the ADD command. A subject must have *grant* privilege over an access type in an object to add a token of that type to the object. It also stipulates that subjects can only add tokens held by them as keys. Rule 4 defines the REMOVE command. It specifies when it is legal for a subject to remove authorization tuples. Rule 5 formalizes the transitive nature of commands.

7.2.3.3 Ticket-Based Access Control. Tickets, message filters and OACLs provide access control in MOM systems. Meta objects can own OACLs with local authorization state information. Each OACL contains authorization tuples of the form <component, access_type, token>. (Tuples with method names in component fields implement method based access control (MBAC) (Gal-Oz *et al.*, 1993).) Messages carry the access type, e.g., GRANT.LOCK, and tokens (tickets) owned by the subject. Message filters authorize messages against OACLs. Messages are authorized if they contain keys (tickets) that match locks held by the intended recipients. Message filtering can occur at all objects along the message route. However, performance can be improved by placing message filters only in strategic objects.

Meta objects refer to metadata repositories to define the initial authorization states for objects they create. Classes are meta objects with methods for constructing instances. Authorizations can be inherited by instances/subclasses by token propagation and message delegation. Access to instance variables is controlled by instances and defined by token propagation at instance creation.

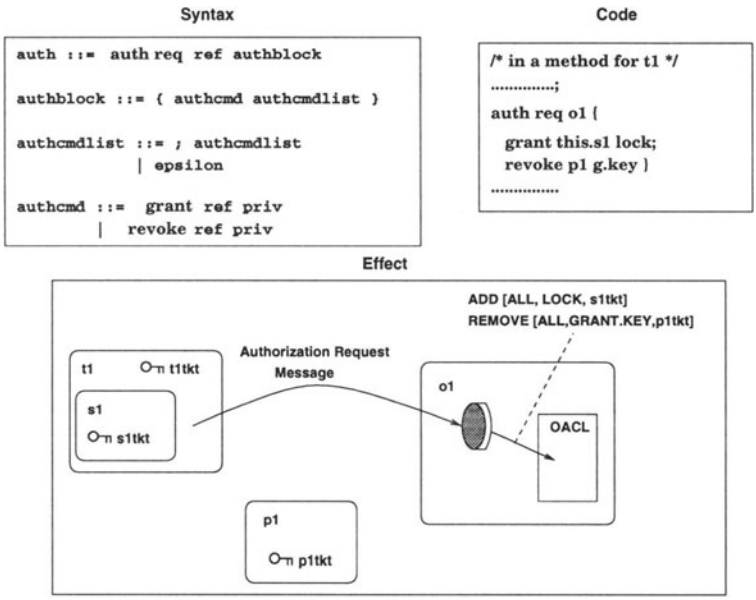

Figure 4. Discretionary access control.

Access to methods can be controlled by classes if invocations are to be delegated from instances to classes. This implements implicit authorization flow.

Objects manifest explicit authorization flow by invoking methods containing authorization commands. Authorization commands issued in messages can modify the OACLs of destination objects.

7.3 PROGRAMMING ACCESS CONTROL

This section shows how MOM's ticket-based scheme is used to implement various access control models.

7.3.1 Discretionary access control

Discretionary access control (DAC) is based on subject identity. It gives the owner of a resource the authority to grant or deny access to the resource. Standard permissions are *read*, *write* and *execute*, but *grant* and *revoke* permissions can be included. MOM models these as permissions to execute methods and send messages. Authorization states can be modified using authorization commands embedded in messages.

Figure 4 shows syntactic constructs for issuing authorization requests within methods. Implementing DAC involves mapping tickets to identities. Therefore, an authorization request specifies a reference to the object at which the request is directed. Simple authorization commands in the request tell the object to add

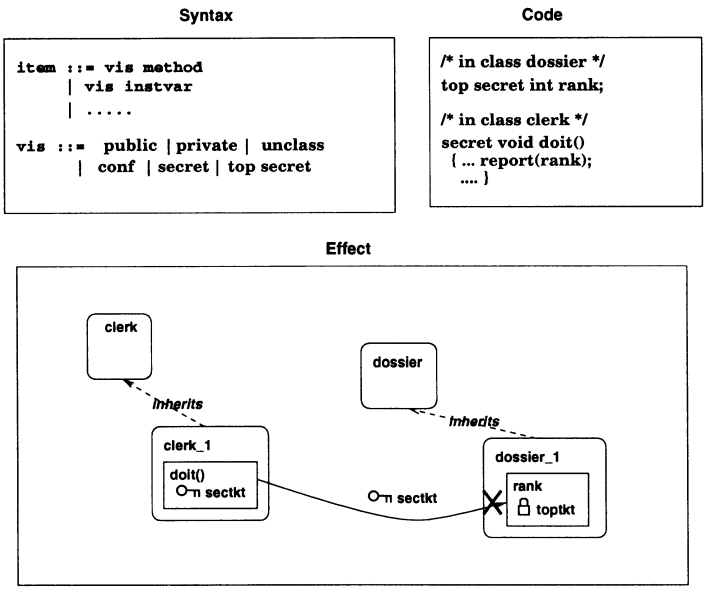

Figure 5. Mandatory access control.

or remove OACL entries. Ticket names, which must reside in each command, are mapped from another reference to an object.

The sample code shows an authorization request to object `o1`. The first command grants a lock for `o1` on behalf of `this.s1`. The second removes from `o1` the ability of `p1` to grant a key. The effect is shown in Figure 4: method `s1`, that houses the request, sends a message to `o1` where it is analyzed by a message filter that makes the appropriate changes to the OACL.

7.3.2 Mandatory Access Control

Mandatory access control (MAC) requires subjects and objects to be tagged with security clearances and classifications defined by a partially ordered set of label pairs: a security level and a category, e.g., `(top secret, crypto)`.

MAC is modeled in MOM by mapping tickets to clearances. A unique ticket exists for each label pair in the partial order, e.g., the `(secret, air force)` clearance might be associated with ticket `secairtkt`.

Figure 5 shows an instance variable and a method (in different classes in the same package) that have been classified as `top secret` and `secret`, respectively. The effect of method `doit()` calling `report(rank)` is to send a method invocation to the read accessor for `rank`. The invocation is denied by the filter controlling access to `rank` because it lacks the appropriate ticket.

7.3.3 Role Based Access Control

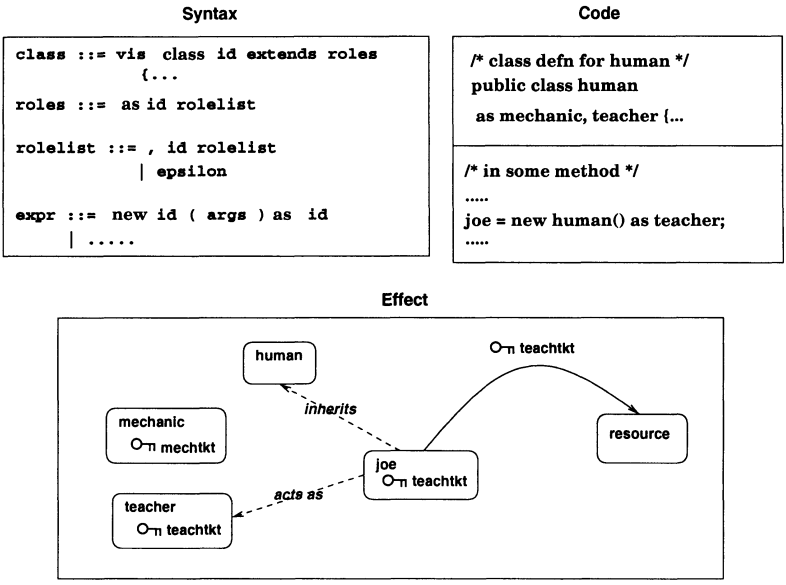

Figure 6. Role based access control.

Role based access control (RBAC) assumes that subjects adopt one or more roles, each defining a set of permissions. MOM implements RBAC by mapping tickets to roles.

Figure 6 shows a class `human` that can assume `teacher` or `mechanic` roles. Templates for the two roles specify the appropriate permissions. When an instance of `human` is created, a role is selected and permissions from the role template are given to the instance. The effect of the code in Figure 6 has instance `joe` taking on a `teacher` role; it is given the appropriate ticket `teachtkt` from the `teacher` role template.

7.3.4 Task Based Access Control

Task based access control (TBAC) apportions trust on a transaction by transaction basis. Implementing TBAC requires functions that operate before and after the transaction to verify preconditions and postconditions. Figure 7 shows a `cashier` charging a `consumer` for a purchase. Note that `consumer` permission is a precondition for the debit procedure. After the debit is completed, `consumer` revokes this permission so that he/she cannot be charged again.

Programming language support for TBAC involves the ability to name tasks and define Boolean functions `before` and `after` that verify preconditions and postconditions, respectively. A task does not commence unless `before` returns `TRUE`. It does not complete and must rollback if `after` returns `FALSE`.

Figure 7 shows how temporary trust is established between `consumer` and `cashier`. First (in the `before` function), `cashier` must get `consumer` approval. Then, `consumer` adds a matching lock and key to `account` and `cashier`, re-

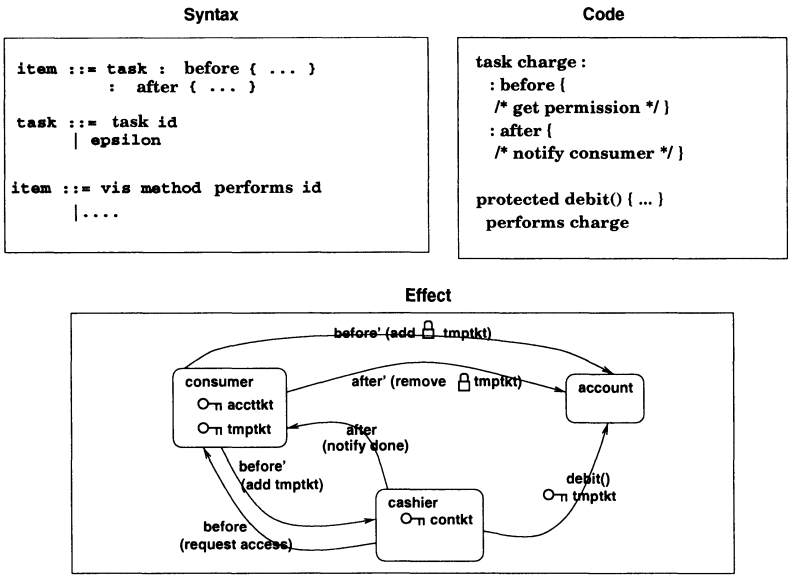

Figure 7. Task based access control.

spectively. After the `before` function terminates and returns `TRUE`, the `debit` method is invoked and a message is sent with the temporary key to debit the `account`. The `after` function then notifies `consumer` that the debit is complete, and `consumer` removes the lock from `account`.

7.4 PROGRAMMING SECURE INTEROPERABILITY

MOM facilitates the secure interoperation of heterogeneous distributed objects at the source and object levels. Source level interoperability is a natural outcome of modeling distributed object systems with MOM as the common substrate. The MOM execution model supports object level interoperability by permitting the encapsulation of native object implementations with MOM wrappings. These wrappings augment native object execution by marshaling communication between objects and engaging MOM's programmable security constructs. The following subsections describe MOM's runtime system and Mumbo, a MOM-based language for orchestrating secure interoperability.

7.4.1 MOM Runtime System

The MOM runtime system is a distributed virtual machine for MOM objects that manifests concurrent message-passing and method invocation. Any object system with a MOM mapping can operate in the MOM runtime system.

Source level interoperability requires mappings between the source languages and MOM. E.g., if C++ and Java are given MOM mappings, any C++ and Java programs could be compiled into MOM and could reside in the same object

space. While this permits interoperation, it does not address secure interoperation. But MOM's programmable security constructs can still be used to create extensions of C++, Java, or new languages that promote secure interoperation.

Source level interoperability is not always practical. Source code may not be available or it may not be efficiently mapped into a MOM system. Native agents in MOM can integrate legacy code seamlessly at runtime using a modified MOM message handler that converts messages into calls to native objects. This message handler generates calls to native functions in dynamic link libraries (DLLs). The approach has two benefits. Wrapping native resources inside MOM objects allows them to inherit the authorization services of the wrapper objects. Classes that form the basis of language extensions and virtual operating systems are readily developed using wrappers. Furthermore, I/O routines are easily implemented as native methods in MOM-based languages.

7.4.2 Mumbo

Mumbo is an object coordination language built from MOM. It permits synchronization of object resources and promotes interoperability using wrapper/translator technologies. Mumbo resembles Java in that instances, classes and interfaces are the main components in program development. However, Mumbo treats classes and interfaces as meta objects for flexibility and fine-grained concurrency. Mumbo also employs MOM's meta object access control scheme to integrate DAC and traditional object-oriented protection.

Mumbo's runtime system is a MOM runtime system. At runtime, a Mumbo domain object is placed in a MOM root domain to encapsulate the compiled system. Mumbo permits the introduction of new systems (users) at runtime. A user can join the runtime environment (when a new Mumbo domain object is added to the existing root domain) or start a new environment (when a distinct root system is spawned for the Mumbo system).

7.4.2.1 Primitive Elements. Mumbo currently has three primitive types: *Names*, *Booleans* and *Lists*. Other primitive types are easily added. Abstract data types can be created by class definitions. A generic *Object* type is introduced to denote a base type for objects.

Names are atomic data elements in Mumbo represented by character strings. They can be compared (by equality) but not modified, e.g., appended or truncated; they are primarily used as list elements.

Booleans are represented by the named constants `TRUE` and `FALSE`. The standard Boolean functions `and`, `or`, `not` and the equality conditional are available. Operators `instanceof`, `implementationof` and `elementof` are Boolean expressions in Mumbo: `instanceof` returns `TRUE` if an object (evaluated from an expression) is an instance of a class object; `implementationof` checks if an object resolved from an object expression is an implementation of an interface; `elementof` tests the membership of an evaluated expression in a list.

Lists in Mumbo are sequences of expressions (including other lists); they evaluate to sequences of primitive data elements. The standard list functions `head`, `tail` and `cons` are available.

7.4.2.2 Mumbo Objects. Classes, interfaces and instances are constructed from MOM objects. Each object expression evaluates to an object reference, a *gid* that uniquely identifies its Mumbo runtime system. Access to a slot or method mandates the use of an object expression to specify a referrent. Any access specified without an object expression is assumed to be local.

Mumbo methods employ the `native` modifier to denote a method interface to be used as a proxy for a native function. The name of the DLL follows the `native` modifier and the method must have the same name as the native function in the DLL. E.g., `native public void mydll.myfunction();` declares a native method implemented by `myfunction()` inside `mydll.dll`.

Since Mumbo classes (and metaclasses) are first class objects, the opportunity exists for dynamically modifying class and metaclass behavior at runtime. Inheritance is modeled by delegating messages to object superclasses.

Interfaces contain method and slot signature information inside the metadata repository. This information is used to determine whether or not an object implements the specified interface. Interfaces can be instantiated to create interface objects that define the roles of class instances. A novel feature of Mumbo methods and slots is their ability to claim responsibility for implementing a piece of an interface. Methods with different names can satisfy a portion of an interface as long as they have matching signatures. This feature facilitates abstraction within the interface/component framework.

7.4.2.3 Discretionary Access Control in Mumbo. The opportunity for interaction between distinct Mumbo units (users) poses hazards to the resource security and integrity. Mumbo employs MOM's ticket-based access control scheme to provide authorization services to Mumbo elements. (Tokens are associated with Mumbo elements to implement DAC.) Mumbo enables developers to use traditional notions of public, protected and private elements to specify initial authorization states for Mumbo programs. Protection can then be fine-tuned using authorization commands issued from within method bodies.

For example, the command `AuthorizationRequest A.mycar {Grant this Lock; Revoke this.private Grant.Lock}`, asks `A.mycar` to perform two services: (i) grant it a `Lock` for the requesting object, and (ii) remove from `A.mycar()` the ability to grant a `Lock` by `this.private`.

Initially, a developer can specify an element as public, protected or private. Public elements are open to everything in the root environment. Private methods are available only to instances of a class. Private slots are only visible inside an object. The designation of an object, method or slot in Mumbo as protected means that it is accessible only within its Mumbo domain. Permissions on all elements can change, e.g., public elements can become private and vice versa. Furthermore, the authorization states of elements at any given time could be

neither public, protected nor private, depending on whether authorization commands are issued. Note that public, protected and private protection modes are intended to permit developers to easily specify the initial authorization states of Mumbo elements.

7.5 RELATED WORK

The foundational object systems, ACTORS (Agha, 1986) and Loops (Stefik and Bobrow, 1985), have motivated this work. Both systems capture object behavior using meta objects. Recent work by Abadi and Cardelli (1996) seeks a common formalism for object behavior. MOM's ability to support a variety of decentralized authorization models for distributed objects also arises from reconciling object behavior and access control in meta objects.

Authorization for distributed objects has been addressed by Nicomette and Deswarte (1997) and van Doorn *et al.* (1996). The former approach is similar in that it relies on collaborative security kernels for decentralized access control. However, it promotes vouchers – indirect access rights transmitted by objects with capabilities. Vouchers are intriguing because they support the principle of least privilege in capability-based systems. Modifying the structure of tickets to include vouchers is a natural extension to MOM. The latter approach extends Modula-3 network objects to secure network objects (SNOs) with security features and promotes subtyping as a means of specifying security properties for objects. SNOs and MOM bind object-oriented programming languages into service for integrating security into objects and methods.

Java addresses program security in open distributed environments with a novel security architecture (Dean *et al.*, 1996). The architecture centers on a security manager that authorizes specific method invocations. Unfortunately, corrupting the security manager effectively circumvents access control. Our decentralized approach integrates security functionality within each object, resulting in more robust security solutions.

Database security research has influenced access control of objects (see, e.g., Dittrich *et al.*, 1989). Several authorization models have been proposed (e.g., Bertino *et al.*, 1994; Rosenthal *et al.*, 1994). Method based access control (MBAC) for object systems was introduced by Gal-Oz *et al.* (1993). Access types are reduced to a single *execute* type by considering methods as the primary basis for access control. MBAC is a natural choice for MOM because all access occurs through method invocation.

ORION/ITASCA adopts DAC for objects (Rabitti *et al.*, 1991). It embraces notions of explicit/implicit, positive/negative and weak/strong authorizations. The authorization model is based on four fundamental access types and incorporates roles. An extension by Bertino *et al.* (1994) supports additional access types, type dependency modeling and distributed authorization control. MOM type definitions can be extended to model positive/negative and weak/strong authorizations by reconfiguring message filters. Semantic-based forms of implicit authorization naturally emerge from object systems built with MOM.

Providing flexible mechanisms and models that support multipolicy access control is becoming increasingly important. In theory, each object in a multipolicy environment could be protected according to a different policy. Bertino *et al.* (1996) employ flexible access control mechanisms and mediators (Wiederhold, 1992) to "tune" access control mechanisms to specific policies. Multipolicy systems seek the highest common ground in access control – as does MOM – to support the interoperation of disparate authorization policies.

The Argos system unifies heterogeneous access control models in an open distributed environment (Jonscher and Dittrich, 1995). It offers a configurability for modeling various identity-based authorization policies. While identity-based authorization models are pervasive, MOM's approach using tickets is more general. A ticket can represent an identity, a role, a transaction, or a clearance level and can therefore be used to implement a variety of authorization policies.

The Distributed Computing Environment (DCE) employs a decentralized authorization service for access control in open distributed environments (Rosenberry *et al.*, 1993). DCE manages authorizations with access control lists (ACLs). Principals (subjects) are registered and assigned group and organization membership. A member's name, group and organization information define its privilege attributes. Member privilege attributes are embedded in a ticket provided by the authentication server at login. An ACL manager resides in each file server to authorize access requests. DCE supports various authorization models by allowing customization of ACL managers. Our approach also provides a common set of mechanisms for interoperability of heterogeneous components, but it also permits the uniform treatment of secure distributed objects in new language-based environments.

7.6 CONCLUSIONS

Online enterprises require high assurance security for mission-critical services. Developers are hampered by the lack of tools and methodologies for constructing verifiably secure distributed object systems. The Meta Object Model (MOM) integrates object functionality and primitive access control mechanisms to facilitate the development of secure distributed object languages and systems. MOM has been used to design Mumbo, a coordination language providing discretionary access control for distributed objects.

Future work will focus on augmenting MOM with authentication and audit mechanisms. Plans also include mapping C++ and Java to MOM, and using Mumbo to pursue secure interoperability of heterogeneous distributed objects.

References

[1] Abadi, M. and Cardelli, L. (1996) A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, **125(2)**, 78–102.

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

[3] 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.

[4] 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.

[5] Dean, D., Felten, E. and Wallach, D. (1996) Java security: From HotJava to Netscape and beyond. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 190–200.

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

[7] Gal-Oz, N., Guddes, 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.

[8] Gilgor, V., Huskamp, J., Welke, S., Linn, C. and Mayfield, W. (1987) Traditional capability-based systems: An analysis of their ability to meet the trusted computer security evaluation criteria, IDA Paper P-1935, Institute for Defense Analyses, Alexandria, Virginia.

[9] Hale, J., Threet, J. and Shenoi, S. (1998) Capability-based primitives for access control in object-oriented systems, in *Database Security, XI: Status and Prospects* (eds. T.Y. Lin and X. Qian), Chapman and Hall, London, 134–150.

[10] 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.

[11] 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.

[12] Nicomette, V. and Deswarte, Y. (1997) An authorization scheme for distributed object systems. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 31–40.

[13] 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.

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

[15] Rosenthal, A., Williams, J., Herndon, W. and Thuraisingham, B. (1994) A fine grained access control model for object-oriented DBMSs, in *Database Security, VIII: Status and Prospects* (eds. J. Biskup *et al.*), Elsevier, Amsterdam, 319–334.

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

[17] Van Doorn, L., Abadi, M., Burrows, M. and Wobber, E. (1996) Secure network objects. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 211–221.

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