

Flexible Enterprise Access Control with Object-oriented View Specification

Mark Evered

School of Mathematics, Statistics and Computer Science
University of New England
Armidale, 2351, NSW, Australia

Email: markev@mcs.une.edu.au

Abstract

The per-method access control lists of standard middleware technologies allow only simple forms of access control to be expressed and enforced. Given the increasing use of web-based applications involving sensitive data, the increased threat and the stringent requirements of privacy laws, a more flexible and secure approach is needed. In this paper we present a three-step approach to access control involving object-oriented encapsulation, middleware based on a new, more secure access control mechanism and the high-level specification of method-oriented views. We demonstrate the use of the approach in a simple web-based E-commerce environment to provide secure electronic cheques.

Keywords: access control, web-based systems, object-oriented systems

1 Introduction

With the development of middleware technology, it has become standard practice to construct software systems as collections of heterogeneous distributed components. Such systems are increasingly used for database integration, decision support systems, electronic commerce and many other applications. In general, the information stored within the components of these systems is sensitive and requires some form of access control. This is particularly important as the internet is increasingly used as the basis for distributed systems and as the threat from hackers and malicious software continues to grow. As well as protecting a system from these external threats, privacy laws require that the access control mechanism also ensures that each principal with access to the system only uses the information exactly as required for their role within an organization. Privacy laws also require that an individual have access to the information stored about him/her by an organization.

Despite the sensitivity of the data and the growing threat, access constraints in middleware technology remain fairly inflexible. OMG's Corba (Blakely, 2000), Microsoft's COM+ (Eddon, 1999) and Sun's EJB (Sun,

1999) all include a form of access control list (ACL) but these are add-on features which support only simple role-based access control. Much attention has been given to encryption techniques but, while encryption is certainly very important, it protects only the communication and authentication in the system. It provides only the *basis* for a secure access control mechanism.

In this paper we describe a three-step approach to providing more secure and more flexible access control. These involve encapsulating the data as persistent objects, using a flexible capability-based approach to restrictions on method invocations and supporting the new concept of method-based view specifications. These view specifications combine the advantages of a strict object-oriented encapsulation with the traditional concept of views from database systems.

We begin in the next section by discussing the advantages of an object-oriented approach to access control and the need for extending the framework of access control criteria to enforce a true 'need-to-know' approach. In the following section we review the concept of bracket capabilities as a more powerful and flexible access control mechanism for distributed systems than is offered by standard middleware. In section 4 we describe the new method for the specification of views to a persistent object and the transformation of these views to a capability implementation as realised in the **Opsis** system. We demonstrate the usefulness of this approach by defining a simple form of secure electronic cheque. Finally, in section 5, we give an overview of and comparison with related research.

2 Semantic access control

2.1 Object-oriented access

The data stored by an organization is typically stored in databases, usually relational databases and more recently also object-oriented databases. In both cases, the data is characterised by entities with attributes. Access to the data can be specified in terms of views, which essentially grant the right to read and/or write certain attributes of the entities. This read/write approach, typical also of traditional file systems, is a fairly low-level and course-grained form of access control.

Alternatively, the components of a distributed system

can be viewed as persistent objects, with each object containing persistent data hidden by encapsulation and accessible via interface methods. A major advantage of this strict form of object-orientation is that the security can be based on the interface methods of an object. This provides a fine-grained *semantic access control* (Evered, 2000) in contrast to the course-grained read/write protection of databases. If the methods are appropriately chosen, the access rights can be based on meaningful, high-level operations associated with the object in the real world. So, for example, we may define a persistent object which implements a set of bank accounts with methods for creating a new account, depositing some amount in one of the accounts, checking the balance in one of the accounts etc. (Fig. 1).

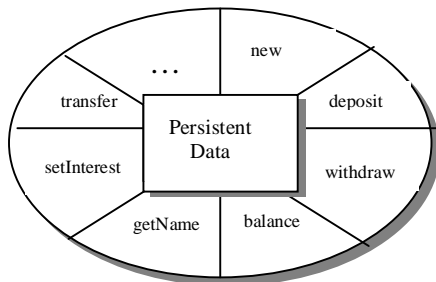


Fig 1: A bank accounts object

This corresponds to the Java interface definition:

```
interface Accounts {
    Key new(String name, String address);
    void deposit(Key key, Currency amount);
    void withdraw(Key key, Currency amount)
        throws insufficientFunds;
    Currency balance(Key key);
    String getName(Key key);
    void setInterest(Percent rate);
    void transfer(Key key, Key toKey,
                  Currency amount)
        throws insufficientFunds;
}
```

The access rights can then be granted on the basis of the roles of the principals within the organization. A bank teller may have access to the deposit and withdraw methods whereas the bank manager may also have access to the methods for adding a new account and setting the interest rate. The first step, then, in providing better access control is to organise the data as persistent objects with application-level interface methods. Of course, this does not mean that the data cannot be stored in traditional databases - the persistent object may in fact simply be implemented as a wrapper object or *façade* around a traditional form of storage.

This idea of method-based restrictions goes back as far as Jones' and Liskov's (1978) suggestion of a static type-based constraint mechanism and has been adopted in some contemporary middleware mechanisms. Sun's EJB can be used in conjunction with a configuration file describing the mapping from users to roles and from roles to methods and Corba (Blakely, 2000) has a similar mechanism. The COM+ security mechanism

supports 'per-method access control lists' which record, for each method, a list of users allowed to invoke that method (Eddon, 1999).

An alternative to a protection mechanism based on access control lists is an *object-based capability* protection mechanism. A capability for an object is simultaneously an identifier for the object and a list of allowed methods. The possession of the capability represents the right to call those methods on that object. This has the added security advantage that the naming of objects is unified with the protection mechanism so that someone who has no access to an object does not even know of the object's existence. Capabilities have therefore long been recognised (Wilkes and Needham, 1979) as providing a greater degree of security in the need-to-know sense than ACLs. Among the disadvantages of capabilities, however, are that they are held by the user and not with the object and so must themselves be protected in some way from forgery and tampering. The Monads system (Rosenberg and Abramson, 1985) supports object-based capabilities for distributed systems but assumes a homogeneous network and a special operating system kernel. The authors have proposed a middleware technology based on a form of sparse capabilities (Anderson et al., 1986).

2.2 Extending access control

The access control mechanisms described in the previous section all limit the access to an object by returning an error message if certain methods are invoked. However, this is not the only kind of access restriction which is possible or useful in controlling access to a persistent object.

In fact, even in terms of per-method access control, the above mechanisms are not ideal. With each of these mechanisms, all the methods of the object are still known to all the principals even if they cannot be called. Ideally, in a need-to-know security environment, someone who is not allowed to invoke a method should not know of the existence of that method, just as someone without a capability does not even know of the existence of the object¹. So, for example, if a bank teller is not allowed access to the **new** and **setInterest** methods of an **Accounts** object, then the software running on behalf of the teller should ideally see the object as if it had the type:

```
interface TellerAccess {
    void deposit(Key key, Currency amount);
    void withdraw(Key key, Currency amount)
        throws insufficientFunds;
    Currency balance(Key key);
    String getName(Key key);
    void transfer(Key key, Key toKey,
                  Currency amount)
        throws insufficientFunds;
}
```

instead of the type **Accounts**. This type effectively

¹ As well as increasing security, it also simplifies the use of the object by the user if only the relevant methods are visible.

defines the *view* that the software has of the underlying object. In contrast to database views, however, this view is defined in terms of methods rather than attributes.

Privacy laws go beyond the access to be granted to members within the organization holding the data. They also demand that the individual whose data is being stored has full access to his/her own data. In our banking example this is particularly relevant. An account owner can be given access to the information not just to check its accuracy but also for the purpose of home banking. So, what view of the **Accounts** object should be given to the owner of an individual account? As well as restricting access to the methods **balance**, **getName** and **transfer**, we must also ensure that only the right account is being accessed. This means that the **Key** parameter of **balance** and **getName** and the first **Key** parameter of **transfer** must be restricted to a particular value.

This requires a mechanism which grants access rights to the user in such a way that an error is returned if the wrong account number is specified in the call. Or, better still, in terms of views, we would like the account owner to view the object as if it had the type:

```
interface Account {
    Currency balance();
    String getName();
    void transfer(Key toKey,
                 Currency amount)
        throws insufficientFunds;
}
```

where it is implicit that the appropriate account is to be accessed.

A number of further kinds of access control are also useful for flexibly specifying the security constraints associated with different roles in a system. In (Evered, 2001), the author has described a formalism in which five basic type operators are used in combination to specify security constraints in terms of a *view type*. The operators modify the methods, parameters and semantics of the type through which a user accesses the underlying persistent object. The operators provide for:

- specifying that some methods should return an access violation error
- specifying that an access violation error should be returned for parameter values other than a specified value
- restricting the view type to exactly the allowed methods and parameters
- enhancing the semantics of the object type with logging of accesses and access attempts
- specifying a state-dependent rule such as ‘access only at specified times’ or ‘access allowed only once’

The view type effectively gives the user the illusion of accessing an object of that type when in fact it is just a

representation of a restricted access to an object of the original type. So, for example, the type **Account** would appear to be the type of an individual bank account object but a call to this object would actually be a call to an **Accounts** object with the account number automatically inserted. The **Account** object can be seen as a *virtual object*².

This extended concept of security constraints requires a more flexible access control mechanism to be incorporated into the middleware of distributed applications.

3 A Flexible Access Control Mechanism

The second step in our approach is to enforce a strict access control mechanism for checking invocations of the methods of a persistent object. As our underlying access control mechanism we use *bracket capabilities*. This mechanism has been presented more fully elsewhere (Evered, 2002a, Evered, 2002b). The mechanism is based on object capabilities rather than ACLs because capabilities enhance and simplify security by unifying object naming with the protection mechanism (Wilkes and Needham, 1979). A number of possible alternatives have been suggested for implementing capabilities. These include special architectures (Rosenberg and Abramson, 1985), encryption (Mullender and Tanenbaum, 1986) and sparse (or password) capabilities (Anderson et al., 1986). We base our mechanism on sparse capabilities since these require no special architecture or costly encryption algorithms and also because they alleviate the revocation problem. A sparse capability generally consists of an object identifier (for locating the object) together with a large (un-guessable) random number (the password). The access rights associated with the capability (that is, with that particular random number) are not stored in the capability itself but with the object being accessed, so can easily be modified or revoked without access to the capability.

Our capabilities differ from traditional sparse capabilities in that they contain not an object identifier, but an identifier for a (capability) server that knows the location of the object. This indirection allows for object migration as well as flexibility in the communication mechanism. These are both particularly important for mobile applications. When a persistent object is created, a capability for the object is created and registered with a capability server.

The main distinguishing characteristic of bracket capabilities is seen in the process of *refinement*, that is, when the possessor of a capability wishes to grant a more restricted view of the object to other principals in the system. This is done by a call to the **refine** method. Each persistent object, as well as implementing an interface such as **Accounts** also implements a number of administrative methods such as

² Such an object may actually exist within the Accounts object or it may not exist if, for example, the Accounts object is just an object-oriented façade around a relational database.

`deleteObject`, `deleteCapability` and `refine`. The `refine` method in a bracket capability system has the form:

```
refCap = x.refine(interfaceName,
                 parameter,
                 comment);
```

where `interfaceName` denotes the type with which the persistent object 'x' is to be viewed when used via the capability `refCap` and `interfaceName+ "Bracket"` denotes the class of an object through which calls to the persistent object will pass when invoked via `refCap`. The result of the `refine` call is depicted in Fig. 2.

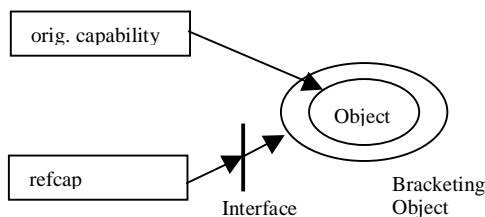


Fig 2: The result of the 'refine' operation

It can be seen that calls using the capability `refCap` are directed through a kind of proxy or *bracketing* object. This bracketing object is stored together with the persistent object in the same way that access rights are stored with the object for traditional sparse capabilities. The `parameter` parameter in the `refine` call is passed to the bracketing object to initialise it and the `comment` parameter is simply a text describing the purpose of this new restricted view.

A copy of `refCap` can be given to the principals who are to have this kind of access. Traditional object-based capabilities or ACLs in which access is restricted to particular methods can easily be simulated with this mechanism while other bracket classes can be used to implement the full range of security constraints described in section 2.2, including restrictions on parameters, logging of method calls and constraints based on time, number of accesses etc.

So, for example, given a capability `c1` to an `Accounts` object, we can generate a new capability `c2` to give to the owner of account 12345 as follows:

```
accs = c1.open();
Capability c2 = accs.refine("Account",
                          "12345", "Access to account 12345");
```

where the interface `Account` is defined as in section 2.2 and a class `AccountBracket` has been defined to be used for a bracketing object. The `AccountBracket` object will implement the `Account` interface and will pass calls on to the original `Accounts` object after adding the account number parameter 12345.

A bracketing class may have more methods than those available in the view given by the interface type. These extra methods can be used by the creator of the capability for monitoring or altering the bracketing such as to inspect logging information or to revoke or alter access constraints. The bracket capability mechanism allows arbitrary bracketing classes but, in a particular security environment, we may want to limit the set of classes that can be used as brackets. While still allowing users to create more restricted views, we may want to specify what kind of restriction they can impose. This is possible since the bracketing is part of the security mechanism. In fact, since the `refine` call is itself just a method call to the persistent object, we can use the mechanism itself to specify that, for some user, it can only be invoked with certain values for the `interfaceName` parameter.

4 Specification of access rules

4.1 View definitions

The bracket capability mechanism described in the previous section is a very flexible and powerful access control mechanism but it is not convenient to use at the application level. The programming of bracketing classes can involve much repetition, be error-prone and, in some cases, be quite complex. The specification of access control restrictions should not need to involve low-level programming. For this reason, the third step in our approach is to allow access control to be specified via high-level method-oriented view descriptions and then automatically generate the bracket classes used for the creation of bracket capabilities.

In the `Opsis` system developed by the authors, the middleware for distributed Java applications is based on bracket capabilities and tools exist for the generation and distribution of capabilities based on *view specification files*. A view specification file generally consists of a set of `interface`, `define` and `grant` constructs. An `interface` construct is an extension of the Java interface type definition and is used to specify a restricted view type to an underlying object type or to a previously defined view type. A `define` construct is used to generate a new capability to a persistent object based on an interface specification. A `grant` construct makes such a capability available to the software running on behalf of some principal in the system.

So, for example, given an `Accounts` object and an initial capability `accountsInfo` for that object, we can grant the use of that capability (and therefore full access to the object) to some principal who may be the bank manager:

```
grant accountsInfo to tom.pipersen;
```

For the tellers of the bank, we can define a view which does not include the `new` and `setInterest` methods:

```

interface Teller to Accounts {
  //! Accounts access for tellers
  void deposit(Key key, Currency amount);
  void withdraw(Key key, Currency amount)
    throws insufficientFunds;
  Currency balance(Key key);
  String getName(Key key);
  void transfer(Key key, Key toKey,
               Currency amount)
    throws insufficientFunds;
}

```

and use this to provide access as:

```

define tellerAccess as
  Teller for accountsInfo;

grant tellerAccess to jack.b.neembol;
grant tellerAccess to george.e.pawji;

```

The syntax '*//!*' in the interface definition indicates a comment which will be used as the **comment** parameter in the **refine** operation to indicate the purpose of the restricted view. A further construct, **revoke**, can be used at a later time to revoke access rights. So, for example:

```
revoke tellerAccess;
```

invalidates the capability **tellerAccess** and all other capabilities which may have been derived from it. This is done by a call to the **deleteCapability** method.

In general, an interface specification may also include a list of *preconditions*. In this case the access is only allowed if all of the preconditions are fulfilled. So, for example, we might want to restrict the amounts of money a teller can transfer without approval by the manager. This can be expressed in a modified **Teller** specification:

```

interface Teller to Accounts {
  //! Accounts access for tellers
  void deposit(Key key, Currency amount);
  void withdraw(Key key, Currency amount)
    throws insufficientFunds;
  Currency balance(Key key);
  String getName(Key key);
  void transfer(Key key, Key toKey,
               Currency amount)
    throws insufficientFunds;
where
  amount < 10000;
  balance(key) < 100000;
}

```

This specifies that a method invocation involving an **amount** parameter will only be allowed to proceed if the **amount** parameter is less than 10000 and that a method invocation involving a **key** parameter will only be allowed to proceed if the **balance** method returns a value less than 100000 for that **key** parameter. This kind of restriction can be used to specify parameterised

role-based access control conditions (Covington et al., 2000).

As well as calls to the methods of the underlying object or view, the conditions can contain calls to a system object giving the time, date, etc. A special built-in condition **onceOnly** specifies that a capability based on this view is to be automatically invalidated after the first method call for which it is used.

Finally, view specifications can be parameterised. This corresponds to the **parameter** parameter which is passed to the **refine** operation. A parameterised view specification for account owners can be expressed as:

```

interface Account[key] to Accounts {
  //! Access to account #key
  Currency balance();
  String getName();
  void transfer(Key toKey,
               Currency amount)
    throws insufficientFunds;
}

```

and instantiated and distributed to the account owner as:

```

define account12345 as
  Account[12345] for accountsInfo;
grant account12345 to jack.njihl;

```

The above **define** specification is implemented as:

```

Capability c1 =
  Capability.load("accountsInfo");
accs = c1.open();
Capability c2 = accs.refine("Account",
  "12345", "Access to account 12345");
c2.save("account12345");

```

where a class **AccountBracket** has been generated from the **Account** view as:

```

class AccountBracket implements Account {
  private Accounts underlying;
  private Key key;

  public AccountBracket(Accounts acc,
                       String param) {
    underlying=acc;
    key=Key.parseKey(param);
  }

  public Currency balance() {
    return underlying.balance(key);
  }

  public String getName() {
    return underlying.getName(key);
  }

  public void transfer(Key toKey,
                      Currency amount)
    throws insufficientFunds {
    underlying.transfer(key, toKey,
                      amount);
  }
}

```

4.2 Example: Secure Electronic Cheques for E-Commerce

We now extend the examples of the previous section to provide an example of a simple system using **Opsis** view specifications for access control in electronic funds management. At the centre of the system is an object of the type **Accounts** as described above. After creating this object, we have a capability **accountsInfo** for unrestricted access.

Next we can create a capability for an individual bank account holder. For account number 12345, this can be achieved as described in the previous section with:

```
define account12345 as
  Account[12345] for accountsInfo;
grant account12345 to jack.njihl;
```

The account owner may then wish to provide a restricted access to his account so that another account owner can transfer a certain amount, say \$20, out of the account as a payment. This will then be a restricted view of the account owner's restricted view. The capability for such an access is in fact a *secure electronic cheque* (see Fig. 3).

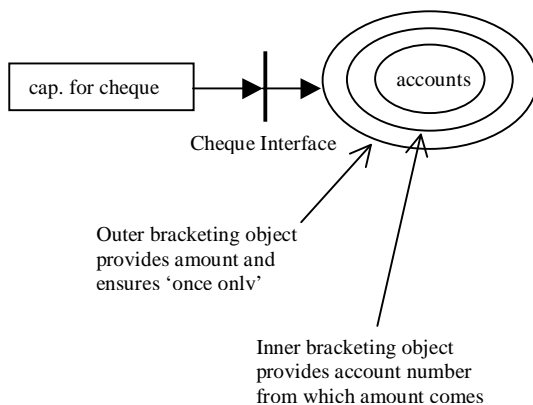


Fig 3: An electronic cheque represented by a view of a view

As well as fixing the amount, we must ensure that this capability can only be used once. We can achieve this by the specifications:

```
interface Cheque[amount, purpose] to
  Account {
  //! Payment of $$amount for #purpose
  void transfer(Key toKey)
  throws insufficientFunds;
where
  onceOnly;
}

define cheque1234 as
  Cheque[20, "one woollen beanie"]
  for account12345;
grant cheque1234 to mary.haddalam;
```

The above **define** construct is implemented as:

```
Capability c1 =
  Capability.load("account12345");
acc = c1.open();
Capability c2 = acc.refine("Cheque", "20",
  "Payment of $20 for one woollen beanie");
c2.save("cheque1234");
```

where the class **ChequeBracket** has been generated from the **Cheque** view as:

```
class ChequeBracket implements Cheque {
  private Account underlying;
  private Currency amount;

  public ChequeBracket(Account acc,
    String param) {
    underlying=acc;
    amount=Currency.parseCurrency(param);
  }

  public void transfer(Key toKey)
    throws insufficientFunds {
    underlying.transfer(toKey, amount);
    underlying.deleteCapability();
  }
}
```

The electronic cheque can be deposited in an account, say account 23456, by the code:

```
Capability c1 =
  Capability.load("cheque1234");
Cheque c = (Cheque) c1.open();
c.transfer(23456);
```

Clearly, the destination account could also be fixed if desired, or, with appropriate methods and conditions, an access specification could be formulated for regular transfers rather than a once-off payment.

5 Related Work

Formal approaches to security specification have generally concentrated either on lattice-based information flow (Bryce, 1997) or on cryptographic protocols (Roscoe, 1995). The idea of defining access via method-oriented view types and operators on those types appears to be new. Sandhu's Typed Access Matrix model (1992), an extension of the HRU access matrix model (Harrison et al., 1976) in which both subjects and objects are strongly typed, aims at using static types for the resolution of decidability issues rather than using types for defining role-based views of an object.

As mentioned above, Corba, EJB and COM+ all include the possibility of a per-method, role-based access control list for limiting the access of users to objects. In some cases, fixed forms of rule-based

access, such as access at certain times of day, are supported. These correspond only to simple, special cases of access control. No direct equivalent of the complex restrictions as required for the above E-commerce example are supported. No direct equivalent of a restricted view type is supported for hiding the existence of unallowed methods and parameters from the users. In both of these middleware technologies, the use of ACLs instead of capabilities makes the security mechanism an add-on feature rather than fundamental and detracts from the security.

Object capabilities have been used in a number of research systems, most notably the Monads system (Rosenberg and Abramson, 1985) but these capabilities require architectural support (or at least a special operating system kernel) and so are not appropriate for heterogeneous networks. In a previous project, the author has developed a capability-based mechanism for heterogeneous distributed applications (Evered, 2000). Like the Monads system and the ACL approaches of Corba, EJB and COM+, however, this supported only simple per-method access control.

The concept of 'bracketing' for applying access constraints has been suggested both as a programming language construct (Keedy et al., 2000) and as a form of 'design pattern' (Gamma, 1995). The suggested programming language approach is interesting in supporting the *reuse* of the bracketing code but it does not allow modification of the interface to the underlying object and, being integrated into the type system of the language, it is a static mechanism.

One use of the *proxy* design pattern is as a protection (or access) proxy. In this case, the interface is identical to the underlying object. The proxy decides whether the access can proceed and returns an error if it should not. Simple per-method access control can be realised by this kind of protection proxy. Bracketing objects which modify the interface offered to a client cannot be seen as strict proxies. They can be seen as special cases of the adapter pattern but whereas an adapter is usually used to provide the view the client would *like* to have of the underlying object, in these cases the adapter is providing the view the client is *allowed* to have.

The concept of providing a user with a restricted view of persistent data is reminiscent of database systems. Traditional database views are attribute-oriented and not method-oriented, however, and therefore cannot support the flexible kinds of access control demonstrated in our example. This attribute-orientation is true even for most object-oriented databases (Mishra and Eich, 1994). Notable exceptions are the method-based model of Fernandez, Larrondo-Petrie and Gudes (1993) and the CACL system of Richardson, Schwarz and Cabrera (1992). The former provides an 'Execute' access right for invoking a method of a persistent object. This is similar to the per-method access control of contemporary middleware systems. The latter supports the concept of an 'authorization type' as a restricted view of an object but does not allow parameter constraints or state-dependent constraints to be specified as part of the view.

Brose (1999) describes a 'view-based' mechanism for Corba but this is again simply a form of language-based per-method access control.

6 Conclusion

The mechanisms for object-oriented access control in contemporary middleware are inadequate in view of the sensitivity of data stored on the internet and the growing threat from hackers and malicious software. They are also inadequate in terms of the complex requirements of privacy laws. These security mechanisms are not integrated at a fundamental level and are capable of enforcing only simple kinds of access control. The access control in a distributed object system should ideally enforce a strict need-to-know view of the data and should support more complex forms of security restriction including restrictions on parameter values and restrictions involving the time of access or the number of accesses.

In this paper we have described a three-step approach to improving access control in distributed applications. The first step is not new. It is to organise the data as a set of persistent objects in which the data is encapsulated by enterprise-level access methods. This allows the formulation of finer-grained role-based access constraints than are possible with the attribute-oriented read/write access of database systems.

The second step is to use middleware which enforces a strict need-to-know view of the persistent objects and which is flexible enough to allow a wide range of access restrictions. In contrast to the access control mechanisms of standard middleware, we have demonstrated that the concept of bracket capabilities can be used for this purpose.

The third step in the approach is to provide an environment in which high-level access specifications can be formulated in terms of method-based views of the persistent objects. These are then translated automatically into the appropriate interfaces and bracketing classes to use for the bracket capabilities. Rights are defined in terms of interfaces which may be parameterised and may specify preconditions. The concepts have been implemented in the **Opsis** system which supports view specification files for describing the access rights of principals for distributed Java applications.

Finally, we have demonstrated the power and simplicity of the approach in controlling the access within an example E-commerce application including the definition of a simple form of secure electronic cheque. Work is currently underway to design and implement a realistic application for data management in an aged-care facility. Object-oriented views will be used for controlling access to personal, health-care and detailed medical data of the residents, for managing electronic payments, for ensuring compliance with privacy laws and for the representation of electronic signatures.

References

- ANDERSON, M., POSE, R.D., WALLACE, C.S. A Password-Capability System, *The Computer Journal*, 29,1, pp.1-8, 1986.
- BLAKLEY, B., BLAKLEY, R., SOLEY, R.M., *CORBA Security: An Introduction to Safe Computing with Objects*, Addison-Wesley, 2000.
- BROSE, G., A View-Based Access Control Model for CORBA, in: Jan Vitek, Christian Jensen (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, Springer 1999.
- BRYCE, C., Security Engineering of Lattice-Based Policies, *Proc. 10th IEEE Computer Security Foundations Workshop*, 1997.
- COVINGTON, M.J., MOYER, M.J., AHAMAD, M., Generalized Role-based Access Control for Securing Future Applications, *Proc. 23rd National Information Systems Security Conference*, Baltimore, 2000.
- EDDON, G., The COM+ Security Model Gets You Out of the Security Programming Business, *Microsoft Systems Journal*, Nov. 1999.
- EVERED, M., A Two-Level Architecture for Semantic Protection of Persistent Distributed Objects, *Proc. Intl. Conf. on Software Methods and Tools*, Wollongong, 2000.
- EVERED, M., Type Operators for Role-based Object Security, 3rd IFIP/ACM Intl. Conf. on Distributed Systems Platforms - Middleware, Heidelberg, 2001.
- EVERED, M., Bracket Capabilities for Distributed Systems Security, *Proc. 25th Australasian Computer Science Conference*, Melbourne, 2002.
- EVERED, M., Ophis: A Distributed Object Architecture Based on Bracket Capabilities, *Proc. Conference on Technology of Object-Oriented Languages and Systems*, Sydney, 2002.
- FERNANDEZ, E.B., LARRONDO-PETRIE, M.M., GUDES, E., A Model of Methods Access Authorization in Object-oriented Databases, *Proc. of the 19th VLDB Conference*, Dublin, 1993.
- GAMMA, E. et al., *Design Patterns*, Addison-Wesley, 1995.
- HARRISON, M.A., RUZZO, W.L., ULLMAN, J.D., Protection in Operating Systems, *Communications of the ACM*, 19, 8, 1976.
- JONES, A., LISKOV, B. A language extension for expressing constraints on data access. *Communications of the ACM*, 21(5):358-367, May 1978.
- KEEDY, J.L., et al., "Software Reuse in an Object Oriented Framework: Distinguishing Types from Implementations and Objects from Attributes", *Proc. Sixth International Conference on Software Reuse*, Vienna, 2000.
- MISHRA, P., EICH, M.H., Taxonomy of views in OODBs, *Proc. ACM Computer Science Conference*, 1994.
- MULLENDER, S.J., TANENBAUM, A.S. The Design of a Capability-Based Distributed Operating System, *Computer Journal*, 29,4, pp.289-299, 1986.
- RICHARDSON, J., SCHWARZ, P., CABRERA, L., CACL: Efficient Fine-Grained Protection for Objects, *Proc. OOPSLA Conference*, 1992.
- ROSCOE, A.W., Modelling and Verifying Key Exchange Protocols using CSP and FDR, *Proc. 8th IEEE Computer Security Foundations Symposium*, 1995.
- ROSENBERG, J., ABRAMSON, D. A. The MONADS Architecture: Motivation and Implementation, *Proc. First Pan Pacific Computer Conference*, p. 4/10-4/23, 1985.
- SANDHU, R.S., The Typed Access Matrix Model, *Proc. IEEE Symposium on Security and Privacy*, 1992.
- SUN Microsystems Inc., Enterprise JavaBeans Technology, <http://java.sun.com/products/ejb/>, 1999.
- WILKES, M.V., NEEDHAM, R.M. *The Cambridge CAP Computer and its Operating System*, North Holland, 1979.