JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 4, May-June 2005

Object-oriented Access Control in Jarrah

Mark Evered, University of New England, Australia

Abstract

Given the sensitivity of the data stored in many information systems and the use of networks to support distributed applications, it is increasingly important to enable precise control of who can access the data in what way. Standard per-method access control lists are not sufficient to capture the complexity of the access constraints which arise if the concept of minimal access is taken seriously. In this paper we describe and justify the design of the access control aspect of the persistent object-oriented language Jarrah, a Java extension for programming secure distributed applications.

1 INTRODUCTION

In general, the information stored within the components of an information system will require some form of access control. This is particularly important when networks are used to implement distributed information systems and as the threat from hackers and malicious software continues to grow. As well as protecting a system from external threats, the access control must also ensure compliance with privacy laws and ideally, should guarantee that each user with access to the system has the minimal access required for their role within the organisation. Although distributed information systems are increasingly used for sensitive applications such as E-commerce and Health Information Systems, surprisingly little attention has been paid to handling the complexities of real-world access constraints. Much attention has been given to the security of encryption techniques but, while encryption is certainly important, it protects only the communication and authentication in the system. It provides only the basis for a secure access control mechanism.

Traditional file access control is based on a simple read/write privilege for defined groups of users. Access control in database systems is similarly based on read/write access to the fields of the stored records. In an object-oriented system, the components of an information system can be implemented as objects, with each object containing data hidden by encapsulation and accessible only via interface methods. These interface methods can be meaningful, high-level operations associated with the object in the real world rather than low-level data access operations. One advantage of an object-oriented approach is that the access control can be based on these high-level interface methods and the access rights can be formulated in terms of who can invoke which of the high-level operations on the data encapsulated within the object. The following interface, for example, defines an object for storing bank account information.

```
interface Accounts {
  void newAccount(int accountNumber, String name);
  String getName(int accountNumber);
  void setInterestRate(float rate);
  void deposit(int accountNumber, int amount);
  void withdraw(int accountNumber, int amount)
    throws InsufficientFunds;
  int balance(int accountNumber);
  void transfer(int accountNumber,
                int toAccountNumber,
                int amount)
    throws InsufficientFunds;
}
```

As can be seen, the interface contains methods for creating a new account, depositing some amount in one of the accounts, checking the balance in one of the accounts etc. The access control associated with this object may allow software executing on behalf of a bank teller to invoke the deposit and withdraw methods whereas the bank manager may also have access to the method for setting the interest rate. This idea goes back as far as Jones' and Liskov's suggestion of a static type-based constraint mechanism [13] and the 'protected subsystems' of Multics [18] and it has been adopted in most contemporary component frameworks for heterogeneous distributed systems.

OMG's Corba [2], Microsoft's COM+ [4] and Sun's Enterprise Java Beans [12] all support a form of per-method access control list (ACL). These kinds of mechanism limit the access to an object by returning an error message if certain methods are invoked by certain principals. However, as has been demonstrated in a number of recent case studies [6, 7], this is not the only kind of access restriction which is possible or useful in controlling access to an object in a real information system. In fact, even in terms of per-method access control, the standard mechanisms are not ideal since all the methods of the object are still known to all the users even if they cannot be called. Ideally, in a need-to-know security environment, a principal who is not allowed to invoke an object should not know of the existence of that object and a principal who is not allowed to invoke a particular method should not know of the existence of that method.

In this paper we outline the development of an access control construct for the persistent object-oriented programming language Jarrah. The following section describes the management of persistent objects in Jarrah using object capabilities. Section 3 investigates requirements for access control both in terms of expressiveness and in terms of generic criteria such as clarity and efficiency. In section 4 we use these criteria to develop an access control construct for use in Jarrah while section 5 describes the implementation of the persistence and access control in Jarrah using the Opsis component framework. Finally, in section 6, we relate this approach to other work on persistent objects and object-oriented access control.

2 PERSISTENT OBJECTS IN JARRAH

The programming language Jarrah is an extension of Java which supports secure persistent objects and, through the work presented in this paper, supports fine-grained access control to these objects. Unlike the light-weight persistence of Java, Jarrah supports persistent objects which can be concurrently shared by multiple processes and which need not be explicitly stored and loaded. These aims are similar to those of PJama [1] and database-based approaches such as JDBC [11]. Jarrah differs in that it focuses on security, allows multiple realisations of persistence within a single application and supports distributed object systems. The persistence interface of Jarrah is very simple. It consists of the predefined class Capability and the predefined interface type Persistent.

Given an expression x which returns an object reference, the call:

```
Capability c = Capability.create(x);
```

creates a new persistent object which is a copy of the object to which x refers, where ‘copy’ is taken in the sense of Java’s serialization and de-serialization. The call returns a Capability object which is later used for gaining access to the persistent object. The capability is essentially a 128-bit password which can be passed to users who are to have access to the object.

An example of the most common form for the call is:

```
Capability c = Capability.create(new AccountsImpl());
```

to create a new object as a persistent object, in this case an Accounts object implemented by the class AccountsImpl. The persistent object is always accessed via its interface definition, in this case Accounts. The create method can also be invoked with further parameters specifying the form of persistence to be used, the location at which the object is to reside and the protocol for remote invocation. Defaults are used if any of these is not given.

In order to use the object, a call to the **open** method of the capability is made. Without the capability there is no way to gain access to the persistent object. The **open** call returns a reference which can be used as if referring directly to the object. (In fact this occurs via a stub object and the underlying Opsis system which takes care of activation, remote access and sharing). For example:

```
Accounts a = (Accounts) c.open();
a.transfer(12345, 23456, 100);
```

As well as implementing its defining interface, a persistent object also automatically implements the standard interface Persistent which provides a number of generic methods for persistent objects:

```
interface Persistent {
  void delete();
  void lock();
  void unlock();
  ...
}
```

So, for example, the object created above can be deleted as follows:

```
Persistent p = (Persistent) c.open();
p.delete();
```

Parameters to methods of persistent objects are passed with copy semantics. For primitive types this is as expected. For transient objects, a copy of the object is made via serialization. Persistent objects can be passed as references by passing a capability object.

It is in this context we now wish to define a programming language construct for defining restricted access to persistent objects. We begin by examining the requirements for such a construct.

3 ACCESS CONTROL REQUIREMENTS

3.1 General criteria

Before we examine access control requirements from the point of view of expressiveness, we first list a number of criteria which we claim should be fulfilled by any mechanism concerned with security.

Concise

Access control is no use if it is not correct and if it is possible to express complex constraints only in an awkward or long-winded way, then errors are likely to be made. Our first criterion is therefore that the mechanism should allow the constraints to be expressed in an easy, concise way. This is especially important for the most common kinds of constraints.

Clear

Closely related to concise expression is the clarity of the mechanism. Access constraints must be not only expressed but also checked. Ideally, at least for the most common cases, it should be apparent at a glance what the rule is saying and therefore whether it is correct in terms of the desired policy.

Non-repetitious

Mechanical repetition, extraneous effort or the use of ‘copy and paste’ in the expression of the rules are indications that something is amiss. A single concept should be expressed in a single place to avoid errors of omission.

Incremental

Persistent objects in real applications can have very many methods on their interfaces. A good mechanism will allow the access rules for these to be grouped into manageable subsets of related methods and then combined.

Aspect-oriented

Access control information should be separated from the functionality of a method, not embedded into or mixed up with it. This corresponds to the idea of aspect-oriented programming [14] where separate aspects of a program such as security and synchronisation are formulated separately and then combined automatically by an ‘aspect weaver’. This separation makes both the application code and the access control easier to understand and also allows the same application object to be used in different security contexts with different access rules.

Fundamental

Ideally, an access control mechanism is integrated at a fundamental level within a system. rather than being an optional add-on. One advantage of this is that developers are forced to address access control questions from the very start. Another advantage is that it is then more difficult for hackers to ‘get around’ the mechanism.

Parameterisable

Often the access constraints for a number of principals are similar and can be expressed by a single parameterised access rule. If this is not possible then it leads to repetition and copying which are susceptible to errors.

Positive

The access rights of a principal should be expressed in terms of what that principal is allowed to do rather than what he/she is not allowed to do. This ensures that the default is that no access at all is allowed and that each permission must be explicitly listed. A strict need-to-know approach to access control is not only desirable in military environments. Financial transactions and the manipulation of sensitive personal data are increasingly being performed electronically via distributed systems and people have a right to expect that no more of their data is being revealed than is absolutely necessary for a particular service.

Dynamic

It is often not sufficient to define all aspects of the access constraints statically. It may be necessary to add a new kind of access or revoke access dynamically. It may also be desirable to modify an existing kind of access.

Composable

It may also be desirable to base a restricted form of access on a previously defined restricted access, ie. to further restrict a restricted access. Amongst other possibilities, this is useful in supporting the concept of hierarchical roles within an organisation.

Efficient

Clearly, a certain overhead will always be involved in performing access control checks but this overhead should be kept at a reasonable level. The temptation will otherwise be there to compromise on the degree of security to improve system performance.

3.2 Expressiveness

The per-method access control lists of standard object-based access control offer a significantly finer control than simple read/write mechanisms but we question whether they are sufficient to express the complexities of real information systems while fulfilling the additional criteria of the previous section. A number of case studies by the authors have shown that even relatively simple information systems can require very complex access rules if the need-to-know principle is taken seriously.

In this section we look at the kinds of access constraints which an access control construct needs to be able to express. We will restrict our discussion to the bank accounts example from above but for a detailed discussion of these kinds of access constraints in a Health Information System the reader is referred to [7].

Subset of methods

This is the fundamental kind of access restriction in an object-based approach to access control. So, for example, nobody but the bank manager may be allowed to invoke the setInterestRate method of the Accounts object. The need-to-know principle implies that methods which are not allowed to be invoked should not even be visible to a principal.

A particularly interesting use of this kind of restriction is in realising signatures. If the access control mechanism can guarantee that only a single person can invoke a method, then the fact that it has been invoked can be utilised as a signature.

Access dependent on data values

Sometimes the question of whether an access should be allowed to proceed depends on the information stored in the object. So, for example, a trainee teller may only be allowed to access accounts with a balance less than a certain limit.

Access based on call history

Like the constraints in the previous category, this kind of constraint depends on the state of the object but here the relevant factor is not the attributes of the information being stored but the allowed sequence of method invocations. A special case of access based on call history is the permission to invoke a method only once. So, for example, we may give someone the right to transfer a certain amount of money from an account to their own account as a payment. This right should expire after it has been used.

Access with restricted parameter values

Sometimes the question of whether an access should be allowed to proceed depends not on the information stored in the object but on the value passed as a parameter. So, for example, a trainee teller may only be allowed to perform a transfer between accounts for an amount less than a certain limit.

Access with missing parameter

An account owner can be allowed to invoke any of the methods which return information stored about him/her but, of course, should have no access to information stored about other account owners. This implies that an account owner only be allowed to invoke methods with a particular value of the accountNumber parameter. Since only a single value for the parameter is allowed, we can, in fact, hide the parameter entirely from the principal and have the appropriate value supplied automatically by the access control mechanism for that principal. This is again in correspondence with the need-to-know principle.

Access based on external information

For added security, we may want to restrict accesses to a certain time of day. The access check will then need to refer to some other object in the system to obtain the current time when a method is invoked. In general, information from other objects in the system may be relevant in deciding whether an access is to be allowed.

Access with external side-effect

This is clearly not desirable in the general case but is necessary for the purpose of logging accesses. Sometimes it is desirable to keep a record of who has done what and when with a persistent object. In a high-security context it may even be necessary to record every invocation of an object but in most information systems this would result in a flood of data which would tend to hide rather than expose the accesses of interest.

Logging may be required both for successful and for unsuccessful method invocations. For example, the manager may need to be informed if a bank employee repeatedly attempts to perform some operation on an object for which he/she has no permission.

4 A LANGUAGE CONSTRUCT FOR ACCESS CONSTRAINTS

4.1 The Jarrah 'access' construct

Access to persistent objects in Jarrah can be controlled via the access construct. We introduce this construct progressively by using the examples discussed in the previous section. The first example involves allowing only a subset of the interface methods. We use a normal Java interface type to describe the interface as it is to be viewed by a certain principal or for a certain role. For example:

```
interface TellerView {
  String getName(int accountNumber);
  void deposit(int accountNumber, int amount);
  void withdraw(int accountNumber, int amount)
    throws InsufficientFunds;
  int balance(int accountNumber);
  void transfer(int accountNumber,
                int toAccountNumber,
                int amount)
    throws InsufficientFunds;
}
```

The access construct can then be used in its minimal form to define the access to the underlying object using this view:

```
access TellerAccess to Accounts
   provides TellerView {}
```

As well as omitting methods from a certain view, we can also omit method parameters. For example, for an account owner we can define the view as:

```
interface Account {
  String getName();
  int balance();
  void transfer(int toAccountNumber,
                int amount)
    throws InsufficientFunds;
}
```

Here the accountNumber parameter is omitted since only a single value is allowed for each account owner. The view then appears to be giving access to a single bank account. In this case the access construct must supply the value to be used for the missing parameter. One way to do this is to declare a constant in the access construct with the same name as the missing parameter. The access provided for the owner of account number 12345 can then be defined as:

```
access AccountOwner12345Access to Accounts
   provides Account {
  static final int accountNumber=12345;
}
```

Of course, as mentioned in section 3.1, it would be tiresome and error-prone to write a new access definition for every account owner. We therefore require a parameterisable access definition. This can be achieved by allowing variable declarations in the access construct and a constructor method for their initialisation. Just as with constants, such a variable is substituted for a missing parameter of the same name. The general account owner access can then be defined as:

```
access AccountOwnerAccess to Accounts
   provides Account {
  private int accountNumber;

  public AccountOwnerAccess(int accountNumber) {
    this.accountNumber=accountNumber;
  }
}
```

and can be instantiated as required.

Many forms of access control require some checks or actions to be performed before and/or after a method invocation. These include access dependent on data values, call history or external information, access dependent on parameter values and access with logging. For these cases, the access construct can include a pre and/or a post section containing code which is executed before/after the call to the underlying persistent object. If the code in the pre or post section uses a parameter name then the code is only executed for methods having a parameter with that name.

For example, for the case of the trainee teller we can define the access as:

```
access TraineeTellerAccess to Accounts
   provides TellerView {
  pre { assert(amount<10000); }
}
```

to ensure that this teller performs only transactions with an amount less than $10000. Similarly, we can define an access with logging by passing to the constructor of the access definition a reference to an object which records information about the method invocations. The pre and/or post sections can then contain calls to this object.

Finally, as mentioned in section 3.1, the access control may need to be dynamic. For example, we may want the transaction limit for a trainee teller to be changeable rather than fixed at $10000. This requires an administrative method for setting the limit value. Therefore, in addition to constants, variables and a constructor, the access construct must also allow method definitions. For clarity, these can be declared in an interface type such as:

```
interface TraineeTellerAdmin {
  void setLimit(int limit);
}
```

with the access now defined as:

```
access TraineeTellerAccess to Accounts
   provides TellerView
     implements TraineeTellerAdmin {
  private int limit=10000;
  pre { assert(amount<limit); }

  public void setLimit(int limit) {
    this.limit=limit;
  }
}
```

It can be seen that, in order to provide the flexibility to handle complex access constraints, the access construct must be able to contain constants, variables, methods and a constructor. In fact, we define it to have the full functionality of a Java class with the addition of the to, provides, pre and post clauses. This includes the possibility of using inheritance via the extends clause to provide an access definition as an extension of a previously defined access definition. This fulfils the criterion of incremental definition from section 3.1 which is important for expressing clear access control to objects with many interface methods.

An access definition in Jarrah can be used just as any class in Java. Objects can be created and manipulated as ordinary objects but can additionally be used to create a restricted form of access to a persistent object. The next section described how this is done.

4.2 The ‘restrictBy’ and ‘admin’ methods

We extend the definition of Persistent to include a number of further methods:

```
interface Persistent {
  void delete();
  void lock();
  void unlock();
  Capability restrictBy(Access a);
  Object admin(Capability c);
  void deleteCapability();
}
```

The restrictBy method is passed an object created from an access definition and returns a new capability object which allows access to the underlying persistent object with the restrictions defined for that definition. For example:

```
Persistent p = (Persistent) c.open();
Capability c2 = p.restrictBy(new AccountOwnerAccess(12345));
```

The capability can then be given to principals who are to have this kind of access, in this case the owner of account 12345, and can be used to gain access to the object via the appropriate view:

```
Account a = (Account) c2.open();
int i = a.balance();
```

In fact, the exact definition of the restrictBy method is that a *copy* of the access object given as a parameter is used to control the access. This is because the access object is held together with the persistent object and this may reside on a network node other than the one where the restrictBy call is made.

The only other requirement is that it is somehow possible for the principal who creates a new capability to gain access to the administrative methods associated with that new form of access. This is accomplished via the admin method. This is called with the new capability as a parameter and returns a reference which enables the administrative interface to be called. For example:

```
Persistent p = (Persistent) c.open();
Capability c3 = p.restrictBy(new TraineeTellerAccess());
```

and then at some later time:

```
TraineeTellerAdmin t = (TraineeTellerAdmin) p.admin(c3);
t.setLimit(20000);
```

The possessor of a capability which provides a restricted access to a persistent object can create further capabilities with a more restricted form of access. This is in accordance with the criterion of composability. So, for example, an account owner can create a capability which allows a certain amount to be transferred from his/her account to some other account. This capability is in effect an electronic cheque. It can be achieved as follows:

```
interface Cheque {
  void transfer(int toAccountNumber)
    throws InsufficientFunds;
}

access ChequeAccess to Account
```

```
     provides Cheque {
    private int amount;

    post { deleteCapability(); }

    public ChequeAccess(int amount) {
      this.amount=amount;
    }
  }
```

The post clause invalidates the capability which was used to gain access and so ensures that the cheque can only be used once. A cheque for $100 can, for example, be created by the code:

```
Persistent p = (Persistent) c2.open();
Capability c4 = p.restrictBy(new ChequeAccess(100));
```

and can then be passed to the recipient and used by code such as:

```
Cheque x = (Cheque) c4.open();
x.transfer(23456);
```

5 IMPLEMENTATION

5.1 The Opsis system

The Jarrah compiler translates the access construct and the calls to Persistent and Capability methods into standard Java with calls to the Opsis component framework system. Opsis is a Java-based system for constructing distributed applications based on the mechanism of bracket capabilities [6]. The two main features of the system are:

- fine-grained access control through the use of bracket capabilities as object identifiers
- flexibility and transparency in the mechanisms used for both remote invocation and persistence

A bracket capability is a 256-bit sparse capability which consists of a 128-bit capability-server identifier (CSID) and a 128-bit password. The CSID identifies a server which knows the location of the persistent object while the 128-bit password is a random (unguessable) value which acts as the permission to access a certain persistent object in a certain way (ie. with a certain view and a certain access (or bracketing) object).

The information stored on a capability server includes the interface type (the view) with which an object is to be accessed when opened with a certain capability, the location of the object, its name and the mechanism to be used in remote invocations of the object's methods. When an object is created, it is registered with a server and the location of that server is stored in the new capability. Capability servers are given only the lowest 64-bits of the capability's password as an index rather than the whole 128-bits. This is because otherwise, the server would essentially own a copy of the capability and therefore have all the associated access rights.

As mentioned above, the Opsis system supports multiple mechanisms for remote invocation and (orthogonally) multiple mechanisms for persistence. In all cases, the mechanisms used for the invocation and the persistence remain completely transparent to the client object. The path of a method call from a client to a server object can be visualised as in Fig 1.

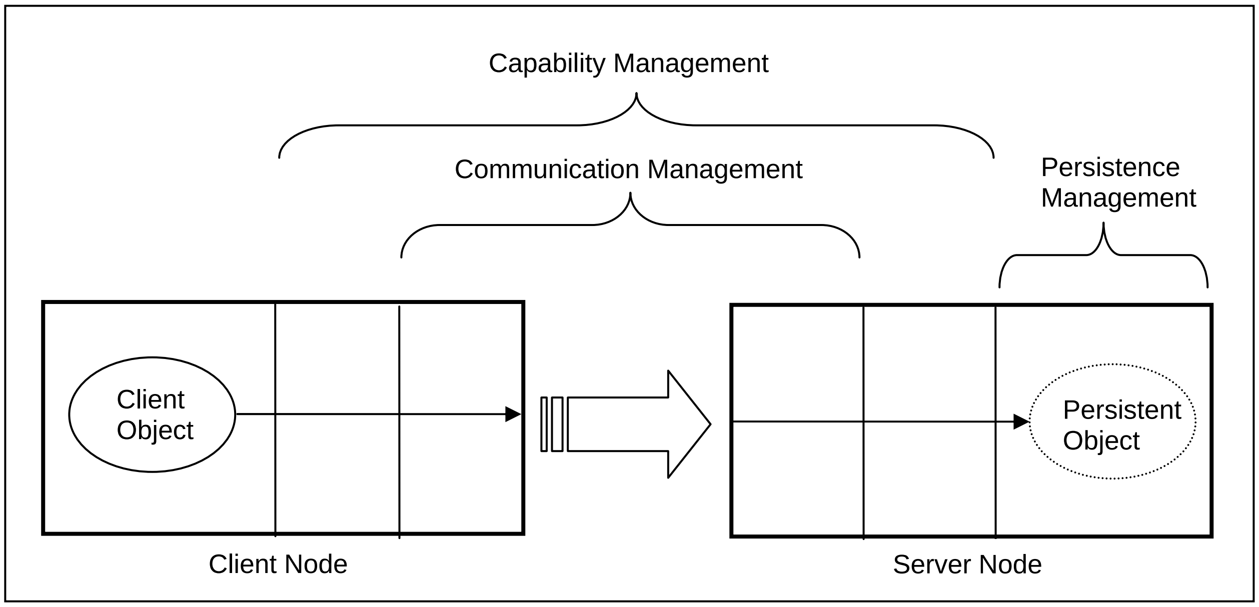


Fig. 1: A method call in Opsis

The communication management uses the appropriate mechanism for securely passing the method call and parameters to the remote system and returning a result. It supports multiple concurrent invocations.

The capability management on the client side is responsible for providing the right view of the persistent object to the client and for appending the capability used to open the object to each method invocation as an extra parameter. The capability management on the server side is responsible for checking that the access proceeds only as allowed for that capability. Nothing that can be done on the client system can increase the access rights to the remote object. The persistence management is responsible for making the persistent data available and sharable, and presenting it in the form of an object of the appropriate type.

Possible mechanisms for remote invocation are:

- Java's RMI
- a web-based CGI mechanism
- a mechanism using 'ssh', the secure shell protocol

Possible mechanisms for persistence are:

- Java's built-in light-weight persistence (using inter-process communication to achieve sharing)
- wrapper objects around a ‘Postgres’ database

5.2 Compilation

The access construct is translated by the Jarrah compiler to three Java classes. One is a stub class for use in the capability management on the client side. Its interface is the interface type declared in the provides clause. An object of this class is created when a persistent object is opened using the open method of a capability.

The second generated class is for a ‘bracketing object’ through which calls will pass on the server side before reaching the underlying object. The bracketing object is created and initialised when the capability is created and it is made persistent together with the underlying object. The capability management on the server side maps the capability password to this object when a method invocation takes place. It is this class which contains the code of the pre and post clauses for each method and which substitutes variable or constant values for parameters when passing on calls.

The third generated class is again a for stub object but this time for the client who has created a capability and now wants to gain access to the administrative methods associated with that capability. The interface of the class is the interface type declared in the implements clause of the access construct. The object is created when the admin method of a persistent object is called. Invocations of the methods on this interface are transmitted across to the capability management on the server side and are then directed to the appropriate bracketing object.

The create method of the class Capability is translated to a method invocation on a special persistent object residing at the location where the new persistent object is to be created. This ‘creator’ object does all the work of setting up the new object with a particular implementation, a particular form of persistence and a particular remote procedure protocol. In order to create a new persistent object at a certain location, a principal requires a capability for the ‘creator’ object at that location. The run-time system automatically looks for this capability in a defined place in the environment of the principal invoking the create method.

5.3 Optimisation

The parameter to the create method of Capability and the parameter to the restrictBy method of a persistent object are both expressions which return a reference to an object. In both cases the definition for the general case is that a copy of the parameter object is made. This definition is necessary because the node where the parameter will be used may be different from the node on which the call is being made.

The most usual case for both calls, however, is that a newly created object is passed as the parameter. Either:

```
Capability.create(new SomeClass())
```

to create a new persistent object or

```
p.restrictBy(new SomeAccessDefn())
```

to create a new form of access to an existing persistent object. In this case, it is very inefficient to create a new object on one node, serialize it, send it to another node, de-serialize it and then discard the original object (since there is no remaining reference to it). The compiler optimises this case by translating to a different call to the Opsis system which directly creates the object on the target node.

6 RELATED WORK

Much work on persistent programming languages has followed the principle of persistence by reachability whereby an object is persistent if and only if it is reachable via one or more ‘persistent roots’. Examples are systems such as PJama [1] and PerDiS [9] which provide a special persistent object store to manage persistence transparently according to this principle. This approach has a number of problems in terms of information security and access control since a piece of data cannot be considered to be ‘held inside’ a persistent container object as is the case with Jarrah. Many references to a piece of data can be passed around and stored in the system. One problem with this is that the data cannot be explicitly deleted. As long as one reference via a persistent root still exists, the data will continue to exist. The problem in terms of access control is that it cannot be guaranteed that an access to the data will occur via a well-defined high-level interface method and so access constraints cannot be formulated or enforced in terms of these methods. In addition, these systems cannot support different realisations of persistence for different components of an application.

Another approach in persistent programming is to couple an existing language to a database system as with Java’s JDBC [11]. In this case the access constraints could be programmed into a wrapper object as a kind of ACL. Assuming access to the data is only possible through this object and that principals can be identified, the expressiveness criteria can be met but the generic criteria of section 3.1 would not be fulfilled. In particular, the aims of clarity and conciseness would not be met.

As mentioned above, standard component frameworks such as Corba, COM+ and EJB include the possibility of a per-method, role-based access control list for limiting the access of principals to interfaces or 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 required for the case studies are supported. No direct equivalent of a restricted view of the object is supported for hiding the existence of unallowed methods and parameters from the principals. In all of these component 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 in Multics' *protected subsystems* [18] and in the Monads system [17] 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, one of the authors has developed a capability-based mechanism for heterogeneous distributed applications [5]. Brose [3] has proposed a language-based extension to the Corba security model in which the allowed 'views' for each user are defined in terms of the methods of an object type. Like the Monads system and the ACL approaches of Corba and EJB, however, these support only simple per-method access control. In all cases, all of the methods are visible to all principals even if they may not be invoked.

The concept of 'bracketing' for applying access constraints has been proposed as a form of 'design pattern' [10] with one use of the *proxy* design pattern being a protection (or access) proxy. With a proxy, however, the interface is always identical to the underlying object. Bracketing objects which modify the interface offered to a client cannot be seen as strict proxies. They could 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. Again, however, the use of general program code cannot provide the clarity and conciseness we require.

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 required for minimal access. This attribute-orientation is true even for most object-oriented databases [15]. Notable exceptions are the method-based model of Fernandez, Larrondo-Petrie and Gudes [8] and the CACL system of Richardson, Schwarz and Cabrera [16]. 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, state-dependent constraints etc. to be specified as part of the view.

7 CONCLUSION

Given the sensitivity of the data stored in information systems and the use of open networks to support distributed systems, it is increasingly important to enable precise control of access. In contrast to a coarse-grained read/write approach, object-based access control can allow the formulation of access rights in terms of the permission to invoke application-level operations. This is still not sufficient, however, to capture the complexity of the access constraints which arise if the concept of minimal access for each principal is taken seriously.

In this paper we have described and justified the design of the access control aspect of the persistent programming language Jarrah, a Java extension for supporting secure heterogeneous distributed object systems.

We have formulated a number of criteria for fine-grained access control mechanism. These include expressiveness criteria, explained here in terms of a simple E-commerce system for accounts, and general criteria such as clarity, parameterisation and composability.

Based on these criteria, we have shown that the access control construct for an object-oriented language requires at least the functionality of the class construct. The ‘access’ construct of Jarrah additionally includes a ‘to’ clause to define the underlying interface, and a ‘provides’ clause to define the view offered to a principal. This view is defined as a Java interface and it can omit methods and method parameters relative to the underlying interface. The ‘access’ construct also allows ‘pre’ and ‘post’ clauses which provide statements for bracketing the call to the underlying object in a concise, non-repetitive way.

All persistent Jarrah objects automatically implement the *Persistent* interface. This offers a method for creating a new capability which provides a restricted form of access to an object. It is passed the appropriate access object as a parameter. The *Persistent* interface also offers a method for gaining access to the administrative methods of an access object.

The language Jarrah is implemented by translation to Java with calls to the Opsis system which uses a form of sparse capability and provides transparent distribution and persistence. A number of optimisations are carried out for the most common cases.

REFERENCES

[1] Atkinson, M.P., Daynes, L., Jordan, M.J., Printezis, T., Spence, S. (1996): An orthogonally persistent Java, *ACM SIGMOD Record 25, 4.*

[2] Blakley, B., Blakley, R., Soley, R.M. (2000): *CORBA Security: An Introduction to Safe Computing with Objects*, Addison-Wesley.

[3] Brose, G. (1999): 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.

[4] Eddon, G. (1999): The COM+ Security Model Gets You Out of the Security Programming Business, *Microsoft Systems Journal*, November.

[5] Evered, M. (2000): A Two-level Architecture for Semantic Protection of Persistent Distributed Objects, *Proc. Intl. Conf on Software Methods and Tools*, Wollongong.

[6] Evered, M. (2002): Opsis: A Distributed Object Architecture Based on Bracket Capabilities, *Proc. Conference on Technology of Object-Oriented*

[7] Evered, M., Bögeholz, S. (2004): A Case Study in Access Control Requirements for a Health Information System, *Australasian Information Security Workshop*, Dunedin.

[8] Fernandez, E.B., Larrondo-Petrie, M.M., Gudes, E., A. (1993): Model of Methods Access Authorization in Object-oriented Databases, Proc. of the 19th VLDB Conference , Dublin.

[9] Ferreira, P., Shapiro, M., Blondel, X., Fambon, O., Garcia, J., Kloosterman, S., Richer, N., Roberts, M., Sandakly, F., Coulouris, G., Dollimore, J., Guedes, P., Hagimont, D., Krakowiak, S. (1998): PerDiS: design, implementation, and use of a PERsistent Distributed Store, Technical report, QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98.

[10] Gamma, E. et al. (1995): *Design Patterns*, Addison-Wesley.

[11] Hamilton, G., Catell, R.G. (1996): JDBC: A Java SQL API., Technical report, Sun Microsystems.

[12] Hartman, B., Flinn, D.J., Benznosov, K. (2001): *Enterprise Security with EJB and CORBA*, Wiley.

[13] Jones, A., Liskov, B. (1978): A language extension for expressing constraints on data access. *Communications of the ACM*, 21(5):358-367, May.

[14] Kiczales, G. et al. (1997): Aspect-oriented programming, *Proc. European Conference for Object-Oriented Programming*, Finland (Lecture Notes in Computer Science, vol. 1241). Springer.

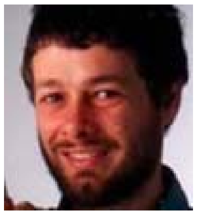
[15] Mishra, P., Eich, M.H. (1994): Taxonomy of views in OODBs, *Proc. ACM Computer Science Conference*.

[16] Richardson, J., Schwarz, P., Cabrera, L. (1992): CACL: Efficient Fine-Grained Protection for Objects, *Proc. OOPSLA Conference*.

[17] Rosenberg, J., Abramson, D. A. (1985): The MONADS Architecture: Motivation and Implementation, *Proc. First Pan Pacific Computer Conference*, p. 4/10-4/23.

[18] Saltzer, J.H. (1973): Protection and the Control of Information Sharing in Multics, *Symposium on Operating System Principles*, Yorktown Heights, NY.

About the author



Mark Evered is a Senior Lecturer in the School of Mathematics, Statistics and Computer Science at the University of New England in Armidale, Australia. He completed his PhD at the Technical University of Darmstadt in Germany. His research interests include Object-based Systems, Security, Persistence and Programming Language Design and Implementation. E-Mail: markev@mcs.une.edu.au.