Reflective Authorization Systems: Possibilities, Benefits, and Drawbacks

Massimo Ancona[1], Walter Cazzola[2], and Eduardo B. Fernandez[3]

[1] DISI, Department of Computer Science,
University of Genova, Genova, Italy
ancona@disi.unige.it
[2] Department of Informatic, Systems and Communication,
2nd University of Milano, Milano, Italy
cazzola@dsi.unimi.it
[3] Department of Computer Science and Engineering,
Florida Atlantic University, Boca Raton, FL, USA
ed@cse.fau.edu

Abstract. We analyze how to use the reflective approach to integrate an authorization system into a distributed object-oriented framework. The expected benefits from the reflective approach are: more stability of the security layer (i.e., with a more limited number of hidden bugs), better software and development modularity, more reusability, and the possibility to adapt the security module with at most a few changes to other applications. Our analysis is supported by simple and illustrative examples written in Java.
Keywords: Authorization, Distributed Objects, Object Orientation, Reflection, Security.

1 Introduction

Security implies not only protection from external intrusions but also controlling the actions of internally-executing entities and the operations of the whole software system. In this case, the interleaving between operations and data protection may become very complicated and often intractable. For this reason, security must be specified and designed in a system from its early design steps [11]. From another point of view

- it is very important that the security mechanisms of the application be correct and stable;
- the security code should not be mixed with the application code, otherwise it should be very hard to reuse well-proven implementations of the security model.

If this is not done, when a new secure application is developed the designer|implementer wastes time to re-implement and to test the security modules of the application. Moreover, security is related to: "who is allowed to do what, where and when"; so security is not functionally part of the solution of the application

problem, but an added feature defining constraints on object interactions. From this last remark we can think of security as a feature operating at a different computational level and we can separate its implementation from the application implementation.
In our opinion it is possible to exploit some typical reflection features, like *separation of concerns* and *transparency*, to split a secure system into two levels: at the first level there are (distributed) objects cooperating to solve the system application; at the second one, rights and authorizations for such entities are identified, specified and mapped onto reflective entities which transparently monitor the objects of the first level and authorize the allowed access to other objects, services, or information.
Working in this way it is possible to develop stable and reliable entities for handling security. It is also possible to reuse them during system development, thus reducing development time and costs, and increasing application level assurance.
In most systems, authorization is defined with respect to persistent data and enforced by the DBMS and|or operating system. Object-oriented systems define everything as an object, some persistent some temporary, where this separation is not visible at the application level. In these systems authorization must be defined at the application level to take advantage of the semantic restrictions of the information [10]. An early system (not object-oriented) (see [11], page 195), attempted this kind of control by defining programs that had predefined and preauthorized accesses. Reflection appears as a good possibility for this type of control because it does not separate persistent from temporary entities. The Birlix operating system [18] used reflection to adapt its nonfunctional properties (including security) to different execution and application environments (for more details on how satisfy nonfunctional requirements using reflection, see [21]).
In this paper we examine how to use a reflective architecture, such as those described above, to manage the authorization aspects of an application and the advantages and drawbacks of using such an approach.

2 Background on Reflection and Security

2.1 Reflection

Computational reflection or just reflection is defined as the activity performed by an agent when doing computations about itself [15]. Behavioral and structural reflection are special cases which involve, respectively, agent computation and structure (for more details see [8]).
A reflective system is logically structured in two or more levels, constituting a *reflective tower*. Entities working in the base level, called base-entities or reflective entities, define the system basic behavior. Entities in the other levels (meta-levels), called meta-entities, perform the reflective actions and define further characteristics beyond the application dependent system behavior.
Each level is causally connected to adjacent levels, i.e., entities belonging to a level maintain data structures representing (or, in reflection parlance, *reifying*) the states and the structures of the entities in the level below. Any change in

the state or structure of an entity is reflected in the data structures reifying it, and any modification to such data structures affects the entity's state, structure and behavior.
Computational reflection allows properties and functionalities to be added to the application system in a manner that is transparent to the system itself (separation of concerns) [20]. To this respect, it is useful to consider also *reflection granularity* [6], that is, the minimal entity in a software system for which a reflective model defines a different meta-behavior. A finer granularity allows more flexibility and modularity in the software system at the cost of meta-entity proliferation. The reflective models considered here are: *meta-object*, and *channel reification*.

Meta-Object Model. In the meta-object model, meta-entities (called *meta-objects*) are objects, instances of a proper class. Each base-entity, called also *referent*, can be bound to a meta-object. Such a meta-object supervises the work of the linked referent. The meta-object model is well known in literature, more on it can be found in [15].

Channel Reification Model. In the channel reification model [1, 2], one or more objects, called *channels* are established between two interacting objects. Each channel is characterized by a triple composed by the objects it connects and by the *kind* of the meta-computation it performs.

$$\textsf{channel} \equiv \textsf{(client, server, channel_kind)}$$

A *channel kind* identifies the meta-behavior provided by the channel. The kind is used to distinguish the reflective activity to be performed: several channels (distinguishable by the kind) can be established between the same pair of objects at the same moment.
Each channel persists after each meta-computation, and is reused when a communication characterized by the same triple is generated. The features of the model are: method-level granularity, information continuity, channel lazy creation, and global view[1]. Each service request of a specific kind is trapped (shift-up action) by the channel of the corresponding kind connecting client and server objects, if it exists. Otherwise, such a channel is created; in either case, it then performs its meta-computation and transmits the service request to the supplier. The server's answer is collected and returned to the requesting object (shift-down action).

2.2 Security: Authorization Systems

An authorization system plays a monitoring role, judging if the requests sent by an object to another object are permissible requests. The judgment is based on

[1] The global view concept is defined in [6] as the situation in which the meta-computation can involve all base-entities and aspects involved in the computation on which it reifies.

security information related to *objects* and *subjects*; where a subject represents an entity performing or requesting an activity (i.e., an active object playing the client role), while an object is a passive entity supplying a service (i.e., a passive object or an active object playing the role of server). One way to model such authorizations is the access matrix model (see [14]). In this model the authorization rules are described by a bidimensional matrix indexed on subject and objects and access to an object o is allowed to a subject s when the item ⟨s,o⟩ of the matrix contains the permitted access type. Such a model can be realized by *capability lists*, *access control lists*, or combinations of these. Formally, an authorization right R for a subject s to access an object o, through message (method) m can be written R(s)=(m,o). Here m usually represents a high-level access type, e.g., hire an employee.
A Role-Based Access Control (RBAC) model is particularly suitable for object-oriented systems [17], because accesses can be defined as operations defined in the objects according to the need-to-know of the roles [9]. Because reflection supports a fine granularity of access, its combination with RBAC can be quite effective.

3 How We Model Security With Reflection

A computational system answers questions and supports actions in some applicative *domain*. Following P. Maes [15] we can say that:

> "A system is *causally connected* with its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other."

For example a *control software* is causally connected with the controlled process if changes of the state of the physical process are reflected by changes of the control software state and vice versa. For this purpose a control software system incorporates structures representing the state of the controlled process. If we consider authorization rights added to a critical control system we note an *indirect* causal connection of authorization with the application domain. Authorization (and more generally security) defines capabilities related to human and/or mechanical agents concerned with the responsibility of performing tasks that are considered critical for the controlled system behavior. Thus, an authorization validation mechanism is a *nonfunctional* feature causally connected with software agents and objects performing control tasks. This is our motivation for adopting a reflective architecture for implementing authorization schemes. In other words, security specifications correspond to nonfunctional specifications [9] and should be implemented into a meta-level.
Using computational reflection we can separate the authorization control mechanism from the application. Moreover, the meta-level should be protected from malicious intrusions and attacks from the base-level or from other processes. A solution could consist in executing the meta-level in a different address space

as an independent process, communicating with the base-level via a mechanism similar to the *local procedure calls* (LPC) of Windows NT [7]. LPC is a locally optimized form of the well known mechanism of *remote procedure call* (RPC) of Unix and other systems: LPC is a message-passing mechanism through which clients make requests to servers and used for the server's reply. The main necessary feature of the LPC mechanism is the protection domain installed around an LPC call.

3.1 Illustrative Scenario

We illustrate our ideas by using the following scenario: the system is composed of several objects interacting in a client-server manner. For security reasons the services supplied by a server and its data are protected and prohibited to some subjects. To support our presentation we use some stubs of Java code. The code is tailored to the case of three clients and one server supplying two methods, one reading the server state and the other modifying it. Each client has different rights on the two supplied services. Java [3] is not a reflective language, for our purposes we realize the reflection in a naive way by emulating it with inheritance. Of course in this way transparency and efficiency are compromised. However, we chose this approach for simplicity and because it permits to point out the advantages and the weak points of the reflective approach. In particular, it shows the necessity to implement the context switch in a secure way. The complete code can be downloaded from `http://www.disi.unige.it/person/CazzolaW/OORSecurity.html`.

In the next section, in order to avoid confusion with objects in object-oriented systems, we use the term *base-objects* to refer either to subjects and objects of the authorization terminology. Moreover, we call *service request* (or method call) every access performed by a subject on an object. There are several reflective architectures suitable for applying authorization rules, we consider only two of them: the *meta-object model* and the *channel reification model*. Each of them provides features suitable to handle different security aspects.

3.2 Meta-Object Approach

In the meta-object model a meta-object is associated with each object (as shown in Fig. 1); this meta-object encapsulates the related access control list of the referred object. Each request performed by a subject s to an object o is trapped by the related meta-object mo, which evaluates the authorization request. Then, it either rejects the request by rising an exception, or it delivers the request to the original destination s.
The Java architecture in shown in Fig. 2. In this case the reflective approach is handled by the server part of the application. The client is directed to talk only to the server. To make this approach to work, we have wrapped the server code (the kernel of Fig. 2) into a shell which receives all client requests and performs the shift-up action by forwarding each request to its own meta-object. Thus, a

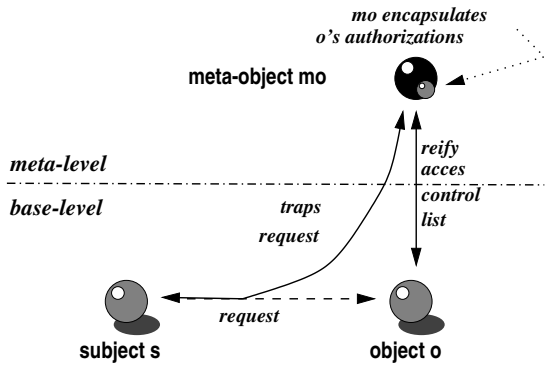

Fig. 1. Meta-object model for security

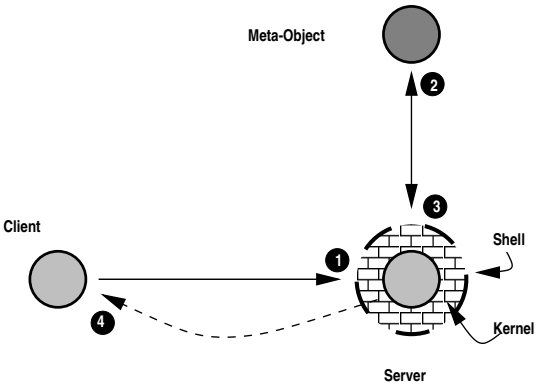

Fig. 2. Meta-object approach: the Java structure

server is composed of two parts: a kernel and a shell, related by the inheritance link. As it can be deducted by inspecting the code (Listing 1), the shell waits for a client request and forwards it to the meta-object by calling the meta-object metaBehavior method (rows ⑤ and ⑩) to check if the request is authorized or should be rejected. Then, if the request is authorized by the meta-object, the shell proceeds to the execution by calling the corresponding method of the kernel (rows ⑥ and ❶). The meta-object implements the authorization policy. All the work is performed by the metaBehavior method. Method metaBehavior, when activated for the first time, reads the referent access rights from a file and encapsulates the referent capability list. In all other activations, method metaBehavior uses the capability list encapsulated by the meta-object to check if the request has to be authorized or rejected.
When a client issues the following call:

$$\text{permission} = \text{server.methodRead}();$$

Listing 1 Java code for the server shell

```
① public class TheServer extends RServer {
②  String REF2;
③  InterSMO rem_obj2;   // declaration of an SMO remote object

④  public boolean methodRead(int IDClient) {
⑤    boolean access = rem_obj2.metaBehavior(METHOD_READ, IDClient);
⑥    if (access) return super.methodRead(1);
⑦    else return false;
⑧  }

⑨  public boolean methodWrite(int IDClient) {
⑩    boolean access = rem_obj2.metaBehavior(METHOD_WRITE, IDClient);
❶    if (access) return super.methodWrite(1);
❷    else return false;
❸  }

❹  public static void main(String args[]) {
❺    System.setSecurityManager(new RMISecurityManager());  // RMI Registration

      // Client declaration of the TheServer+SMO client-serving, in which we look
      // up and fetch the remote object SMO.

❻    rem_obj2 = (InterSMO) Naming.lookup("//SMO"+IDServer);
❼    REF = "//CS";

      // Creation of one instance of the remote object TheServer

❽    TheServer rem_obj  = new TheServer();
❾    Naming.rebind(REF, rem_obj);
❿    System.out.println("Waiting for some client request...");
①  }

②  }
```

a request for the execution of methodRead is trapped by the shell (step ❶ in Fig. 2) which calls the meta-object for obtaining the authorization (step ❷ in Fig. 2). The meta-object replies to the shell (step ❸ in Fig. 2) and if authorized, allows the kernel to return the reply to the request. Otherwise, the shell notifies the client that a violation occurred by sending it an error message (step ❹ in Fig. 2).
This model is suitable for implementing highly specialized role rights (specialized per object, per subject or per service request). Moreover, each meta-object may hold the methods necessary to modify authorization rules of the base-objects [12], which can be called for modifying active authorizations. In this way, the developer of base-objects does not need to worry about validity of authorization updates, which are encapsulated into the meta-object behavior. Obviously our example suffers of efficiency penalties due to the reflective implementation in Java.

3.3 Channel Reification Approach

As we can see by inspecting the meta-object code of the previous section, the meta-object reifies many pieces of information, and when the authorization

Listing 2 Java code for the server kernel

```
public class RServer {

 public boolean methodRead(int i) {
   System.out.println("sending methodRead");
   return true;
 }

 public boolean methodWrite(int i) {
   System.out.println("sending methodWrite");
   return true;
 }

}
```

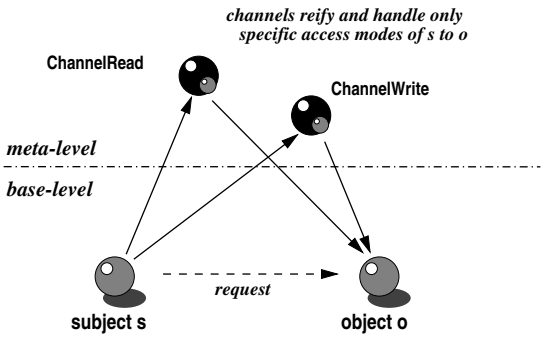

Fig. 3. Channel to model access mode authorization

mechanisms become more complex, the metaBehavior code grows in complexity and size, thus making the code error prone. In our example relative to the meta-object, we use two simple access modes: *allowed to read,* and *allowed to write,* both managed by a single meta-object. The example is quite simple but sufficiently complex to show the increase in complexity of the code necessary to handle all cases.

Another approach consists of using a reflective model with a finer granularity than that of the meta-object model, by entrusting each access mode into a different reflective entity. By using the channel reification model [2] we can define a channel kind for each existing access mode. Each element of the access matrix is encapsulated by one or more channels (one for each value assigned to the matrix item). For example, when a request is sent by s to o, with access mode write, the request is trapped by the channel of kind *write* (ChannelWrite), established between s and o, which validates it (see Fig. 3). In this way the authorization validation process is simplified: each channel checks few authorizations and there exists reflective entities only for base-entities requiring an authorization validation protocol. Obviously, the main drawback of this approach with respect to the meta-object approach is a higher number of reflective entities created, however it is possible to limit this number by merging together meta-entities showing a

Listing 3 Java code for the security meta-object

```
public class SMO {
  static String REF2;

  int idserver; // referents id

  boolean[] caplist_read = null;
  boolean[] caplist_write = null;

  public boolean metaBehavior(char MethodType, int ClientName) {
    if (caplist_read == null) {
      FileInputStream rights = new FileInputStream("rights");
      int clients, servers;
      servers = rights.read();
      caplist_read = new boolean [clients = rights.read()];
      caplist_write = new boolean [clients];
      rights.skip(idserver*(2*clients+1));
      for (int i=0; i<clients;i++) {
        caplist_read[i] = ((rights.read() == "R")?true:false);
        caplist_write[i] = ((rights.read() == "W")?true:false);
      }
    }
    return ((method_type == "R")?caplist_read[ClientName]:caplist_write[ClientName]);
  }

  public static void main(String args[]) {
    idserver = args[0];
    System.setSecurityManager(new RMISecurityManager());
    REF2 = "//SMO";
    SMO rem_obj2 = new SMO();
    Naming.rebind(REF2, rem_obj2);
    System.out.println("Security meta-object waits for some request...");
  }
}
```

similar behavior.

Due to the *global view* property (see [6]) of the channel reification model, each reflective action starts from the client and involves the client, the server, and a channel in its execution. The reflective behavior is achieved by overriding the normal behavior of the binding|look-up between the client and the server, and the remote method invocation in order to divert the request from the server to the channel. The client is composed of three layers (see Fig. 4). The inner layer supplies the operations necessary to communicate with the server. The central layer encapsulates all operations of the inner layer and uses them to communicate with the right channel. The outer layer defines the behavior of the client.

The server code corresponds exactly to the kernel code of the previous approach. There is no need to wrap it because the server receives requests that have been previously filtered by the channels, it cannot be accessed directly by the clients.

The above splitting of responsibility of the validation mechanism, makes the code of a channel simpler and more modular than that of the meta-object. Operations channelRead and channelWrite are very similar, they differ only in the encapsulated information and the use they make of it. In our example channels and meta-objects are similar because the validation phase is simple; however, it should be noted that the amount of information encapsulated and managed by

Listing 4 Java code for the inner layer

```
public class operations {
  int IDClient;

  public static Object rem_server;

    // Here the remote channel object is bound to the URL

  public Object Binding(String str) {
    return Naming.lookup("//" + str);
  }

   // This function invokes the remote methods in TheServer

  public Object rmi(int method, int ClientName) {
    if (method == 0) return ((InterCS)rem_server).Method_Read(ClientName);
    else return ((InterCS)rem_server).Method_Write(ClientName);
  }

}
```

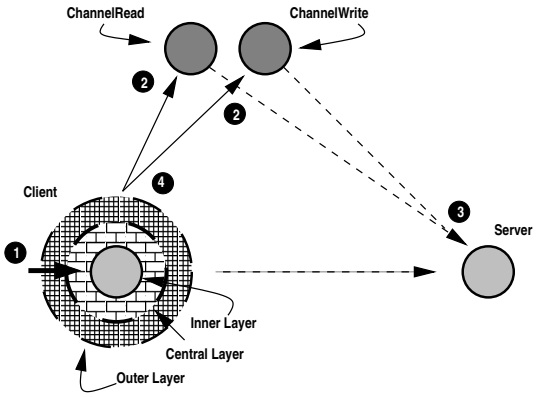

Fig. 4. Java structure for the channel reification approach

a single channel has decreased.
When a client issues the call (see Fig. 4):

$$\text{permission} = \text{server.methodRead}();$$

the control is dispatched to the central layer of the client (step ❶ in Fig. 4) which determines the right channel to be invoked on the basis of the request kind. The request is then forwarded to the channel while the client idles until a reply is sent back to it (step ❷ in Fig. 4). The channel validates the request and, if legal, forwards it to the server (step ❸ in Fig. 4). When the channel receives the reply from the server, it forwards it back to the client (step ❹ in Fig. 4). The server is unaware of the validation process: it executes only filtered requests. Thus a

Listing 5 Java code for the central layer

```
public class reflectiveLayer extends operations {

  Boolean Right;

  private InterCCRead rem_chr;
  private InterCCWrite rem_chw;

    // Here the remote channel object is prepared to bind the URL
    // (see the Operations class for the actual binding )

  public Object Binding(String str) {
    rem_chr = (InterCCRead) super.Binding("CCR"+IDClient+"-"+IDServer);
    rem_chw = (InterCCWrite) super.Binding("CCW"+IDClient+"-"+IDServer);
    return null;
  }

    // Here the rmi method invokes the metaBehavior function in the Channel
    // classes, where the checking about client permissions is performed

  public Object rmi(int method) {
    if (method==0) Right = rem_chr.metaBehavior(method);
    if (method==1) Right = rem_chw.metaBehavior(method);
    return Right;
  }

}
```

good *separation of concerns* and implementation modularity is achieved.

Security models based on communication flow [5] can be easily modeled by the channel reification model: in particular, models based on the concept of *flow* such as those of Escort of Scout [19] and Corps [16]. A path is a first class object encapsulating data flowing through a set of modules. A path is a logical channel made up of data flow connecting several modules. Other two security mechanisms adopted in Escort are *filters* and *protection domains*. A filter restricts the interface between two adjacent modules. However, filters include no mechanism to ensure that a module does not bypass the interface by directly accessing the memory of the other module. A protection domain is a boundary drawn between a pair of modules to ensure that the mutual access is performed only through the defined interface. Filters and protection domains may be modeled respectively by standard reflective channels and *protected* channels, i.e., channels executing in a different address space. The concept of path is more complex and requires an extension of the channel reification mechanism. A channel controlling a path may be obtained by piping or composing the channels controlling the sequence of modules forming a path or by defining a complex channel successively controlling the interface of a sequence of modules in a path. Another approach could be based on a three-level reflective tower, but this approach increases the system complexity without significant advantages.

Listing 6 Java channel code

```
public class channelRead {
  int IDClient, IDServer; Boolean result = null; String REF; InterCS rem_objS;

  // Method Constructor with two parameters used to identify channel's client and server
  public channelRead(int c, int s) {
    IDClient = c; IDServer = s;
  }

  public Boolean metaBehavior(int MethodType) {
    if (result == null) {    // we have to read access matrix in order to initialize channel
    ...
    }
    if (cell == 34) {              // 34 is the int code for "R"
      System.out.println("\nYou have the READ permission !");
      return rem_objS.Method_Read();
    else {
      System.out.println("\nYou don't have the READ permission !!");
      return Boolean.FALSE;
    }
  }

  public static void main(String args[]) {
    channelRead rem_obj;

      // creation and installation of the security manager
    System.setSecurityManager( new RMISecurityManager());

      // Here the remote TheServer object is bound to the URL
    rem_objS = ( InterCS ) Naming.lookup("//CS");

      // this is the server set-up for TheClient-channelRead system
    REF = "//CCR"+args[0]+args[1];

      // creation of an instance of the remote class
    rem_obj = new channelRead(Integer.parseInt(args[0]), Integer.parseInt(args[1]));

      // creation of the registry-remote object binding
    Naming.rebind(REF, rem_obj);
    System.out.println("Reading channel["+args[0]+","+args[1]+"] waits for some requests");
  }

}
```

4 Evaluation

Using computational reflection to include an authorization mechanism into a software system offers many advantages during software development. We can specify, develop, implement and test the modules which implement authorization mechanisms separately from the rest of the application. In this way we encourage reuse and improve software stability. Moreover, the authorization process is hidden to the application entities, thus the code of such entities is simplified. While this can be accomplished by current DBMS authorization systems, we are now controlling access to all executing entities, not just the application's persistent data.

The advantages from the security point of view are that only the entities performing the validation of the authorizations of an object know its authorizations

constraints. In this way authorization leaking is minimized.
A first drawback is that flexibility has a cost in terms of efficiency due to repeated method activation. The Achille's heel of using reflection to realize security could be its implementation mechanism. In fact, as it is stressed by our examples, each reflective approach has a mechanism, which permits to forward each request to meta-entities. These trap actions (shift-up and shift-down) are critical actions. It is possible for a malicious user to intercept the trap and to hijack the request to another entity bypassing the authorization system and illegally authorizing each request. For this reason it is important to protect the meta-level and the access to it and for the same reason it is important to limit the meta-entities communication and, thus, using a reflective model having the global view feature. One way to avoid this attack consists of implementing the whole meta-computation, and in particular the trap actions, as protected routines. From another point of view this drawback can be converted into an advantage of the model with respect to standard authorization models, because it minimizes the vulnerable points to known system locations that can be more efficiently protected. The possibility of using different address spaces (e.g., different processes) to implement the two reflective layers with kernel system intervention for exchanging information represents an improvement to the security of the complete system.
Some recent proposals to control actions of Java applets have similar purposes to our proposal. In those systems, e.g., in [13], execution domains are created and enforced for specific applications using downloaded content. However, the objective of these approaches is to control access to operating system resources, e.g., files, memory spaces, ..., not to control high-level actions between objects.

5 Conclusions

Reflection offers several advantages when used to model authorization mechanisms. Its main advantages are separation of concerns and modularity. Authorization mechanisms can be designed within the application from early development stages, but, at the same time, they can be maintained separate both from the logical and implementative point of view. This fact improves reusability of both functional and authorization software and supports an independent testing of both. More important, it controls access to all executing entities, not just to persistent data.
Another advantage is the ability of implementing a protection layer around the authorization software, thus making the system more robust to unauthorized attempts to change role rights. Obviously, there are also drawbacks: the first is a reduced execution efficiency; flexibility costs in efficiency. The second problem could be presented by the protection mechanism around the authorization layer (meta-level). Running it in a different address space may make programs too inefficient for most applications. Thus, more efficient protection mechanisms, not performing a complete context switching, should be designed. Hardware capability systems appear promising for this purpose.
Moreover, the existence of such a protection makes more *understandable*, to

malicious users, *where to address attacks to the fortress* and easier to discover Achilles' heels.
Future developments are represented by complete workflow authorizations combining specific sequences of service requests: they require that the meta-entities controlling the activated server objects, interact with each other for discriminating legal sequences from illegal ones. More complex authorization schemes may require the introduction of meta-meta-levels, i.e., to raise the reflective tower and the system complexity.
Finally, some prototyping of the proposed architecture and practical experiments will improve the understanding of the role played by reflection in the implementation of authorization systems of high assurance. In particular, its possible use to control the actions of downloaded content would be of high practical interest. Another possible use is for the control of information flow in object-oriented systems as proposed in [4].

Acknowledgments

A preliminary version of this work appears in the proceedings of the 1^{st} ECOOP Workshop on Distributed Object Security, pages 35-39, Belgium, July 1998.

References

[1] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: a Reflective Approach to Fault-Tolerant Software Development. In *OOPSLA'95 (poster section)*, page 137, Austin, Texas, USA, on 15th-19th October 1995. ACM. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.

[2] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: A Reflective Model for Distributed Computation. In *proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, 98CH36191, pages 32–36, Phoenix, Arizona, USA, on 16th-18th February 1998. IEEE.

[3] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachussetts, second edition, December 1997.

[4] Elisa Bertino, Sabrina De Capitani di Vimercati, Elena Ferrari, and Pierangela Samarati. Exception-Based Information Flow Control in Object-Oriented Systems. *ACM Transactions on Information and System Security (TISSEC)*, 1(1), November 1998.

[5] W. E. Boebert and R. Y. Kain. A Pratical Alternative to Hierarchical Integrity Policies. In *proceedings of 8th National Computing Security Conference*, Gaithersburg, October 1985.

[6] Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.

[7] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[8] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *proceedings of the workshop section*, in IJCAI'95 (International Join Conference on AI), Montréal, Canada, August 1995.

[9] Eduardo B. Fernandez and J. C. Hawkins. Determining Role Rights from Use Cases. In *proceedings of the 2nd ACM Workshop on Role Based Access Control (RBAC'97)*, pages 121–125, November 1997.

[10] Eduardo B. Fernandez, Maria M. Larrondo-Petrie, and Ehud Gudes. A Method-Based Authorization Model for Object-Oriented Databases. In *proceedings of the OOPSLA'93 Workshop on Security in Object-Oriented Systems*, pages 70–79. ACM, 1993.

[11] Eduardo B. Fernandez, Rita C. Summers, and Christopher Wood. *Database Security and Integrity*. Addison-Wesley, Reading, Massachusetts, 1981.

[12] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Communication of the ACM*, 19(8):461–471, August 1976.

[13] Trent Jaeger, Nayeen Islam, Rangachari Anand, Atul Prakash, and Jochen Liedtke. Flexible Control of Downloaded Executable Content. http://www.ibm.com/Java/education/flexcontrol, 1997.

[14] Butler W. Lampson. Protection. *Operating System Review*, 8(1):18–34, January 1974. Reprint.

[15] Pattie Maes. Concepts and Experiments in Computational Reflection. In *proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*, pages 147–156. ACM, October 1987.

[16] Edwin Menze, F. Reynolds, and F. Travostino. Programming with System Resources in Support of Real-Time Distributed Applications. In *proceedings of the 1996 IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pages 36–45, Laguna Beach, Ca, February 1996. IEEE.

[17] Ravi Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, February 1996.

[18] Susann Sonntag, Hermann Härtig, Oliver Kowalski, Winfried Kühnhauser, and Wolfang Lux. Adaptability Using Reflection. In *proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 383–392, 1994.

[19] Oliver Spatscheck and Larry L. Peterson. Escort: A Path-Based OS Security Architecture. Technical Report TR-97-17, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, November 1997.

[20] Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.

[21] Robert J. Stroud and Zhixue Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, Inc., 2000 Corporate Blvd.,N.W., Boca Raton, Florida 33431, 1996.