Aspects Can Be Efficient: Experience with Replication and Protection

Fabienne Boyer, Sara Bouchenak, Noel de Palma, Daniel Hagimont

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Aspects Can Be Efficient: Experience with Replication and Protection

Fabienne Boyer, Sara Bouchenak, Noel De Palma and Daniel Hagimont

N° 4651

Novembre 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4651--FR+ENG

Aspects Can Be Efficient: Experience with Replication and Protection

Fabienne Boyer, Sara Bouchenak, Noel De Palma and Daniel Hagimont

Thème 1 — Réseaux et systèmes
Projet Sardes

Rapport de recherche n° 4651 — Novembre 2002 — 31 pages

Abstract: Separation of concerns, which aims at separating different aspects involved in complex applications, is a general trend in software programming. It allows a given aspect to be programmed in a more or less isolated manner from the functional code of an application. This trend has been studied by the AOP (Aspect Oriented Programming) community, through the provision of language supports for programming and composing aspects. It has also been experi-mented in the context of component-based middleware, which usually address system-related aspects (e.g., transactions, security, persistence, etc). In both domains, most implementations of separation of concerns involve indirection objects and extra method calls that incur a non-negligible performance over-head. While performance was initially not the main motivation of "separation of concern environments", we believe that it is possible to efficiently integrate as-pects in such environments. In this paper, we report on an experiment which aims at optimizing aspects through code injection techniques. We consider two aspects, replication and access control, and present a preliminary performance evaluation which confirms that the overhead can be significantly reduced.

Key-words: components, AOP, optimization, code injection

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Les Aspects Peuvent Etre Efficaces : Expérimentations avec la Duplication et la Protection

Résumé : La séparation d'aspects permet au programmeur d'implanter les aspects d'une application de façon plus ou moins indépendante du code métier de l'application. La séparation d'aspects a été étudiée par la communauté AOP (Aspect Oriented Programmin), à travers un support langage de programmation et de composition d'aspects. La séparation d'aspects a également été étudiée dans le contexte des intergiciels à composants, qui traitent plus particulièrement la mise en oeuvre d'aspects système (ex. transactions, sécurité, persistance, etc.). Dans les deux cas, la plupart des mises en œuvres proposées pour la séparation d'aspects repose sur des objets d'indirection et des appels de méthodes supplémentaires. Ceci implique un surcoût non négligeable sur les performances des applications. Même si les performances n'étaient pas la motivation principale des "environnements de séparation d'aspects", nous pensons qu'il est possible d'intégrer des aspects de façon efficace dans de tels environnements. Nous utilisons pour cela des techniques d'injection de code. Dans ce rapport, nous décrivons nos expérimentations concernant l'optimisation de deux aspects : la duplication et la protection. Nous présentons également les résultats préliminaires de notre évaluation de performances, résultats qui confirment que de telles techniques peuvent considérablement réduire le surcoût sur les performances.

Mots-clés : composants, programmation par aspects, optimisation, injection de code

1 Introduction

1.1 Objectives

Complex applications generally include many aspects. A general trend in software development is to separate, as long as possible, these different aspects from applica-tions' code in order to improve the quality of software which is easier to maintain [9].

The principle of aspect separation has been studied in several contexts by different research communities:

- Aspect-oriented programming (AOP). AOP aims at allowing different aspects of complex software to be programmed separately and then woven into a single appli-cation program. AOP led to language and runtime supports for programming and composing aspects, such as the AspectJ [9] environment, where the declaration of an aspect may include join point declarations, i.e., the place/moment the aspect takes place, and advice declarations, i.e., the aspect-related code.

- Component-based programming (CBP). CBP aims at improving code evolution and reuse by enabling configuration of complex component-based architectures and association of non-functional properties, namely aspects, with applications' business components. CBP led to the development of middleware environments such as CCM [14] and EJB [19], where aspects are described separately from applications' business code, and associated at runtime with applications through configurable ob-jects (containers) that link aspects to business code components.

Researchers in AOP have focused on providing language support for aspect pro-gramming and weaving while in the context of CBP, researchers have mainly focused on the integration of system-related aspects (e.g., transactions, security, persistence, etc.). However, in both contexts, most of the implementations rely on intermediary objects for the integration of aspects into the base code of applications. Intermediary objects, that from now on we will call indirection objects, are responsible for a sig-nificant overhead at runtime. Our objective is to investigate techniques for efficient integration of aspects in the base code of applications. A long term objective is to provide a generic facility which would include language support for describing as-pects, and enabling automatic optimized integration of aspects.

A first step in this direction has been to study the feasibility of optimizations for two aspects: object replication and protection. In previous experiments, we have shown that both replication and protection can be managed as orthogonal aspects relying on indirection objects [7][6]. In this paper, we report on our experience in optimizing these two aspects. Optimization aims at removing indirection objects usu-ally used in implementations, thus improving performance at runtime . We show that, although this task may be complex for some aspects, optimization remains possible.

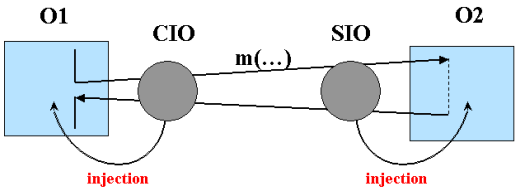

Figure 1: Non-functional code injection

1.2 Approach

In AOP and CBP environments, several successive indirection objects may be in-volved in the implementation of aspects, forming a chain (more or less long) between an invoking object and an invoked object . To place ourselves in a general and simpli-fied framework, we consider the case where two indirection objects are involved: the first one, associated with the invoking object, is called the Client Indirection Object (CIO) and the second, associated with the invoked object, is called the Server Indirec-tion Object (SIO) . In some cases, only the CIO or the SIO may be present. But in the two aspects that we further consider, these two indirection objects are necessary to integrate the aspects.

In Figure 1, CIO and SIO capture interactions between objects and allow the association of aspect-related code with the invoking and invoked objects. This may consist in controlling binding to a replicated object as with the replication aspect (see sec-tion 2) or implementing access control checks as in the protection scheme (see section 3). The general idea here is to inject the aspect-related code within the code of the application, and thus eliminate indirections, which reduce performance overhead. Code injection can be performed at compile time or dynamically at load time.

Code injection techniques that we experimented apply at the level of Java bytecode. In the rest of the paper, and for clarity purpose, we will illustrate our experiments through Java source code.

1.3 Roadmap

The rest of the paper is organized as follows. Sections 2 and 3 respectively describe our experiments with the replication and protection aspects. Each section includes a description of the aspect, its implementation using indirection objects and the pro-posed integration based on code injection. Section 4 discusses code injection tech-niques used in both experiments and shows that a common injection scheme appears. Section 5 presents our preliminary performance evaluation, and section 6 discusses the related work. We finally conclude in section 7.

```
interface Obj_itf1 {
    void foo(Obj_itf2 obj);     read
    Obj_itf3 bar();             write
}
```

Figure 2: Description of access modes

2 Replication aspect

Replication of shared distributed objects has many objectives, such as performance improvement by transforming remote calls into local calls [8], or service availability in order to face disconnections [11] and tolerate faults [13].

Javanaise is a Java-based system that provides distributed shared object applica-tions with replication [8] . The Javanaise system transparently manages, on the one hand, the replication of distributed shared objects, and on the other hand, the consis-tency of replicated objects. Therefore, an application programmer manipulates its objects as in a centralized/non-replicated/non-shared scheme.

In Javanaise, when an object on a client node requests for the first time another object located on a server node, and if the requested object is "replicable", the requested object is transparently brought on the requesting node and cached until invalidated by the consistency protocol. Later, if the local replica is accessed by the requesting object subsequently to an invalidation (because of modification of another replica of the same object), an updated replica is brought on the requesting node. Thus, managing object replication requires mechanisms for:

- object binding when faulting on objects at first access or access after invalidation
- object synchronization for invalidating objects (after modification of another rep-lica) and updating objects to ensure consistency.

Binding and synchronization mechanisms are built in the Javanaise system relying on a replication server [7]; and the implemented synchronization mechanism is based on the multiple-readers/single-writer entry consistency protocol [2]. Thus, the applica-tion programmer specifies the access mode associated with its application's objects, i.e., read/write modes of the application's methods. This is done through a Java-based extended IDL (Interface Description Language), as illustrated by Figure 2, where method foo is accessed in read mode and method bar is accessed in write mode.

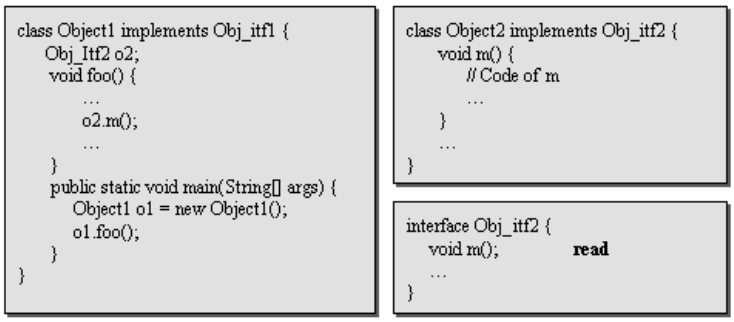

Figure 3: Example: Object1 references Object2

2.1 Indirection-based implementation of replication

A first implementation of the replication aspect in the Javanaise system follows an indirection object scheme [7]. Here, transparent replication, and more precisely trans-parent object binding and synchronization, are implemented using a pair of indirection objects (CIO-SIO, c.f., section 1.2) that represents inter-object references [7]. When a caller object references a callee object, CIO and SIO are dynamically generated and transparently inserted between the caller and the callee objects. This transparency is ensured by the fact that the caller object views CIO as being the callee object (CIO implements the same interface as the callee). Figure 3, and more precisely Object1 and Object2 classes, illustrate a sample program where an object o1 calls a method foo which itself calls a method m on an object o2. In a centralized, non-shared, non-replicated object environment, the caller object o1 would directly reference the callee object o2. But when implementing transparent replication using the indirection object-based scheme, and in the case object o2 is replicable, object o1 references CIO which itself references SIO that references object o2.

Let us consider the access mode as presented by the interface description in Figure 3. In this case, the associated CIO and SIO are built as illustrated by Figure 4. Here, CIO and SIO are respectively intended to implement object binding (between o1 and o2) and object synchronization (between o2 replicas):

- CIO: object binding. CIO manages binding to a callee object that may be brought dynamically from remote nodes. Dynamic binding is implemented as follows. First, CIO contains a unique identifier associated with object o2 (id_o2 in o2_CIO's code, see Figure 4), and a reference to the associated SIO (reference o2 in o2_CIO's code, see Figure 4). If the reference to SIO is null, it means that it is the first time that object o1 accesses object o2. In this case, a copy of object o2 is fetched, either locally if the object is already cached or remotely from the Javanaise server using o2's unique identifier.

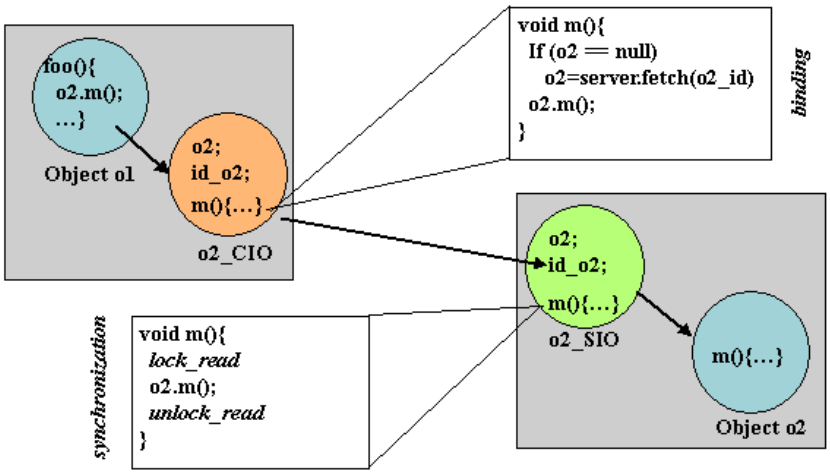

Figure 4: Implementation of replication with indirection objects

- SIO: object synchronization. SIO manages synchronization, i.e., invalidation and update, of the o2 replicated shared object. This is implemented as follows. First, SIO contains the unique identifier associated with object o2 (id_o2 in o2_SIO's code, see Figure 4), and a reference to the replicated object (reference o2 in o2_SIO's code, see Figure 4). With the entry consistency protocol, and according to the access modes specified in the replicated object's interface, the methods of this replicated object are bracketed with lock_read/lock_write and unlock_read/unlock_write calls. The lock_read/lock_write procedures ensure that a consistent copy of object o2 and a lock are present on the local node. If a consistent copy and lock are found locally (cached), then the lock is taken and the method in-vocation is invoked on o2. If not, it means that the local copy was invalidated. In this case, a consistent copy and a lock are fetched from the Javanaise server using o2's unique identifier. The full explanation of this protocol can be found in [7].

2.2 Injection-based implementation of replication

In the previous section, we showed how the replication aspect is built using indirec-tion objects. Thanks to this scheme, replication is implemented transparently. But it presents two main drawbacks:

- The multiplication of indirection objects (CIO and SIO) used to implement interactions between objects; this incurs additional memory consumption.

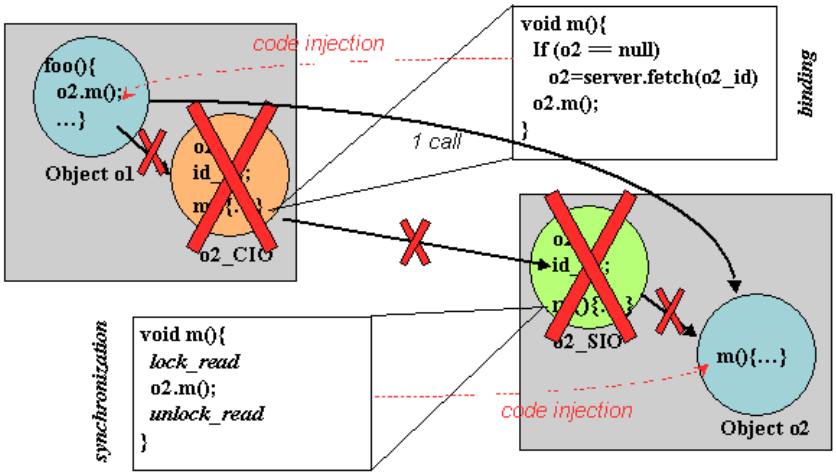

Figure 5: Implementation of replication with code injection

- The multiplication of method calls. Indeed, the call of method m from object o1 on object o2 is transformed into three method calls: first o1 calls method m on o2_CIO, then o2_CIO calls method m on o2_SIO, and finally o2_SIO calls the ef-fective method m on object o2. This results in lower performance.

In this section, we propose an optimized scheme in which indirection objects and additional method calls are removed while transparently keeping the replication as-pect. Our approach leads to a second implementation of the Javanaise system where we use code injection techniques. This is illustrated by Figure 5, where the code re-lated to object binding, i.e., CIO's code, is injected into the caller object's code, and the code related to object synchronization, i.e., SIO's code, is injected into the callee object's code. In the following, we discuss how code injection takes respectively place in the caller object and the callee object.

2.2.1 Injection of binding code into caller

The code of the caller object is transformed by our code injection scheme as follows:

- For each replicable object that is referenced in the caller object's code, as a field or as a method's local variable, the declaration of a new variable is added and a code associated with this new variable is injected. The new variable represents the unique identifier associated with the replicable object. And the code associated with this new variable is injected each time the reference to the replicable object is assigned: the injected code manages the assignment of the new variable. With this scheme,

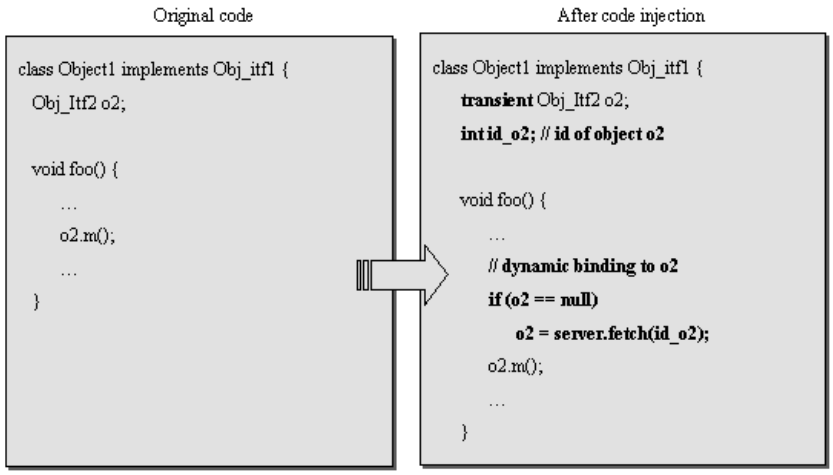

Figure 6: Injection of binding code into caller code

the caller object does not point to CIO but rather has a straight reference to the callee/replicable object.

- The management of dynamic binding between the caller object and the callee ob-ject is implemented as follows. Before each method call on the replicable object by the caller object, a code that checks binding is injected into the caller's code. The injected code checks if the reference to the replicable object is null; if it is, the Javanaise server is contacted for a binding request using the injected variable that represents the unique identifier of the replicable/callee object.
- When a reference to a replicable object is transmitted by the caller to the callee object, as a method parameter for example, the identifier associated with the replic-able object is also transmitted. This is performed by adding a new parameter to the method signature.

Figure 6 illustrates the transformation applied by our code injection scheme on the program of the caller object o1 that was previously introduced in Figure 3. Here, a new variable id_o2 is added and the code that checks binding to o2 is injected before the call of method m on object o2.

2.2.2 Injection of synchronization code into callee

The transformation applied by our code injection scheme to the code of the replic-able/callee object is straightforward. First, a new field which represents the unique identifier of the

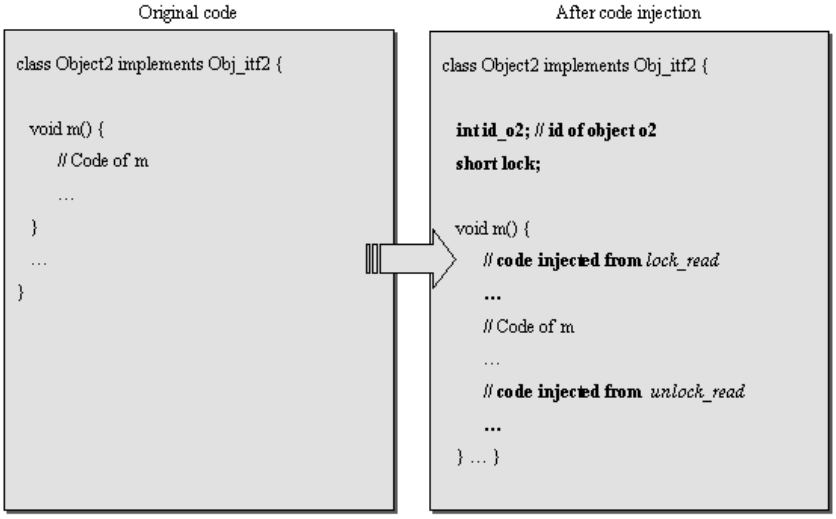

Figure 7: Injection of synchronization code into callee code

replicable object is added. The code of the lock_read/lock_write/unlock synchronization procedures is injected into the associ-ated methods, with respect to the access modes of these methods. A lock field is also added; it is managed by the synchronization procedures and tells them whether a lock is cached on the local node. If the required lock is not cached, the lock and the most recent copy of the replicable object are fetched from the Javanaise server using the object identifier. If the lock is already cached, no communication with the server is required and the executed code is a simple (synchronized) check of the lock.

Figure 7 illustrates the transformation applied by our code injection scheme on the program of the callee object o2 that was previously introduced in Figure 3. Here, two new variables id_o2 and lock are added and the lock_read and unlock_read proce-dures are injected in method m.

3 Protection aspect

The second aspect that we experimented is protection. The purpose of protection is to control interactions between mutually suspicious objects. The protection mechanisms control, for each object in the system, the objects it can access and the operations it can invoke on them. In a previous work, we have proposed a protection model where protection is built as an orthogonal aspect that is transparent to applications. This model is called hidden software capabilities [6] and is detailed in the following.

3.1 Hidden software capabilities

In order to illustrate the protection aspect, let us consider the example of a printing service that allows a client to print out a file; this service provides:

- a print operation, for printing a text, and
- an init operation, for resetting the underlying printer.

In a protection-free application, any user is allowed to access all the operations pro-vided by the printing service. Symmetrically, and for the particular case of the print operation, there is nothing to prevent the printing service modifying the printed text.

Now, let's consider a protection policy has to be added to the application with the following properties:

- On the one hand, to differentiate between two groups of users: administrators and general clients. Unlike administrators who are granted full access to the service, a protection policy allows the printing service to restrict the access for general clients to printing texts while resetting the printer is denied.
- And on the other hand, to take into account mutual suspicion between the printing service and clients. Here, when a client requests the print operation of the printing service, he allows the service to read the text to be printed but not to modify it.

In other words, two views of the "interactions" between the client and the printing service exist, depending on the "viewer" and the restrictions it puts for the protection purpose:

- The client views the "interactions" as follows: he can access the operations pro-vided by the printing service but he restricts access to the printed text to read-only mode, i.e., the printing service is only capable to read the printed text.
- The printing service views the "interactions" as a restriction to print operations, i.e., the client is only capable to request printing operations.

In our hidden software capability model, where protection is implemented as an orthogonal aspect of applications, capabilities and views are defined as follow.

3.1.1 Capabilities

A capability associated with an object provides access rights on that object. For pro-tection purpose, in order to be allowed to access an object, an application must own a capability to that object with the required access rights.

Initially, when an object is created, the capability associated with this object is re-turned to the creating application; it usually contains all the rights on the object. The capability can then be used to access the object.

The capability can also be copied and passed to another application (through method invocation), providing it with access rights on that object. And when a capa-bility is copied restricting the rights associated with the copy, the rights of the receiv-ing application on the object are limited. In the printing example previously presented, the client has the capability to access his text in read-write mode but he restricts the access rights of the printing service to read-only mode (for general clients).

3.1.2 Views

In order to describe capability management, we rely on the definition of views. A view of an object describes the following:

- The set of authorized operations/methods.
- For every object/reference parameter (or result) of an authorized method, the capa-bility associated with this parameter. The capability to be passed with the parameter is described with a view.
- If no capability is associated with an object parameter (or result) of an authorized method, it means that no access restriction applies on that parameter.

Two interacting objects define their own protection policies with two views and both are taken into account at runtime.

3.1.3 Extended IDL

Let us consider again the printing service as illustrated by Figure 8, where a Client object submits a Text object, for printing purpose, to a Printer object, with the protec-tion policy described before.

A capability on the Printer object is given to clients providing them with the right to print texts. When the Client calls the print method on the Printer object, with a Text object as an input parameter, the Client passes to the Printer object a read-only capa-bility (Text_capa) on the Text object (see step 1 in Figure 8). This capability allows the Printer object to read the content of the text (see step 2 in Figure 8).

In order to describe the applied protection rules, i.e., capabilities and views, sepa-rately from the functional code of applications, we use a Java-based extended IDL (Interface Description Language). This IDL is illustrated by Figure 9 which shows the views related to

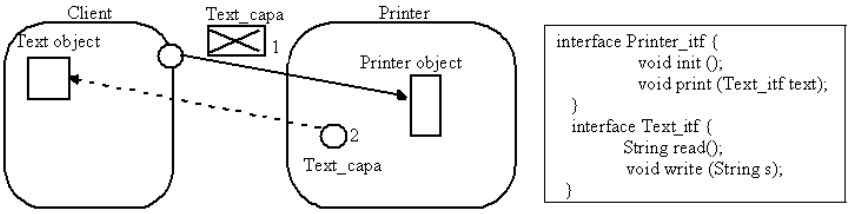

Figure 8: Protection in printing service

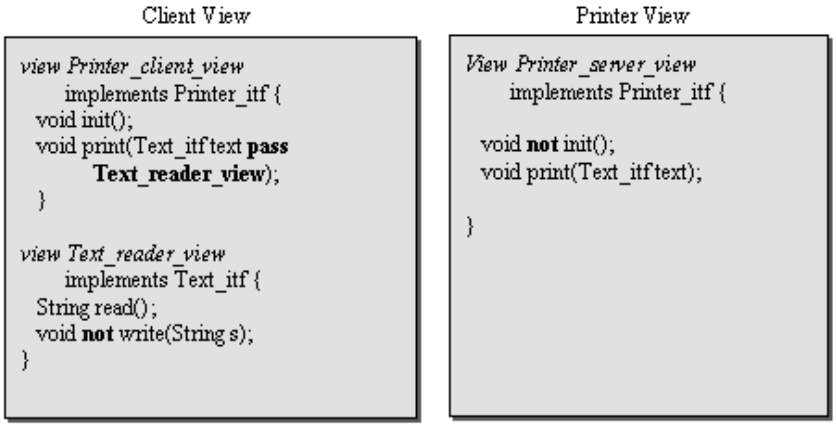

Figure 9: The Printer example views

the printer example. Each view "implements" the Java interface that corresponds to the type of the object it protects. A not before a method declaration means that the method is not permitted. When an object reference is passed as pa-rameter in a view, the programmer can specify the view to be passed with the refer-ence using the key-word pass. If no view is specified, it means that no restriction is applied to this reference.

Printer_client_view represents the protection policy from the client point of view: it specifies that a capability with Text_reader_view must be associated with the text pa-rameter when the print method is invoked. And Text_reader_view only authorizes read operations on texts. Similarly, Printer_server_view represents the protection policy from the printer point of view. It prevents clients from calling the init method on printers. Notice here that the client has no reason to forbid itself from using the init method, it is a decision taken by the printer; this is why no restriction regarding init is made on Printer_client_view.

3.2 Indirection-based implementation of protection

For the implementation of the protection model presented above, we relied on the fact that Java object references are almost capabilities (Java is strongly typed). Indeed, since Java is a safe language, it does not allow object references to be forged. This implies that if an object O1 creates an object O2, object O2 will not be accessible from other objects of the Java runtime, as long as O1 does not explicitly export a reference to object O2 towards other objects. References to O2 can be exported (as a parameter) when an object invokes O1 or when O1 invokes another object. Thus, a Java object reference can be seen as a capability. However, they are all-or-nothing capabilities since it is not possible to restrict the set of methods that can be invoked using this reference. We relied on indirection objects in order to implement our capa-bilities.

Our implementation relies on CIO and SIO indirection objects as described in sec-tion 1.2. CIO and SIO respectively correspond to the caller protection view and to the callee protection view for a particular interaction, (i.e., a method invocation). They are thus inserted between the caller and the callee. For each view defined by an applica-tion, the class of the indirection object is generated (by a pre-processor). These indi-rection objects are dynamically inserted between the referenced object and the object which contains the reference, each time a reference is transmitted.

When a reference to an object is passed as an input parameter of a method call, the reference of the indirection object is passed instead of the reference to the object it-self. This indirection object implements all the methods declared in the interface of the view. It defines an instance variable that points to the actual object and which is used to forward the authorized method calls. If a forbidden method is invoked on an in-stance of an indirection object, then the method raises an exception.

The reference to the indirection object, which is passed instead of the reference pa-rameter, is inserted by the CIO of the interaction. In Figure 10.a, the invocation of o2 performed by App1 passes a reference to o1 as parameter. CIO1(o2), which corre-sponds to the protection policy of App1 for invocations of o2, inserts SIO1(o1) instead of the parameter o1. Therefore, indirection objects that are associated with reference parameters are installed by indirection objects that are used upon method invocations.

Conversely, when a reference is received by an application, a reference to an indi-rection object is passed instead of the received parameter; Figure 10.b, SIO2(o2), which corresponds to the protection policy of App2 for invocations of o2, inserts CIO2(o1) instead of the received parameter. Therefore, two indirection objects, S1O1(o1) and CIO2(o1), are inserted between the caller and the callee for the parame-ter o1 passed from App1 to App2. These two objects behave as follows:

- SIO1(o1): it verifies that only authorized methods can be invoked by App2 and it inserts indirection objects on the account of App1 for the parameters of invocations on O1 performed by App2.
- CIO2(o1): it inserts indirection objects on the account of App2 for the parameters of invocations on o1 performed by App2.

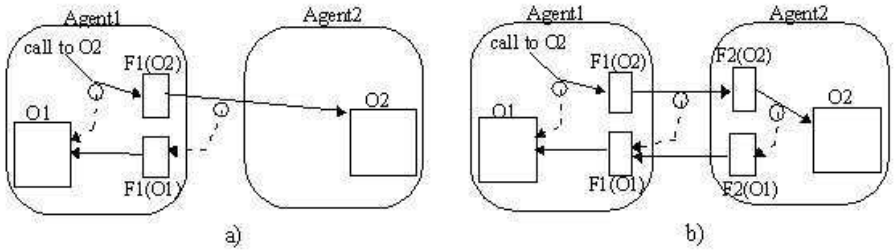

Figure 10: Management of indirection objects

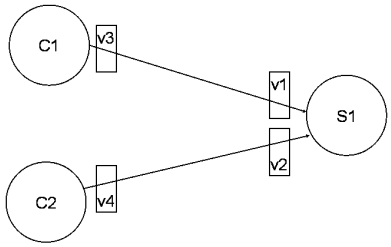

Figure 11: Indirection objects for the printer example

In the case of the printer application, when a Client object invokes a Printer object, it passes a reference to a Text object as a parameter of the print method. Thus, the involved indirection objects are Printer_server_view, Printer_client_view and Text_reader_view, as illustrated in the figure below:

Finally, correspondence between the definition of views made by the protection programmer and the indirection objects is rather direct. The Java classes related to the implementation of the protection aspect in the printer example, and corresponding to the IDL description presented in Figure 9, are given by Figure 12. Printer_client_view and Printer_server_view classes respectively implement the client and server views used to protect the interaction between a client and a printer service. When invoking the print method, Printer_client_view installs an indirection object (Text_reader_view) to protect the text object. The reference to the text object trans-mitted as a parameter of the print method is indeed a reference to the indirection ob-ject Text_reader_view.

IO for the client

```
public class Printer_client_view
    implements Printer_itf {
  // reference of the component
  Printer_itf comp;

  public Printer(Printer_itf p) {
    comp = p;
  }
  public void init() {
    comp.init();
  }
  public void print(Text_itf text) {
    Text_reader_view ois = new
     Text_reader_view (text);
    comp.print(ois);
  }
}

public class Text_reader_view
    implements Text_itf {
  // reference to the component
  Text_itf comp;

  public Text_reader_view
      (Text_itf c) {
    comp = c;
  }
  public String read() {
    return comp.read();
  }
  public void write(String s) {
    Exception !!!
  }
}
```

IO for the Printer

```
class Printer_server_view
    implements Printer_itf {
  // reference to the component
  Printer_itf comp;

  public Printer_server_view
      (Printer_itf c) {
    comp = c;
  }

  public void init() {
    Exception !!!
  }

  public void print(Text_itf text) {
    comp.print(text);
  }
}
```

Figure 12: Code of the indirection objects of the print server

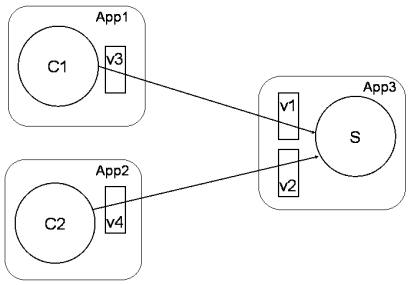

Figure 13: A simple example

3.3 Injection-based implementation of protection

In this section, we show how to use code injection technique to implement our capa-bility-based access control as an orthogonal aspect without using indirection objects.

3.3.1 Avoiding indirection objects

We illustrate our approach by using a simple example depicted Figure 13, then we described the injection mechanisms and detail the injection in the case of the printer application presented in section 3.1.

In the example Figure 13, the object S provides a service to objects C1 and C2. Since S trusts C1 more that C2, it provides the service through two different views: the view v1 provides the access rights to C1 and the view v2 provides more restricted access rights to C2. C1 and C2 do not trust S. Consequently, they also provide a view to protect themselves (respectively v3 and v4).

When we implement the protection mechanism using indirection objects, any bind-ing between two objects provides two indirection objects corresponding to both points of view. We have thus an indirection object for each view:

- The code of the indirection object placed on the server side (v1 or v2) contains the protection policy corresponding to the view it implements.
- The code of the indirection object placed on clients side (v3, v4) contains the protection policy enforced from the client point of view.

When we implement the protection mechanism using the injection technique, we only have C1, C2 and S to implement each view:

- The code enforcing the protection policy for each server view has to be in-jected in S (i.e. S contains the protection policy for each view it provides). Therefore,

an identifier should be associated with each view, for instance the view_id. These identifiers are used by S to implements the protection policy corresponding to each view (corresponding to each binding).

- The code enforcing the protection policy for the view of C1 (resp. C2) has to be injected in C1 (resp. C2).

Therefore the protection code injection proceeds as follows:

- Any reference to a protected object (a capability) should be extended with the identifiers of the client view and the server view associated with the capability. As a consequence, for each reference variable (field or local variable) pointing to a protected object, we have to inject the declaration of two new variables which represent the two views associated with the capability: the view speci-fied by the object owner (the callee) and the view specified by the owner of the capability (the caller). These two variables are view identifiers (integers). They are assigned when the reference variable is assigned (we inject the code that does it).
- To check that a method invocation is allowed on a protected object, the server view is transmitted at invocation. The code which checks whether this invoca-tion is authorized is injected at the beginning of the invoked method.
- When a reference to a protected object is passed as parameter, we inject in the caller method and in the called method the code which calculates the identifier corresponding to the client view and the server view related to the capability passed as parameter. The object which provides the reference calculates its server view (the view identifier is thus passed as parameter with the reference) and the object which receives the reference calculates its client view. These views calculations depends on the definitions of the views (client and server) associated with the capability used for the current method invocation. As a consequence, when a reference to a protected object is passed as a method pa-rameter, we inject the code which will initialize the client and server view vari-ables on the parameter receiver side.

This is further clarified on the printer example.

3.3.2 The Printer example

In the case of the printer example, when a client receives a reference to the printer (from a name server), it also receives the client and server views corresponding to the capability (the view identifiers). The print method invocation to the printer takes two additional parameters:

- The server view of the reference to the printer. This view (printer_server_view), passed as parameter, allows checking access rights on the printer side.

- The server view corresponding to the text parameter. This view (text_server_view) specifies the access rights that are granted to the printer on the text object. The server view to pass with the text is defined from the client view associated with the printer capability. This client view is identified by printer_client_view in the code below (Figure 14).

On the server side (Figure 15), the printer checks that the access rights, associated with the view received as parameter (printer_server_view), grant access to the method (for init and print).

In the print method, the injected code initializes the view variables associated with the text received parameter. The client view for the text is defined in the server view for the printer (depending on printer_server_view). The server view for the text is received as parameter (text_server_view).

```
public class ClientPrinter {

public static void main(String args[]) {

  // Extended reference to the printer
  Printer_itf printer_ref;
  short printer_client_view;
  short printer_server_view;

  // Extended reference to the text component
  Text text;
  short text_client_view;
  short text_server_view;

  // Fetch the reference from the name server
  ...

  // Create a text component
  text = new Text();

  // printer_server_view: passed for checking
  // the capability on the server side
  // text_server_view: the view passed with
  // the text, depends on the (local)
  // definition of printer_client_view
  switch (printer_client_view) {
    ...
    text_server_view = new reader();
    ...
  }
  printer_ref.print(printer_server_view, text,
                    text_server_view);
  }
}
```

Figure 14: Code of the client after protection code injection

```
class Printer implements Printer_itf {

public void print (short printer_server_view, Text_itf text,
                   short tex_server_view) {
  // Extended reference for the text component
  short text_client_view;
  short text_server_view;

  // Check the capability ... based on
  // the (local) definition of printer_server_view

  if (printer_server_view != ... ) {
      Exception !!!
  }
  // Initialize the views corresponding to the text component
  text_server_view = tex_server_view;
  switch (printer_server_view) {
      ...
      text_client_view = ...
  }
  text.read(this.text_server_view);
  return;
}}
```

Figure 15: Code of the print server after protection code injection

4 Discussion

In this section, we provide a synthesis which summarizes our study of the feasibility of using code injection techniques to improve the performance of the aspects. Having considered the aspects of replication and protection, we outline the functions required to implement these aspects by using code injection techniques.

In the case of the replication aspect, the main requirements that have been identi-fied are:

- Extend the reference of an object with a variable which uniquely identify the object in a distributed way. Any assignment of an object reference implies the assignment of the corresponding identifier. Any object reference passed as pa-rameter of a method implies also the transmission of the identifier.
- Inject, in the caller before each method call on a replicable object, a processing that checks the binding between the caller and the replicated object.
- Inject, at the beginning of each method of a replicable object, a processing that locks the replicated object in the required mode (read/write) and enforce the consistency protocol.
- Add data in an object (object identifier, local lock).
- Add methods in a replicable object to allow an external entity (the replication server) to invalidate the local replica.

In a symmetric way, we identify below the requirement for the capability-based protection:

- Extend an object reference with the view identifiers corresponding to the capa-bility. As in the previous case, any reference assignment or parameter passing implies also the transmission of the identifiers associated with the reference.
- Inject, in the caller before each method call on a protected object, a processing for the calculation of the views associated with the object references passed as parameters.
- Inject, at the beginning of each method of a protected object, a processing for the calculation of the views associated with the object references received as parameters. This injected code should also check the access rights on the method.
- Extends the signature of a method to add new parameters (for example to transmit the server view from the called object to the callee object).

These requirements exhibit a set of common patterns used to optimize applications built with non functional aspects. These patterns can be supplied to the programmers of aspects to customize the aspects implementation at a finer level. They define ab-stract extensions,

which apply to the basic elements of an object model (objects, bind-ing, etc.). These patterns are described below:

Reference extension

An extended reference augments an object reference with several data which can be of any type. Any assignment, transmission or comparison involving an extended refer-ence may be redefined, involving all the elements which belong to the extended refer-ence. This is a way to extend the notion of reference provided by the Java Virtual Machine without changing its implementation.

Object extension

An extended object is an object in which data and\or methods have been added with regard to its initial definition.

Method extension

An extended method is a method in which the code and/or the signature were changed with regard to its initial definition:

- Signature: adding one or several parameters to the method.
- Code: adding some control in the caller and the called method. It can be used for example to add a processing before or after the execution of a method on the callee side, as well as to add a processing before and after a method invoca-tion on the caller side.

To summarize, while object and method extensions have been largely studied and experimented in AOP and CBP environments, the question of extended references is still open.

We notice that it is not sufficient to directly inject the code of the indirection ob-jects (CIO and SIO) in the corresponding objects (client and server). In particular, the optimizations we implement cannot be achieved with classic compilation tools such as the Just-In-Time compilers. JIT uses techniques that expand the called code in the calling code (In-lining). We consider in a first example our replicated shared object service. The calling object references directly the local copy of the called object, in-stead of referencing the CIO. For each of these references, it contains the unique iden-tifier of the called object and it makes the binding test. During the assignment of an object reference, it is necessary to assign the unique identifier associated with the reference. It is the same processing for parameters. This cannot be achieved with a simple in-lining technique. It is necessary to transform the code of objects in a finer way. Let us consider now the management of protected objects. The injection of CIO and of SIO is much more complicated than in the case of replication, because an indi-rection object (client or server) correspond to a view. It is necessary to inject code corresponding to all the potential views in the caller and the callee, and to manage view identifiers which distinguish the views associated with a capability. This work can not be made with a JIT compiler.

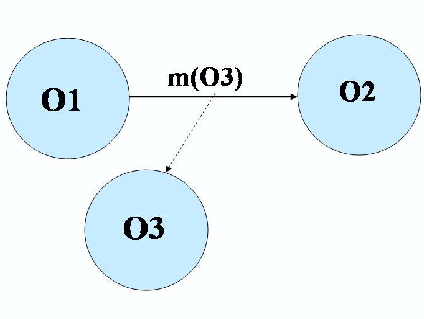

Figure 16: Basic scheme for performance evaluation

5 Evaluation

This section first gives the current implementation status of our prototypes of injec-tion-based integration of aspects, and then describes the performance evaluation we conducted on these prototypes.

5.1 Implementation status

Our prototypes have been developed using BCEL (Byte Code Engineering Li-brary) [1] on top of JDK 1.3. The BCEL is a Java-based library that is intended to give programmers a convenient facility to analyze and manipulate Java bytecode. For example, the org.apache.bcel.generic.MethodGen class provided by BCEL is used to build/instrument applications' methods, and the addLocalVariable method of this class adds a new local variable (or a new parameter) to the method. Similarly, the addField method provided by the org.apache.bcel.generic.ClassGen class adds a new field to a class. BCEL provides many low-level tools for the instrumentation of Java bytecode.

Nevertheless, it is important to notice that for using BCEL, it is necessary to know the basics of the Java virtual machine, such as the class file format, runtime data areas, runtime sub-systems, etc. This is described in the JVM specification [12].

We built two bytecode transformers for orthogonally injecting replication and pro-tection aspects into applications' code using BCEL. Each "injector" accounts for 5000 lines of Java source code. The implemented API is a straight implementation of the injection scheme described in sections 2.2 and 3.3.

5.2 Performance

In this section, we provide the results of performance evaluation in the case of a ba-sic scheme illustrated in Figure 16.

In this scheme, object o1 invokes a method m() on object o2. We consider the case where method m() does not take any parameter and the case where it takes a reference to another object o3 as parameter. The code of method m() is empty. These measure-ments have been done under three conditions:

- with the replication service implemented with indirection objects or with code injection,
- with the protection scheme implemented with indirection objects or with code injection,
- on Java without integration of any aspect. We give the cost of a straight method invocation in order to situate the cost of orthogonal aspect implemented with code injection, compared to the same application code without any orthogonal aspect.

Table 1 presents the resulting performance figures using a 1Gh Pentium processor with 256 Mo of RAM. These results are given for 100000000 iterations over the method call.

These performance figures show a general speedup when using injection code technique. In the following we detail each case.

5.2.1 Protection

In the case of a single method call with no parameter, the speedup is 48%. This speedup is explained because we avoid two indirection calls.

In the case of a method call with a reference parameter, the speedup is 75%. In the version based on indirection objects, when we transmit a protected object reference (to o3) as parameter, o2_client_view has to instantiate o3_server_view to protect o3 .

In the code injection version, we do not have to instantiate any indirection object since the protection code is embedded in the caller and callee objects (however we have to pass new parameters to implement the capability transfer).

5.2.2 Replication

In the case of a single method call with no parameter, the speedup is near 45%. Like for the capability experimentation, this speedup is explained because we avoid two indirection calls.

In the case of a method call with a reference parameter, the speedup is about the same (47%). This is explained by the fact that in the indirection objects version, pass-ing a reference parameter implies passing a reference to a CIO object which can be shared by both referencing objects. Therefore, no CIO creation is required for passing a reference parameter.

The implementation of the replication service has a higher cost than the implemen-tation of the protection service, because the replication service requires costly syn-chronization operations.

These preliminary performance figures are encouraging and they show the per-formance benefits that we can expect from using code injection techniques for the management of orthogonal aspects. This evaluation is ongoing. In future evaluations, we plan to take into account additional parameters like the impact of our optimization techniques on code size and memory consumption.

6 Related work

The overall objective of our work is to investigate techniques used to integrate aspects in the business code of applications. Such techniques have been studied in the context of component-based programming (CBP) and aspect-oriented programming (AOP). In the context of CBP, researchers have experimented with the integration of non-functional properties such as security, transactions or persistence in component-based middleware such as EBJ [19] or CCM [14]. The integration generally relies on indi-rection objects which allow capturing the interactions between components, and there-fore to reify invocations. In the context of AOP, the focus has rather been on the defi-nition of language support for programming aspects (e.g., AspectJ [9]). The integra-tion often relies on extra method calls (on aspect objects) which are injected in the code of the applications before, after or at entry of business method calls. This is also a means to reify invocations. Consequently, in both cases, the reification of business method calls incurs additional method calls which can impact performance. The in-curred overhead can be significantly reduced using an aggressive code injection pol-icy, as described in this paper.

From a more technical point of view, many different projects have experimented with Java bytecode transformation tools in order to inject additional code in applica-tions' functional code. Were notably considered applications resource control [3], distribution [21], and thread migration [20][17]. We believe bytecode engineering tools such as Javassist [5] or BCEL [1] open many perspectives and will play an im-portant role in future middleware systems which aim at allowing runtime adaptation of system services.

From the performance point of view, binary transformation is a technique which al-lows many optimizations. It was previously applied to manage software components sandboxing with the Software Fault Isolation technique [22] which injects binary code that verifies that a component does not address the memory region allocated to an-other component. Software fault isolation allows component confinement without having to manage components in separate address spaces, which would be costly due to address space boundaries crossings. The approach presented in this paper shares many ideas with the software fault isolation proposal, especially the motivations and technical approach.

Another point that should be considered is the domain of reflexive environments, which instrument and reify applications' behavior during execution [18]. Such envi-ronments may be used for managing aspects [10]. However, these environments usu-ally use additional objects at runtime (meta-objects), leading to the same problem as the one we addressed in this paper.

Finally, there are several problems which are addressed in the AOP community, that we have not yet addressed but which should be considered in the near future. One of these issues is aspect weaving [15]. Aspect weaving is a very difficult issue which is yet very open. The next step for our optimizations is to provide a specific language which would allow the description of the optimized integration of one aspect. Such a language support should allow an aspect expert to describe the optimized integration of combined aspects. We plan to investigate in this area. Another issue is dynamic aspect integration [4][16]. Managing aspects in indirection objects provides a means to dynamically add/remove

aspects. Injecting aspects in the business code of the ap-plication makes it more difficult to dynamically modify aspects, since it may modify the structure of the application objects. Such flexibility could be provided by imple-menting primitives which capture (serialize) the state of the application's objects and rebuild (de-serialize) a new version of these objects with the new integrated aspects. This is also a perspective to our work.

7 Conclusion and perspectives

Separation of aspects is promoted by several domains, such as component-based environments and AOP languages and runtime supports. Resulting software is easier to build, reuse and adapt. The main motivation of these environments is to provide pro-grammers with the flexibility of integrating orthogonal aspects into their applications. But usually, this flexibility is obtained to the detriment of performance, by incurring a non-negligible overhead on applications. This is mainly due to the implementation of aspects through indirection objects.

In this paper, we followed a complementary approach: we investigated the issue of combining the flexibility of separation of aspects with efficiency. In the proposed approach, rather than implementing aspects using indirection objects or meta-objects, code injection techniques are used in order to efficiently integrate orthogonal aspects into applications' code. Therefore, performance overhead is reduced while maintain-ing flexibility.

We implemented two Java-based prototypes that illustrate orthogonal integration of aspects into Java applications. The prototypes are more precisely implemented upon a Java bytecode engineering tool for transparent injection of aspects at the level of ap-plications' bytecode. We have experimented with two aspects, namely object replica-tion and protection. For each of these aspects, we prototyped and compared an indi-rection-based implementation and an injection-based implementation. Our preliminary performance evaluation showed that the proposed injection-based scheme allows a speedup of 1.8 to 4. The results of our experiments show that it is possible to build efficient, complex and realistic aspects.

Our work to date has focused on evaluating and comparing indirection-based scheme and injection-based scheme for orthogonal integration of replication and pro-tection aspects. We plan to continue the exploration of other aspects, to conduct both complementary performance evaluations that take into account parameters like mem-ory consumption, and to assess the benefits of using injection-based aspect integration in real software applications. Looking forward, our goal is to investigate a generic facility including language support for describing aspects, and enabling automatic optimized integration of aspects.

References

[1] BCEL, 2002. http://bcel.sourceforge.net/

[2] B. Bershad, M. Zekauskas, W. Sawdon, The Midway Distributed Shared Memory System, 38th IEEE Computer Society International Conference (COMPCON'93), February 1993.

[3] W. Binder, J. Hulaas, A. Villazón, R. Vidal. Portable Resource Control in Java: The J-SEAL2 Approach, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2001), October 2001.

[4] K. Bollert. On Weaving Aspects. Aspect-Oriented Programming Workshop at the European Conference for Object-Oriented Programming (ECOOP'99), Lisbon, Portugal, June 1999.

[5] S. Chiba. Javassist - A Reflection-based Programming Wizard for Java, ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java, October 1998.

[6] D. Hagimont, J. Mossière, X. Rousset de Pina, F. Saunier. Hidden Software Capabilities, Sixteenth International Conference on Distributed Computing Systems (ICDCS), May 1996.

[7] D. Hagimont, D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), September 1998.

[8] D. Hagimont, F. Boyer. A Configurable RMI Mechanism for Sharing Distributed Java Objects, IEEE Internet Computing, Volume 5, number 1, January 2001.

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. European Conference for Object-Oriented Programming (ECOOP '97), Jyväskylä, Finland, June 1997.

[10] M. O. Killijian, J. C. Ruiz, J. C. Fabre. Portable Serialization of CORBA Objects: a Reflective Approach. 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2002), Seattle, WA, USA, November 2002.

[11] K. K. S. Lee, Y. H. Chin. A New Replication Strategy for Unforeseeable Disconnection under Agent-Based Mobile Computing System. International Conference on Parallel and Distributed Systems (ICPADS'98), Tainan, Taiwan, December 1998.

[12] T. Lindholm, F. Yellin. The Java Virtual Machine Specification, (Second Edition). Addi-son Wesley, February 1999. http://java.sun.com/docs/books/vmspec/

[13] B. E. Modzelewski, David Cyganski, M. V. Underwood. Interactive-Group Object-Replication Fault Tolerance for CORBA. 3rd USENIX Conference on Object-Oriented Technologies (COOTS), Portland, Oregon, USA, June 1997.

[14] Object Management Group, CORBA Components: Joint Revised Submission, OMG TC Document orbos/99-08, August 1999.

[15] R. Pawlak, L. Duchien, G. Florin. An automatic aspect weaver with a reflective programming language. 2nd International Conference on Meta-Level Architectures and Re-flection (Reflection'99), Saint-Malo, France, June 1999.

[16] A. Popovici, T. Gross, G. Alonso. Dynamic weaving for aspect-oriented programming. 1st Aspect Oriented Software Development (AOSD'02), Enshede, The Netherlands, April 2002.

[17] T. Sakamoto, T. Sekiguchi, A. Yonezawa, Bytecode Transformation for Portable Thread Migration in Java, International Symposium on Mobile Agents (MA'2000), September 2000.

[18] B. Smith Reflection and Semantics in a Procedural Language. Technical Rapport, Laboratory for Computer Science, Massachussets Institute of Technology, 1982.

[19] Sun Microsystems, Enterprise Java Beans Specifications, Version 2.0, 2001.

[20] E. Truyen, B, Robben, B, Vanhaute, T, Coninx, W, Joosen, P, Verbaeten. Portable Sup-port for Transparent Thread Migration in Java. 4th International Symposium on Mobile Agents (MA'2000), Zurich, Switzerland, September 2000.

[21] M. Tatsubori, T. Sasaki, S. Chiba, K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. European Conference on Object-Oriented Programming (ECOOP'2001), Budapest, Hungary, June 2001.

[22] R. Wahbe, S. Lucco, T. Anderson, S. Graham, Efficient Software-Based Fault Isolation, 14th ACM Symposium on Operating System Principles (SOSP'93), pp. 203-216, December 1993.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399