Abstracting Services in a Heterogeneous Environment

Salah Sadou[1], Gautier Koscielny[2], and Hafedh Mili[3]

[1] Valoria Lab., Université de Bretagne Sud, France
Salah.Sadou@univ-ubs.fr
[2] LIFL Lab., U.S.T. Lille, France
Gautier.Koscielny@lifl.fr
[3] Département d'informatique, Université du Québec à Montréal, Canada
hafedh.mili@uqam.ca

Abstract. Applications often use objects that they do not create. In general, these objects belong to an execution environment and are used for some services (*server* objects). This makes applications strongly dependent on these objects and make them vulnerable to any modifications to these objects.
In this paper, we present a solution to this problem through the *service group* concept. A *service group* is an intermediary between the applications and the server objects. A *service group* is defined by the administrator of the shared services, for the entire set of client applications. A *service group* embodies a collection of signatures corresponding to the provided services and maintains the required associations between these signatures and the actual implementations of these services by the server objects. The client applications access the services through *service groups*, and are no longer directly related to servers, thus becoming more independent and better protected from modifications to the server objects. The *service group* will not only make it possible to pool disparate services in order to structure execution environment, but also to construct new services by composing existing ones.

Keywords: Distributed objects, Dynamic reuse, Disparate services, Environment structuring, Service compatibility, Service group.

1 Introduction

When building applications, we need various forms of reuse, i) the reuse of classes by creating objects from them, and ii) the reuse of *existing objects* in the system. An example of the second form of re-use is the printing daemon object that applications use when they need to carry out a printing job. Indeed, an object of this type manages the requests for printing jobs that come from various applications. It is an object that belongs to a common environment, shared by several applications.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 141–159, 2001.

What we call a common environment can be a machine or several machines on a local area network. In such an environment, there can be many such objects that provide shared services, and applications may be more or less tightly coupled to those objects.

We want to address the problem of using the services offered by these objects without being too dependent on the way those services are defined and implemented.

Two significant points arise from the statement of this problem:

Service use. Often we only reuse a part of an already built object. In this kind of situation, we are more interested in a subset of the services that it offers (its methods), than in its actual type (and full type). For instance, it is often the case that two objects, with two different types, offer equivalent services. In such a case of reuse, we must "abstract away" differences in types and focus only on the services that they offer.

Independence. We spoke about an abstraction on servers' types, but it will be necessary to go further: to have an abstraction on individual service signatures. Indeed, two similar services do not always have the same signature and if we want to replace one by the other, the call to the service, in our application, should not depend on what we consider to be non-essential aspects of a server's signature.

So, this problem is related to the evolution of service systems. In this paper, we propose a solution to the problem that makes a distinction between the definitions of services from the point of view of client applications, from the definitions of actual services as provided by actual servers. Naturally, we provide "service managers" with the means to describe *dynamically* the correspondence between these two definitions. This makes it possible to integrate a new server object in the system and to use it immediately thereafter by the existing applications. Further, we introduce the notion of a *service group*, which is a way to logically organize a disparate set of services.

In the next section we discuss related work in order to highlight the motivation of our approach. Section 3 presents the *service group* concept that provides means to define a set of needed services within the context of a particular environment. In section 4, we show the techniques used to establish correspondence between the client view and the server view of a set of services. We will discuss choices for the implementation of a *service group* in section 5, and conclude in section 6 by discussing directions for future research.

2 Related Work and Motivation

Our work touches on four separate research issues, dynamic service (re)use, typing relationships, encapsulation of multiplicity, and behavioral composition. We discuss research related to these four areas in order to highlight the motivation of our approach.

2.1 Dynamic Service Reuse

Typically, the way to implement dynamic service management and invocation involves going through intermediaries such as naming and trading services [23]. A naming service manages a hierarchical tree of name-object reference bindings. Although this approach enables an object to be designated by a unique machine-independent name, clients must know the *name* of the service beforehand. By contrast, a trading service allows clients to access services by function (however it may be described) rather than by name. More precisely, servers publish through the trader service the services that they offer. In turn, clients request services from the trader service by specifying the characteristics of those services. The trader matches the client criteria to the registered servers, and returns to the clients object references for those servers that match their criteria.

While the trading allows clients to be configured without prior knowledge of server objects, it has the following implications:

- *Strict typing requirements*. Service reuse remains complex due to the existence of different types representing the same service. Neither subtyping rules of *CORBA* and *RM-ODP* [21] nor service subtyping rules of the trading service are able to accommodate the variations in signatures (even the functionally equivalent ones) that our approach supports. Moreover, type evolution is only partially addressed in these platforms. Clearly, we need a more flexible approach to service conformance than the one offered by these approaches.
- *Server multiplicity*. Notwithstanding the problem of having servers of different types offer the same service, we also have the problem of several server *instances* that can offer the same service. Having to choose among these adds another complication to the use of these servers. There are two ways of handling this: we could either let the service manager (e.g. trader service) choose one, or let the client application choose among possible offers. The trader service does support an API for the addition, withdrawal, and selection of offers, and we may see an advantage in letting client applications choose among service offers. However, because traders "get out of the way" as soon as a client application gets a reference to a server, it becomes the responsibility of the client to ensure that the server they got is still active. If the server in question cannot be restarted for some reason, then it is the responsibility of the client application to go back to the trader for another server. This is a recurring problem that must be treated directly in the code of client applications, before *each* service invocation. Note finally that there is no transparency of access to the service invocation (access is done in two steps), leading to complexity when writing client code.

The notion of *service group* addresses both problems. First, we use the service conformance relationship (see section 4) to accommodate syntactic (renaming, reordering) and minor semantic differences (more parameters). Second, the *service group* object acts as a permanent interface between clients and servers, shielding clients from server life-cycle issues.

In [27], Singh *et al.* proposed an extension of a trading service, called a *facilitor*, that manipulates interfaces of servers, much like our approach. A *facilitor* reduces the coupling between clients and servers by hiding the number and the specifications of servers to the client. However, their approach relies on a complex agent communication language consisting of a vocabulary, a content language and a communication language.

2.2 Type Relationships

In interoperable systems, the operational interface of an object is formally specified in an interface definition language such as the *CORBA IDL* or the *DCE IDL* [1]. Type conformance in these languages is generally defined by subtyping,*i.e.* an instance of inclusion polymorphism which specifies a substitution rule between types [6]. Roughly speaking, a type τ must replace a type σ in each context where σ is expected. Thus, it allows services of one type to be substituted at run-time by instances of the subtype. The subtype relationship as defined by *CORBA* consists of pure extension, *i.e.* subtypes are obtained by *adding* operation signatures to super-types. With *RM-ODP*, a subtype may *refine* the signature of an operation of the super-type by widening (super-type) the type of an argument and narrowing the type of the result. The subtype relationship is reflexive, antisymmetric and transitive. On the contrary, our definition of type substitutability is a pre-order. It is a reflexive and transitive relation that is weaker than subtyping, *i.e.*, a subtype τ of a type σ is also coercively compatible with σ.

Several research efforts have addressed the problem of type compatibility and introduced other forms of type relationships (equivalence, conversion, service relations). In [4] and [20] a type manager is proposed that permits the addition and deletion of various compatibility relationships between types in a generic type repository. This repository consists of a type repository and a relationship repository. It provides operations for type matching and for various queries, including finding all the types that are compatible with a given type. In our approach, we want each *service group* to act as a repository of both roles and type conformance relationships between roles and service types; this repository is queried when new objects join a server group.

2.3 Encapsulation of Multiplicity

Several models have used the object group concept to make the fact that several servers may offer the same service (multiplicity transparency) transparent to clients [26,16]. Such models often consist of grouping active objects implementing a particular service so as to ensure service availability through replication. The *Gaggles* [2] model hides from clients the number of objects that implement a service. It is designed as a mechanism for invoking one out of a number of equivalent servers, in the same way that our approach does. However, *Gaggles* does not support the run-time definition and addition of services, like our *service groups* do.

Platforms such as *ANSA*[24], *GARF*[9] and *Electra*[17] propose stronger object group protocols than *Gaggles*. The *GARF* platform, for example, relies on the *Isis* [12] group communication toolkit, which provides various multicast primitives to ensure consistency between replicas in spite of failures and concurrency. Object groups are generally homogeneous, which means that all objects within a group are of the same type. The *Electra* model, like *DCOM* [19], enables the aggregation of different types of objects using a high-level interface. However, invoking a method specified in the group interface amounts to multicasting the invocation to *all* the members that implement the method. By contrast, our *service groups* provide flexible invocation mechanisms, supporting several types of protocols (one of, all of, the first one that terminates). Further, *service groups* support the behavioral composition of the available services.

2.4 Behavioral Composition

Behavioral composition refers to the composition of methods of different interacting objects. Helm *et al.* note in [11] that "patterns of communication within a behavioral composition represent a reusable domain protocol or programming paradigm" and propose *contracts* as constructs for the explicit specification of behavioral relationships between objects. A contract defines a set of communicating participants and their contractual obligations. It provides two operations for expressing a complex behavior in terms of simpler behaviors, refinement and inclusion. Contractual obligations consist of type obligations and causal obligations. In our approach, we consider only type conformance obligations to join a *service group*. However, contrary to contracts, *service groups* allows relaxed type conformance and accept the membership of different type specifications for a given role.

3 Service Group

The objectives of our approach may be summarized in four keywords: adaptation, pooling, transparent access, and service composition:

adaptation. It refers to the ability of applications to have their own expectations of services, and to match those expectations with the definitions of the actual services;

pooling. Refers to the convenience of having a single point of service for a set of related services, instead of having to bind to each server individually;

transparent access. To access a service, the client does not need to know the actual server that provides it. This enables us to interchange servers that provide a given service, without affecting the clients. Transparent access is made possible in part thanks to adaptation and pooling;

service composition. It refers to our ability to compose the services offered by actual servers to build new and more complex services. The composition of services involves the collaboration of existing servers, which is orchestrated by a *service group*.

The *service group* concept is useful to the extent that:

1. the needed services at hand are provided by generic objects that are not specific to a particular set of applications;
2. the same service may be offered by different objects, possibly of different type, or even with different invocation signatures;
3. to make the effort worthwhile, we assume that *several* applications share the same services.

3.1 Principles

Consider a situation where we have, on the one hand, a set of independently defined active objects that play the role of *servers*, and on the other, a set of client applications that have the same definitions (expectations) of the services they need. In other words, in the execution environment, there is only one definition for any given service, for the entire set of client applications. In order to shield client applications from an evolution in the way those services are implemented, we introduce a bridge between them in the form of a *service group*. For conceptual cohesion and tractability, we further assume that *service groups* pool services that are logically related such as printing services, database querying services, etc. for all of the client applications.

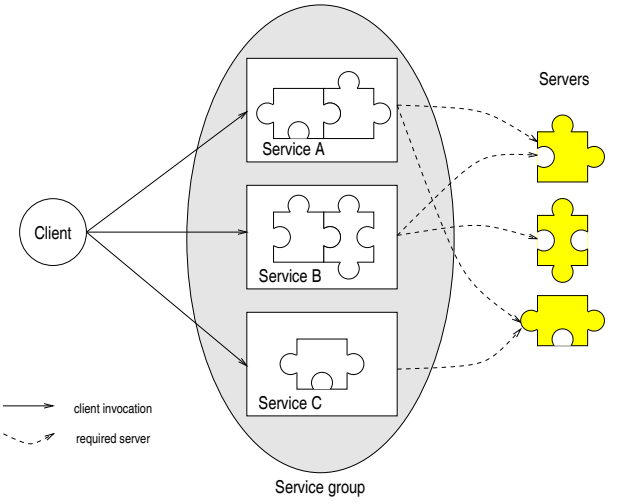

Fig. 1. Interaction of clients with a Service Group.

A *service group* is an intermediary between the applications (clients of the service) and the server objects (implementers of the service), see figure 1. A

service group is an active object defined by the administrator of the shared services, for the entire set of client applications. A *service group* embodies a collection of signatures corresponding to the provided services and maintains the required associations between these signatures and the actual implementations of these services by the server objects. Viewing a *service group* as an object, the various services it manages should not be viewed as *methods* of that object, in the OOP sense; in non-prototype OOP languages, the behavior of an object is specified at object creation time, and doesn't change after that. Services are better viewed as data held by a *service group*, making possible to add and remove services to a *service group* after it has been "created". A service invocation then is just a way to specify one item of that data; it's the responsibility of the *service group* object to translate a service invocation into actual method calls on actual server objects. In other words, a *service group* is a meta-level (reflective) object [18] centralizing the definitions and invocations of the behaviors of several other objects.

As suggested by figure 1, the services offered by a *service group* may either be implemented by a single service of server objects, or as a composition of services provided by several server objects.

3.2 Defining Service Groups

Simply stated, the definition of a *service group* consists of, 1) a description of the services provided to the clients, and 2) a description of the services required (needs) of the servers to provide them.

provided services a collection of signatures of services related to the same application domain;

needs a set of basic services/functionalities required from servers, to be able to provide **services**,

The needs are defined by method signatures and gathered in subsets representing *roles* to be played by members of the group (see figure 2). Servers join a *service group* to play a particular role. We distinguish between *role* and *role instance*. A role[1] is a generic service type that represents a set of object types providing similar services, whereas a role instance represents a binding to a particular server. A role is described by a name and a set of method signatures that define its behavior. It is instantiated at run-time by servers whose methods conform to those defined in the role (see section 4). Several instances of the same role can exist simultaneously, and independently. This means that the group can send the same request to any or several servers. Similarly, a server joining a *service group* can play (conform to) several roles. Thus, servers are dynamically bound to roles.

[1] A role may be considered as an object template and even if our model let server objects choose their roles, our approach is quite different from the role abstraction described in[15].

Further, note that an existing group may be bound to a role defined in another group, since it is also an object with a well-defined interface. A service group may represent a part-whole hierarchy[3] whose behavior may be composed in terms of other groups. Hence, roles may refer to service subgroups.

```
Group PrinterSG {
   role Printer {
     string printPs(Source doc, boolean twoSide);
   }
   role Translator {
     Source dviToPs(Source dviFile);
     Source psToRtf(Source psFile);
   }

   private Source dviToPs(Source dviFile) {
     return Translator.dviToPs(dviFile);
   }

   public string printDVI(in Source dvi,
                          in boolean twoSide) {
     Source ps = dviToPs(dvi);
     return printPs(ps, twoSide);
   }

   public string print(Source ps, boolean twoSide) {
     return Printer.printPs(ps, twoSide);
     }
}
```

Fig. 2. A printing service group definition.

A service, as shown in figure 2, may be a behavioral composition of *role methods*. A service may be declared either **public** or **private**. Only the public services of a role may be invoked by clients; private services are provided as a way of constructing more complex services to be offered by the *service group*. An example of a simple coordination is given by the *printDVI* service of figure2, which prints a document given its *dvi* format. A composite public service will not be available to the *service group* clients if one of the roles involved in the composition has not been instantiated (bound to an actual server).

New services may be supplied at run-time and added to the interface of a *service group* (see section 5). The *service group* shown in figure 2 is specified using our own extension to the *Java* language called *JGroup*. This extension isn't a new language. It's just a means to simplify the use of *service group* framework.

3.3 Service Composition

By gathering services related to the same application domain, a *service group* may offer new services by composing existing ones. This composition can be either sequential or parallel.

Sequential Composition. In the previous example, we defined a service called printDVI that prints *dvi* formatted documents directly. Based on the available servers, we know that we cannot print such documents directly: we have to transform them into a *postscript* format. Accordingly, we defined a role called *Translator* to reflect the fact that we need servers that can translate a *dvi* document into a *postscript* one, and defined the printDVI in terms of that role: to print a *dvi* document, the *service group* has to, first, translate it to *postscript* using the *Translator* role, and then, print the *postscript* document using the *Printer* role. The sequentiality of these operations is represented using the ";" operator. In the sequential composition, the invocation of a service on a role is synchronous: the first call blocks the composed service and must wait until the end of the first invocation to continue the computation.

Parallel Composition. We propose a mechanism of asynchronous invocation through the use of a parallel composition operator '||'. The expression of parallelism is thus explicit but the collection of results remains implicit.

The parallel composition combinator '||' acts as a separator between instructions (the same way the ';' sequential composition operator does). Consider, for example, the dviToRtf service defined in figure 3. This service consists (in part) of two invocations: an invocation on the translator t1 followed by an invocation on the translator t2. These two invocations are independent, and thus, can be executed in parallel. The results are collected later and the variables ps1 and ps2 represent implicit futures [7]. So, the execution of this service, will continue until it needs the values of these variables.

Because several objects can play the same role, an invocation can be multicast to several role instances. In this way, we provide a simple fault tolerance mechanism through replication [10]. For example, the new definition of the dviToPs service that is shown in figure 3 consists of invoking, asynchronously, two translator instances to perform the translation. The first result to be received is the one returned by the *service group*. The operation is performed even if only one server is bound to the translator role. In that case, the second part of the instruction is ignored.

Note, finally, that one can broadcast the invocation of a particular service to *all* of the servers that play that role. The psToRtf service illustrates this: it interacts with all the translators and returns the first available answer. If one needs several answers, servers may be called consecutively and asynchronously, as in the dviToRtf service definition.

```
Group PrinterSG {
    ...
   public bytes dviToRtf(in bytes dvi) {
     Translator t1, t2;
     bytes ps1, ps2;
     ps1 = t1.dviToPs(dvi) || ps2 = t2.dviToPs(dvi);
     ...
   }

   public bytes dviToPs(in bytes dvi) {
     Translator t1, t2;
     bytes ps = t1.dviToPs(dvi) || t2.dviToPs(dvi);
     return ps;
     }

   public bytes psToRtf(in bytes ps) {
     bytes ps = Translator||.psToRtf(ps);
     return ps;
   }
    ...
}
```

Fig. 3. Various parallel service compositions.

4 Conformance of Services

In order to play one role of a *service group*, a server must conform to the specification of that role, i.e. the set of method signatures that are specified in it. This means that for each method signature listed in the role, the server has (at least) one method that conforms to that signature. This section deals with the meaning of "conformance" in this context.

Thereafter, the concept of *type of service* refers to a particular invocation (method) signature, and to the expected behavior of the service. A strict interpretation of conformance requires that implementers of the service support a *method* that has the exact same signature and that produces the same behavior. In our case, we use a broader interpretation of conformance that makes it possible for a service offered by a group to be carried out by servers whose methods may have a different signature from that which is advertised by the server group. The type of a service, as defined by a role, may be regarded as a generic type of service [14]; it represents a *class* of services in the sense that they may have different types, provided that those differences are not 'essential'.

We consider the generic service type as the canonical representative of the set of types that are compatible in the sense of contravariance and *coercive compatibility*, defined further below. This enables us to use a service invocation protocol that factors out the commonalities between the interfaces of the various servers.

To illustrate this, we elaborate the printing service example. We assume that the users of a local area network (LAN) have several printers with *different* characteristics; some accept only simple postscript printing while others allow color and/or double-side printing. Figure 4 shows the interface definitions for the various printers.

```
interface Printer1 {
 ...
 void printPs1(Source doc, String status);
}

interface Printer2 {
 ...
 String printPs2(Source doc, boolean rv);
}

interface Printer3 {
 ...
 String printPs3(Source doc, boolean rv,
                 boolean color);
}
```

Fig. 4. Interfaces of several printing service servers.

The LAN manager wishes to unify the access to these various printers through a single definition of the printing service so that all the applications can use this unique definition of the printing service, while benefiting from all available printers. A solution is shown in the example of figure 2.

Before showing how this manager could define the links between his own definition of the printing service and those provided by the existing servers, we give some formal definitions of the relations that must exist between the two.

4.1 Some Formal Definitions

The behavior of a role is described, syntactically, by an interface that represents a set of generic service types. Several objects may conform to this definition while being completely unrelated in the type hierarchy. Relating a type to a role consists of matching a subset of its methods/services to the services defined in the role. This is done by applying a *coercive compatibility* relationship. The coercive compatibility relationship consists of a signature matching and a conformance to the role at hand.

Service Signature Matching. We must be able to define a matching procedure for each operation that is supposed to conform to the generic service

definition. Signature matching establishes a structural correspondance between the pair $\langle name, parameters \rangle$ of the operation and the pair $\langle name, parameters \rangle$ of the generic service. When the two operations have the same number of parameters, the correspondence is straightforward and only the order and the types of the parameters matter. Zaremski *et al.* in [28] proposed various flavors of relaxed match for functions defined in *ML*, including a reorder transformation that defines a permutation applied to a tuple of parameters. Using this transformation, we can reorder generic service type parameters to match operations parameters.

However, when the number of parameters is different, we must narrow the set of all appropriate inputs to the generic service type. For instance, the generic service type "*string print(Source doc, boolean twoSide)*" of figure 2, allows clients to print postscript documents in gray-scale and possibly in double-side while *printPs1* allows one side printing and *printPs3* allows color printing. Hence, *printPs3* matches *printPs* if its parameter *color* is bound to $false$ and *printPs1* matches *printPs* only for one-side printing.

Definition 1 (service signature matching).
Let $f : A \to B$ *and* $g : A' \to B'$ *be two services with:*
$A = \tau_1 \times \ldots \times \tau_n$, $B = \sigma_1 \times \ldots \times \sigma_k$,
$A' = \tau'_1 \times \ldots \times \tau'_m$, $B' = \sigma'_1 \times \ldots \times \sigma'_k$
and n, m, $k \geq 0$.
g matches f if:
(i) there exists a composition $\mathcal{T} \circ \mathcal{C} : A \to A_{\mathcal{T} \circ \mathcal{C}}$
of a coercion $\mathcal{C} : A \to A_{\mathcal{C}}$
and a reorder transformation $\mathcal{T} : A_{\mathcal{C}} \to A_{\mathcal{T} \circ \mathcal{C}}$
such that $A_{\mathcal{T} \circ \mathcal{C}} <: A'$, *and;*
(ii) $B' <: B$.

Coercion $\mathcal{C}$ is a structural transformation of the parameters of one operation f onto the parameters of another operation g. $\mathcal{C}$ is an injection onto a sub-domain $A_{\mathcal{C}}$ of the domain of g when f has less parameters than g, a projection onto another sub-domain $A_{\mathcal{C}}$ of the domain of g in the opposite case. In both cases $\mathcal{C}$ is a partial function that is not defined for some inputs belonging to the domain of f or g. In order for g to replace f, it must accept arguments of type greater than A, hence the contravariance rule on domains (condition (i)). Inversely, the result of the application of f may be passed as a parameter of another method. Thus, the result of the application of g must be able to be passed as a parameter to the same method hence the covariant rule on codomains (condition (ii)).

Conformance Relationship. Signature matching may be extended to a type, allowing the definition of a conformance relationship between a type and a role. It is a coerced behavior compatibility relationship similar to the one defined in *RM-ODP*[13]. A type τ conforms to a role ρ applying a coercion of the behavior of τ onto the behavior of ρ.

Definition 2 (role conformance).
Let τ *be a type and* ρ *be a role type. let* F_τ *and* F_ρ *be respectively, a subset of* τ *operations and a subset of* ρ *operations.*

τ conforms to ρ if for each operation $g_\tau \in F_\tau$, there exists one and only one $f_\rho \in F_\rho$ such that f_τ match g_ρ.

This definition may enable only a partial compatibility between a type and a role since matching may be applied to subsets of their interfaces.

4.2 Conformance Specification

A conformance specification describes how a type supports a role in a *service group* according to definitions 1 and 2. It is the designer of the *service group* who creates typing obligations within a *service group* in order to flesh out roles.

```
Printer1 conforms to PrinterSG::Printer {
   string printPs(Source doc, boolean twoSide)
     coerces {
       string printPs1(Source doc, String status);
      return status;       }
     where {
      twoSide in [FALSE];
     }
};

Printer2 conforms to PrinterSG::Printer {
   string printPs(Source doc, boolean twoSide)
     coerces {
       string printPs2(Source doc, boolean twoSide);
     }
};

Printer3 conforms to PrinterSG::Printer {
   string printPs(Source doc, boolean twoSide)
     coerces {
      string printPs3(Source doc, boolean color,
                                  boolean twoSide);
     }
     where {
      color in [FALSE];
     }
}
```

Fig. 5. Example of a conformance declaration.

Figure 5 shows different conformance specifications for the *Printer* role of the printer *service group PrinterSG* (see figure 2). Conformance specifications are also defined in *JGroup*. Here we show how the three printing servers of figure 4 match the generic service *printPs*. In one case there is only a name mismatch;

in the other two, we also have parameter mismatch. For example, *printPs3* is compatible with *printPs* provided that the parameter *color* is set to false. To describe the link between a generic service parameter and the corresponding one from the server-provided service, we use the same parameter name in both signatures. If the server-provided service does not return a value (or the *desired* value), we use the "return" key word to return the desired value. That is the case of the *printPs1* service of the *Printer1* server, which uses a *status* parameter to indicate the status of the printing, but which doesn't return it explicitly; the coercion relationship forces the return of the *status* and ignores the actual return value of the service.

4.3 Conformance in Action

Once a conformance relationship is defined between a server type and a role, and once the definition is added to a *service group*, instances of the server type are authorized to join the *service group* in order to play that role. When a server joins a *service group*, its interface is inspected and compared to the type obligations that are stored in the group. Consequently, we may find zero, one or many role specifications that match its interface. In the case where the type of the server does not match any role specification, the server cannot join the group. Otherwise, for each role specification that the type of the server matches, the corresponding role is instantiated. A composite service will remain unavailable if one of its component roles has not been instantiated.

Specifications of conformance relationships are defined separately from the *service group* and may be provided at run-time. This approach allows the definition of new role conformance relationships when finding out about new services that are likely to match a role. The *service group* manager may also withdraw role definitions to reflect the disappearance of services or the creation of a new version of a service.

5 Implementation Choices

The reader may think that we can use *EJB app servers* [22] to implement the *service group* in order to manage resources. But, in *EJB* the pooling uses objects of the same type while *service group* allows the pooling of objects from different types.

Our main concern in implementing the *service group* concept was ease of use. As mentioned earlier, *service groups* are described by Java-like constructs illustrated in figure 2. We have developed a processor that compiles such definitions and generates Java classes. The *service group* itself is represented by a class. The behavior of a *service group* is implemented by *class* (static) methods, for several reasons, 1) to centralize (and share) the management of the various services, 2) to obviate the need for programmers to create/instantiate *service groups*, and 3) to make them aware of the fact that they are accessing a *shared* service that they don't own.

We implemented the *service group* as distributed objects to meet the needs for distributed applications. If the application is centralized, a simple proxy would be enough.

5.1 Using a Service Group

The example of figure 6 illustrates an invocation of one service of the *PrinterSG service group*. It consists of building an array of the service arguments and then, sending the request to the group through its *invokeService* class method; *invokeService* takes the name of the service and its argument list as parameters.

```
import  SG.printer.PrinterSG;
public class User{
 Source documentPs ;
 public String toPrint (boolean doubleSide) {
  String status;
  try {
    Object[] args = new  Object[2];
    args[0] = documentPs;
    args[1] = new Boolean(doubleSide);
    status =
    (String)PrinterSG.invokeService("printPS", args);
    } catch (Exception e) {
        System.err.println("printPS service error");
        }
  return status ;
  }
}
```

Fig. 6. Using a service group

For each *service group* described in the *JGroup* syntax, we generate a class by using a template that consists of the definition of the class methods representing the possible actions on the group. These methods are of two kinds:

public for methods destined to the clients/users of the *service group*. These methods include the invocation of services (as shown in figure 6), and querying actions to know about the existing services or roles, to check whether a service is active or not, etc.

protected for methods destined to the *service groups* manager. These methods manage services, roles, support the addition of new conformance relationships, and handle the registration (joining) of new servers.

5.2 Service, Role, and Server Conformance Specifications

As mentioned earlier, the specification of conformance relationships between servers (server types) and roles is provided separately, and is treated as data that *service groups* can add and remove at will during run-time. In the implementation, these specifications are provided in the syntax illustrated in figure 5, in a separate file. In fact, this is true of all the components of a *service group*: each specification, be it of a service, a role or a conformance relationship, is stored in a separate file, and compiled to produce a class. To add any of the three types of components to the *service group*, the group manager provide the *service group* with the class name so that it loads the corresponding class, creates an instance of it, an adds it to the appropriate part. To this end, the group manager has various external commands to manage existing groups (see figure 7. All these commands use the protected services of service groups.

---

addRole *groupName roleClassName*
removeRole *groupName roleClassName*
addService *groupName serviceName*
removeService *groupName serviceName*
addConformance *groupName conformanceName*
removeConformance *groupName conformanceName*
addServer *groupName serverRMIAdress*
removeServer *groupName serverRMIAdress*

Fig. 7. External commands for service group managing.

Note that the services and roles that are specified as part of the initial group definition (as opposed to defined separately) will also have classes generated for them. The difference with those that are defined externally (and added dynamically) is that they are created automatically as part of the construction (instantiation) of the group.

This approach has the advantage of making *service groups* sufficiently flexible to change their behavior during run-time, without requiring a shut-down or an interruption of service.

6 Conclusion and Future Work

To conclude this paper, we will point out the contributions of our work, its limitations, and discuss directions for future research.

6.1 Contributions

The contributions of our work do not reside in the proposal of a mediator or a facade pattern, which are well-known design patterns [8] but rather in tackling different issues, some of which are orthogonal:

Execution environment structuring. We argue that the proper structuring of the execution environment of applications is as important as the structuring of the applications themselves. A well structured environment simplifies the writing of applications. *Service groups* help in this structuring by providing a simple organization of disparate services. It unifies/factors the definitions of compatible services in order to simplify their use. The unification of the definition of a shared service for the purposes of several applications, creates a certain coherence between these applications. Because the unified definition does not depend directly on the existing server objects, the environment can evolve (change in the type or the implementation) without affecting the applications that use it.

Dynamic (re)use of services. By gathering services that are similar in function but different in protocol, we can, by the same token, hide the multiplicity of the *existing* implementations of the service (with possibly different types), and make it possible to add *new* implementations (also with different types) and let client programs take advantage of those implementations, even if they don't know their type. Our implementation goes one step further by supporting the addition of new *services*, enabling us to evolve the *service group* without shutting it down.

Composition of services. Through the JGroup extension to Java, we provide a simple mechanism for composing existing services in order to provide new ones. Thus, client applications can use services that no server alone could provide.

Our work on integrating disparate "infrastructure" services (printing, communications, database, etc) can be applied to *domain* services or objects that may be shared by several applications. For example, a set of financial applications could require the same wire-services to access currency exchange rates or live stock quotes. In this case, the *service group* concept can be used to implement pools of "read-only" objects of different types to accommodate varying application loads. The applications become like views on this environment. This helps ensure a cohesion and stability of the core of these applications.

We experimented with this approach in our own computer science department. Students, professors, administrators and courses are active objects connected to *service groups* shared by various applications. For example, we can have a professor objects that is connected, at the same time, to the professor group and to the administrator group, to reflect the fact that it may be used as both a professor and an administrator. This work was made within the "Group" project [25] that was financed by the regional council.

6.2 Limits

Our definition of a generic service type relies only on signatures but does not handle behavioral specifications [5]; conformance relationships can only guarantee type safety, but do not guarantee the behavioral conformance of the servers to the expected service. For the time being, behavioral conformance is ensured

by the vigilance of the service manager who should only define conformance relationships for those services that she/he knows do implement the actual service!

6.3 Future Work

Currently, we are working on the behavioral conformance problem. We are exploring the possibility of specifying the behavior of a service through functional and nonfunctional assertions. This will be used for both actual server types and for the definition of generic services. Some sort of theorem proving procedure may then be used to validate the behavioral conformance of a server type to a defined service, enabling us to automate all of the functions of a service manager. We are currently developing a pragmatic formal specification language that includes both linguistic constructs (such as the ones used by trader services) as well as formal ones.

References

1. P. A. Bernstein. An architecture for distributed system services. Technical Report CRL 93/6, Digital Equipment Corporation, Cambridge Research Lab, March 1993.
2. A. P. Black and M. P. Immel. Encapsulating Plurality. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 57–79, Kaiserslautern, Germany, July 1993. Springer-Verlag.
3. E. Blake and S. Cook. On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. In *ECOOP'87*, pages 41–50, 1987.
4. W. Brookes and J. Indulska. A type management system for open distributed processing. Technical Report 285, Department of Computer Science, Queensland U., Brisbane QLD (Australia), February 1994.
5. D. Buchs and N. Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, 26(7):635–652, 2000.
6. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
7. D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
9. R. Guerraoui, B. Garbinato, and K. Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4):32–39, 1997.
10. R. Guerraoui and S. André. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.
11. R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In N. Meyrowitz, editor, *Conference on Object-Oriented Programming, Systems, Languages, and Applications/European Conference on Object Oriented Programming (OOPSLA/ECOOP'90)*, ACM SIGPLAN Notices, pages 169–180, October 1990.
12. IONA and Isis. An Introduction to Orbix+Isis. Technical report, IONA Technologies Ltd, November 1995.

13. ITU/ISO. Reference model of open distributed processing - part 2 : Foundations, 1995. ISO/IEC 10746-2, ITU-T Rec. X.902.
14. G. Koscielny and S. Sadou. Type de service générique pour la réutilisation de composants. In Jacques Malenfant and Roger Rousseau, editors, *langages et modèles à objets (LMO'99)*, pages 115–130. Hermès Science Publications, 1999.
15. B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Sytems*, 2(3):143–160, 1996.
16. D. Lea. Objects in groups. Technical report, SUNY Oswego, 1993.
17. S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, February 1995.
18. J. McAffer. Meta-Level Programming with CodA. In W. Olthoff, editor, *European Conference on Object Oriented Programming (ECOOP'95)*, Lecture Notes in Computer Science, pages 190–214. Springer-Verlag, aug 1995.
19. Microsoft. Dcom.
http://www.microsoft.com/cominfo/.
20. M. Muenke, W. Lamersdorf, B. O. Christiansen, and K. Mueller-Jones. Type management: A key to software reuse in open distributed systems. In *EDOC'97*, Gold Coast, AUSTRALIA, 1997.
21. ODP. Open Distributed Processing.
http://info.gte.com/ftp/doc/activities/x3h7/by_model/ODP.html.
22. R. Monson-Haefel. Enterprise JavaBeans, 2nd Edition. March 2000, O'Reilly & Associates, INC.
23. OMG. *CORBAservices : Common Object Services Specification*, chapter Trading Object Service Specification. Object Management Group, Inc. Publications, March 1997. 97-12-23.
24. E. Oskiewicz and N. Edwards. A model for interface groups. Technical Report APM.1002.01, ANSA, Architecture Projects Management Limited, Cambridge, UK, May 1994.
25. S. Sadou, G. Koscielny, P. Frison, and J-M. Inglebert. Groupes pour la coopération entre activités. Technical report, Valoria/Orcade, Université de Bretagne Sud, December 1999.
http://www.iu-vannes.fr/ sadou/groop/rapport/rapport.html.
26. K. Shimizu, M. Maekawa, and J. Hamano. Hierarchical Object Groups in distributed Operating Systems. In IEEE Computer Society and Technical Committee on Distributed Processing, editors, *The 14th International Conference on Distributed Computing Systems*, pages 18–24, San Jose, California, June 1988. IEEE, Computer Society Press.
27. N. Singh and M. A. Gisi. Coordinating distributed objects with declarative interfaces. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models (Coordination'96)*, number 1061 in Lecture Notes in Computer Science, pages 368–385. Springer, 1996.
28. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.