Experiences with OpenORB's Compositional Meta-Model and Groups of Components

Katia B. Saikoski* and Geoff Coulson
Distributed Multimedia Research Group,
Department of Computing, Lancaster University,
{saikoski,geoff}@comp.lancs.ac.uk

1 Introduction

The *group* abstraction is a crucial element in the support provided by middleware platforms. Application areas such as CSCW, dependable systems or media dissemination can take advantage of groups to facilitate the communication and management of the several end-points participating in the application. Although there have been proposals for the provision of groups in CORBA and other middleware platforms (e.g., Electra [7], Eternal [9], OGS [6], JGroup [8]), there remains an unsatisfied demand for a more comprehensive group support in order to address the following requirements:

(a) Groups should equally accommodate the requirements of the variety of emerging distributed applications. For example, an application that manages the dissemination of audio over the Internet is different in nature from an application that controls a set of replicated servers. While the former involves the distribution of audio from one producer to a number of receivers, the latter manages the transparent access of clients to a set of servers. Additionally, the requirements associated to the communication (reliability, ordering, etc) also vary.

(b) Groups should also address the dynamic requirements of distributed applications by providing run-time adaptation. Changes in application requirements occur because the environment changes (e.g., a mobile user facing network problems) or because the application needs to evolve (e.g., a software module needs to be updated). Because a number of applications cannot be stopped either because they are essential applications (e.g., traffic control systems, life support systems) or because the changes occur very frequently (e.g., users connecting and disconnecting from the Internet, network fluctuations), it is essential that run-time adaptation be provided.

We address these issues in the context of the OpenORB project [3], the aim of which is to provide a highly flexible middleware architecture based on reflective component technology. Component technology brings the possibility of creating particular instances of the middleware tailored for specific purposes, while reflection provides a suitable mechanism to uniformly handle adaptation.

In this paper, we present the experiments in the use of the OpenORB reflective middleware to address groups of components. The Group support for OpenORB (GOORB) platform is an OpenORB instance targeted at the development of group services. Its main objective is to provide a flexible infrastructure for the creation of group services and groups that can be tailored to meet application needs. At design and start up time, flexibility is provided through component configuration, which allows group services and groups to be built by assembling software components shaped to a specific use. At run-time, flexibility is provided by means of dynamic adaptation, which enables group services and groups to adjust themselves to new requirements.

2 The OpenORB basics

As mentioned above, OpenORB [2, 5] is built according to a component-based architecture. At load-time, components are selected and appropriately composed to create specific instances of the middleware. In addition, components can be loaded into capsules at run-time. Reflection is used to facilitate the change and reconfiguration of the set of components in a capsule at run-time, and thus dynamically adapt the middleware functionality.

OpenORB incorporates the following characteristics. Component interfaces are specified in (an extended version of) the CORBA Interface Definition Language (IDL), and components may export any number of interface types. RM-ODP compatible *signal* and *stream* interaction types are supported for events and continuous media. As well, each interface takes

*PhD Student sponsored by CAPES and PUCRS, Brazil.

one of two possible *roles*: *provided* (services the component offers) or *required* (services the component requires).

Communication between the interfaces of different components can only take place if the interfaces have been either explicitly or implicitly *bound*. In terms of role, required interfaces can only be bound to provided interfaces and vice versa. To-be-bound interfaces must also match in terms of their interaction types (i.e. method signatures etc.). Crucially, bindings between interfaces are themselves components. There are two categories of binding component: *local bindings* and *distributed bindings*. The former, which are simple and primitive in nature, are used only where the to-be-bound interfaces reside in the same address space (capsule). Distributed bindings are themselves *composite* and *distributed* components which may span capsule or machine boundaries. Internally, these bindings are composed of sub-components (bound by means of local bindings or, recursively, by lower-level distributed bindings) that represent various aspects of the communications system. Examples are primitive transport level connections (e.g., a "multicast IP binding"), media filters, stubs, skeletons etc. Distributed bindings are often constructed in a hierarchical (nested) manner; for example a "video binding" may be created by encapsulating a configuration consisting of a "primitive" multicast IP binding augmented with H.263 filter components.

A representation of these concepts can be seen in Figure 1.

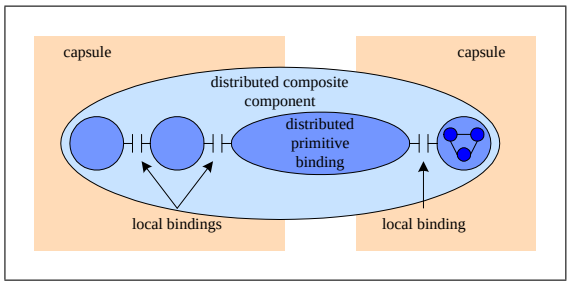

Figure 1: The OpenORB Basics.

In terms of reflection, every OpenORB component has an associated *meta-space*, which is accessible from any of the component's interfaces, and which provides reflective access to, and control of, the component in various ways. To help separate concerns, meta-space is partitioned into various orthogonal *meta-models*. In this paper, we focus exclusively on the *compositional* meta-model which gives access to the internal representation of composite components.

3 GOORB's Compositional Meta-Model

The structure of a composite component in OpenORB is represented by the compositional meta-model, which takes the form of a *component graph* data structure. This serves as a causally connected self-representation of a composite component's internal structure, and therefore, manipulations of the component graph's topology result in corresponding changes in the component's composition.

The meta-object protocol (MOP) for the compositional meta-model was initially proposed in [4], then refined in [5] and finally restructured in [3]. In this paper, we discuss an implementation of the compositional meta-model based on the MOPs mentioned and its use for manipulating groups of distributed components.

GOORB's *ICompositionalMOP* interface provides methods for manipulating a composite component through its meta-object. Figure 2 presents part of the IDL specification for that interface.

Listing 1: IDL for the *ICompositionalMOP* interface.

```
// CORBA IDL
interface ICompositionalMOP {
  ComponentGraph getGraph( );
  CompList getComponents( );
  EdgeList getLocalBinds( );
  IfaceList getExternal( );
  void addIfaceExternal( out IfaceName extiface,
                         in CompID comp,
                         in IfaceName iface )
     raises ( CompNotInGraph, IfaceNotInComp );
  void delIfaceExternal( out IfaceID )
     raises ( IfaceNotInComp );
  void addComponent( in NewComp newcomp,
                     in Position position,
                     in IfaceList extifaces )
     raises ( InvalidPosition, DanglingInterface );
  void delComponent( in CompID delcomp,
                     in EdgeList mapping )
     raises ( CompNotInGraph, InconsistentMapping );
  void replaceComponent( in CompID oldcomp,
                         in NewComp newcomp )
     raises ( CompNotInGraph, CompsNotCompatible );
  void localBind( in IfaceID iface1,
                  in IfaceID iface2 )
     raises ( NotBindable );
  void breakLocalBind( in IfaceID iface1,
                       in IfaceID iface2 )
     raises ( LocalBindNotInComp );
};
```

As can be seen in Listing 1, the ICompositionalMOP interface provides a number of methods for inspecting and performing adaptation on a composite object. In more details, the methods for inspection are *getGraph()*, *getComponents()*, *getLocalBinds()* and *getExternal()* and the methods for adaptation are *addComponent()*, *delComponent()*, *replaceComponent()*, *addIfaceExternal()*, *delIfaceExternal()*, *breakLocalBind()* and *localBind()*. The methods for inspecting a composite component return the current configuration of that component. The *getGraph()* method returns the represen-

tation of the graph that describes the structure of a composite component. The *getComponents()* method returns a list of component identifiers for the components that compose a composite component. The *getLocalBindings()* method returns a list of local bindings in a composite component in the form of a pair interface identifiers. Finally, the *getExternal()* method returns a list of interface identifiers for the interfaces external to the composite component. The methods for adapting a composite component allow the insertion, deletion and replacement of components in the graph. Besides, local bindings between components and external interfaces can also be destroyed or created.

Using the *ICompositionalMOP* interface, adaptation can be performed in several ways. For example, the *addComponent()* method can be used to add a component and unbind/bind the necessary interfaces (using the *position* and *extifaces* parameters) or it can be used in conjunction with *breakLocalBind()* and *localBind()* methods to explicitly control the rearrangement of the graph. In this case, the *position* and *extifaces* parameters are left empty.

4 A Model for Distributed Reconfiguration

In order to perform adaptation in a distributed manner, we propose the following infrastructure. Each OpenORB address space (capsule) has an associated runtime environment to support adaptation. This is composed of an `Adaptation Manager` component and an `Adaptation Data` component. Besides, remote requests for adaptation are issued by `Remote Adaptor` components, which act as surrogates for the `Adaptation Manager`. Figure 2 illustrate these components and how they interact.

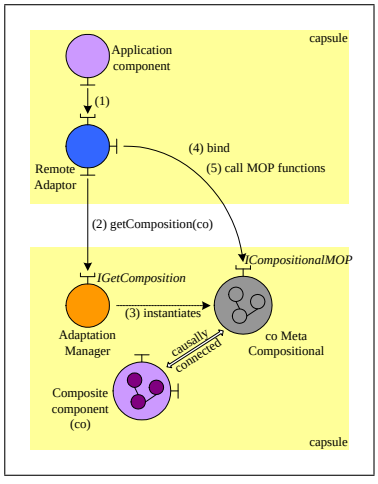

Figure 2: Reconfiguration Model.

In the figure, an application component requires access to the compositional meta-model of component `co` via the `Remote Adaptor` (1), which is the local proxy for performing remote adaptation. The remote adaptor issues a request to the `Adaptation Manager` installed in the `co`'s capsule to access the `co` composition meta-object (2). The `Adaptation Manager` requests the `Local Factory` (not shown in the figure) to create the `co`'s meta-object (3). It returns the reference to the *ICompositionalMOP* interface to which the remote adaptor can bind its required *ICompositionalMOP* interface (4) and call methods to inspect and adapt the remote component `co` (5).

The `Adaptation Manager` component, which is instantiated when capsule is created, gives local support for adaptation. It receives requests from remote capsules to retrieve information about the components in that capsule. The `Adaptation Manager` component implements the *IGetCompositional* whose IDL specification is shown in listing 2. This component can be customised to include access control and authentication to make adaptation safer.

Listing 2: IDL for the *IGetCompositional* interface.

```
// CORBA IDL
interface IGetCompositional {
  typedef long IfaceID;
  typedef long ComponentID;
  IfaceID getComposition ( in ComponentID comp )
    raises ( InvalidComponent );
};
```

The `Adaptation Data` component is a passive component that holds the set of adaptation rules of a particular composite component.

5 Group of objects as a composite component

In order to ease the control over a set of components, GOORB provides support for groups in terms of configuration and adaptation of groups. At the component-level, a group is a distributed composite component, which includes members and the explicit communication between these members. In addition, components to control the functioning of the group such as failure detectors or membership can also be part of the composite (group) component.

Groups are created by group factories based on templates that describe the behaviour and structure of the group (read [10] for more details). When a group is created, a default membership component with methods to join and leave the group is created (*IGroup* interface). Requests to join and leave a group are issued by an application component or a system component in the *IGroup* interface. Then, these requests are redirected to the `Local Factory` component located in

the *member*'s capsule which instantiates the necessary components if these have not been instantiated yet.

Because changes in the structure of the composite component are done via the compositional meta-model, joins and leaves of components (members) to/from a group are represented by requests to add or remove components to/from a composite component. At this point, the infrastructure presented in section 4 (Figure 2) is used.

Other changes in the group structure are also realised by means of reflection. Special adaptation managers are instantiated when the group and the members of the group are created. Each group has a `Group Adaptation Manager` (`GAM`) and each member of the group has a `Member Adaptation Manager` (`MAM`). See figure 3 for details. The `GAM` is responsible for receiving adaptation requests to be performed in the group (1). Then, the component that holds the adaptation rules is contacted (2) to verify if the adaptation can take place. Because groups can be heterogeneous, distinct adaptation can occur in each type of group member. Requests to perform individual adaptation are also accepted (e.g., destroy a particular member of the group). The `GAM` contacts the membership component (3) to identify the members of the group and so it can contact their `MAM` adapt interface in order to send the particular adaptation requests (4). When a `MAM` receives a request, it instantiates the meta-object associated to its member (5) and starts modifying the meta-object through its *ICompositionalMOP* interface presented in listing 1. Changes in the meta-object reflect in the actual component (6) as explained earlier.

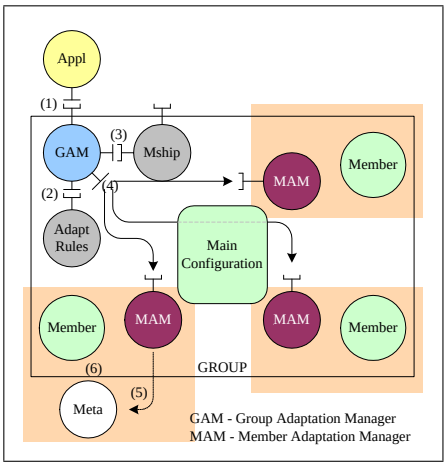

Figure 3: The group adaptation architecture.

Requests to perform adaptation can be issue individually, i.e., a single operation or in batch, i.e., a script which contains a set of operations. Requests to the `GAM` should be issued with the identification of the member or type of member where the changes should take place. Next section presents a simple example of adaptation.

6 Example

This example involves a group as the support for a simple audio and video conference application. The creation of the group has specified the following member types: an audio/video producer (`AVProd`), an audio producer (`AProd`), a video producer (`VProd`), an audio/video consumer (`AVCons`), an audio consumer (`ACons`) and a video consumer (`VCons`). Member types are specified in a template that also includes the minimum and maximum cardinality for each type of member and details about how members communicate with each other (the *main configuration* component) (more details in [10]). In terms of adaptation in this example, we suppose there are no restrictions on the type of adaptation that can take place. Figure 4 shows a possible configuration of the group described above with one `AVProd`, one `AVCons` and one `VCons`. The components `vs`, `as`, `vp` and `ap` are the proxies to the application and `vb` and `ab` are video and audio distributed bindings, respectively.

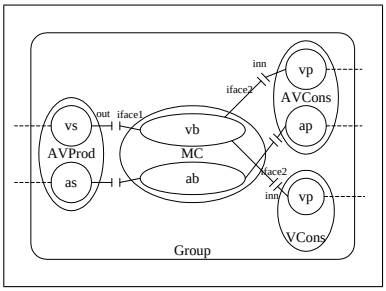

Figure 4: AV Example.

The adaptation we propose here is the removal of all components associated with the transmission of video, leaving the group only with audio components. In order to do so, the script shown in Figure 5 is sent to the group adaptation manager (`GAM`):

The script identifies the operations (in the form of a single operation or a script with one or more operations (see Figure 6)) to be performed in each member type or in a specific member. As well, operations can be performed in the main configuration (`MC`) component, which represents the explicit communication between members of the group. The `GAM` verifies the identification of the members of the group affected by the adaptation (`AVProd`, `VProd`, `VCons`, `AVCons` and `MC`) and sends the associated operations to the adequate member adaptation manager (`MAM`) [1]. A number of operations (tagged `G` in the script) are performed by the `GAM` itself. Because

[1] Note that the `MC` also has a `MAM`.

```
AVProd::stop("vs")
VProd::stop("vs")
AVCons::stop("vp")
VCons::stop("vp")
MC::http://location/scripts/StopMC.scp
G::breakLocalBind(("vs", "out"), ("vb", "iface1"))
G::breakLocalBind(("vp", "inn"), ("vb", "iface2"))
MC::delIfaceExternal(("vb"), ("iface1"))
MC::delIfaceExternal(("vb"), ("iface2"))
AVProd::delIfaceExternal(("vs"), ("out"))
VProd::delIfaceExternal(("vs"), ("out"))
AVCons::delIfaceExternal(("vp"), ("inn"))
VCons::delIfaceExternal(("vp"), ("inn"))
AVProd::delComponent("vs")
VProd::delComponent("vs")
AVCons::delComponent("vp")
VCons::delComponent("vp")
MC::delComponent("vb")
```

Figure 5: Group adaptation script.

```
StopMC.scp
Stop("vb")
```

Figure 6: Main configuration script.

a group is a composite component composed of a number of member components and a `MC` component, the `GAM` has to control the connections (bindings) between these components. This is why the script explicitly removes external interfaces of other composite components.

Start and stop operations can be issued by the `MAM` to freeze a component if it implements the *ISysCtrl* interface. This interface also provides methods for getting and setting the components state. Operations that are present in the *ICompositionalMOP* interface (e.g., *delComponent*) are passed to the respective meta-object.

Individual operations can be performed by getting the member identification with the membership component. The following is an example of operation in a specific member: `VCons:member:6::delComponent("vc")`. This operation is invoked by using *adaptMember()* rather than *adaptType()* on the `GAM` adapt interface.

7 Conclusion

The present paper has presented the experiments in applying reflection for performing adaptation in a group context. Each group is represented by a composite component and reflection is used to access and manipulated this component's internal structure. The example presented in Section 6 has been implemented using the OpenORB Python Prototype [1] along with the extensions proposed for the support for groups. This includes support for multiparty stream and operational bindings and other components used for group communication and management (e.g., membership, collation). Several applications can benefit from adaptation performed in a group. In this paper, we presented the modification of the configuration of a conference support. Other examples such as the insertion of a filter or a monitor can also be realised.

References

[1] A. Andersen. The Open-ORB Python Prototype API. NORUT IT Report IT302/2-99, NORUT IT, Oct. 1999.

[2] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 191–206. Springer-Verlag, 1998.

[3] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The Design and Implementation of OpenORB v2. *To appear in IEEE Distributed Systems Online*, 2(6), 2001. http://www.computer.org/dsonline/.

[4] F. Costa, G. Blair, and G. Coulson. Experiments with Reflective Middleware. In *Proceedings of ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems*, Brussels, Belgium, 20, July 1998.

[5] F. M. Costa, H. A. Duran, N. Parlavantzas, K. B. Saikoski, G. Blair, and G. Coulson. The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 79–99. Springer-Verlag, Heidelberg, Germany, June 2000.

[6] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake (Canada), October 1996.

[7] S. Maffeis. Adding Group Communication Fault-Tolerance to CORBA. In *Proceedings of USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995.

[8] A. Montresor. The JGroup Reliable Distributed Object Model. In *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, Helsinki, Finland, June 1999.

[9] P. Narasimhan, E. Moser, and P. M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering Journal*, 4(3):139–150, Sept. 1997.

[10] K. B. Saikoski, G. Coulson, and G. Blair. Configurable and Reconfigurable Group Services in a Component Based Middleware Environment. In *Proceedings of the SRDS Dependable System Middleware and Group Communication Workshop (DSMGC)*, Nürnberg, Germany, 15, October 2000.