Enabling Reflection and Reconfiguration in CORBA

Thais Batista
Departamento de Informática e Matemática Aplicada
UFRN
thais@dimap.ufrn.br, http://www.dimap.ufrn.br

Renato Cerqueira
Noemi Rodriguez
Departamento de Informática
PUC-Rio
rcerq, noemi@inf.puc-rio.br, http://www.inf.puc-rio.br

Abstract

The OMG CORBA specification offers a series of reflective mechanisms that enable dynamic component definition and dynamic configuration of applications. However, these mechanisms are very hard to use in conventional language bindings, causing programmers to avoid their use and consequently choose to develop applications with static behavior. This paper presents search facilities that take advantage of the dynamic nature of the LuaOrb binding to provide the programmer with simple interfaces for identifying appropriate components on the fly.

1 Introduction.

The dynamic and heterogeneous nature of current systems and networks has triggered interest in techniques that allow applications to obtain information about available resources and their usage and to reconfigure themselves using this information. The CORBA specification, similarly to other middleware technologies, contains a series of reflective mechanisms that support such behavior, allowing programs to query their runtime environments and determine current properties and services. The dynamic invocation interface (DII) allows a client to incorporate access to new interfaces at runtime. On the server side, the dynamic skeleton interface allows a program, also at runtime, to incorporate new interface implementations. The Trading, Naming, and Interface Repository services allow applications to obtain information about current active objects and the interfaces they implement.

However, although these reflective facilities have been available in several implemented ORBs for almost ten years, few systems have been reported to take advantage of them. We believe that this is largely due to the complexity of using and even of understanding the relevant mechanisms and service interfaces. Programmers prefer to create applications that are not so flexible as they could be than to complicate their code and development process by including access to CORBA's reflective mechanisms.

We have been involved for some years in investigating the flexibility that an interpreted language can add to component-based programming. The LuaOrb system [3] provides a binding from CORBA IDL to Lua [7], an interpreted programming language with dynamic typing and flexible extension mechanisms. This binding builds on the reflective facilities of Lua to provide transparent support for usage of DII and DSI. The LuaOrb system has been available for some years and its use in some projects [5, 6, 12, 4] has shown that programmers will be ready to use the dynamic interfaces if they are made easily available.

In this paper, we discuss how access to services that are relevant for dynamic configuration can be similarly simplified through the use of appropriate abstractions and programming interfaces. We focus on the problem of identifying appropriate objects when configuring a CORBA application.

The LuaSpace environment [2] builds on LuaOrb and related facilities to provide support for dynamic configuration of CORBA-based applications. In LuaSpace, the programmer uses Lua scripts to describe the composition of his application. In this paper, we describe the abstractions available for invoking objects that satisfy sets of functional and/or non-functional requirements.

The next section in this paper describes the search facilities in LuaSpace. We first describe the LuaTrading and LuaRep libraries, which simplify access to CORBA's interface repository and Trading Service, and then discuss a set of facilities that provide the programmer with high-level abstractions for searching for components. Section 3 dicusses scenarios in which these facilities are useful. Finally, section 4 contains some final remarks.

2 LuaSpace

LuaSpace is an environment for dynamic reconfiguration of component-based applications that follows a configuration-driven programming style where the structure of the application is independent of component implementations. In this environment, an interpreted and procedural language — Lua [7] — is used as the configuration language. Lua is dynamically typed: variables are not bound to types although values are. Lua includes conventional aspects (syntax and control structures simliar to those of Pascal) and also has several non-conventional features, such as functions as first class values, *tables*, which implement associatives arrays, and a set of reflective facilities. Among Lua´s reflective facilities, *tag methods* are the most generic mechanism. Several situations in which the interpreter would intuitively generate an error can be captured by a programmer-defined function, called a tag method, changing the language behavior in some way.

LuaSpace is composed by Lua and a set of tools based on Lua. At the core of these tools is LuaOrb [3], a binding between Lua and CORBA based on CORBA´s Dynamic Invocation Interface (DII) and on Lua's tag method facility. This binding allows dynamic access to remote CORBA objects as if they were ordinary Lua objects. In addition, LuaOrb uses CORBA Dynamic Skeleton Interface (DSI) to allow dynamic instalation of new Lua objects in a running server. Most LuaSpace tools use LuaOrb to access CORBA objects.

2.1 Mirror Repositories: LuaRep and LuaTrading

In this section we describe the Lua libraries that allow the LuaSpace programmer to access the Interface Repository and Trading Service. Although they are very different as to the service they provide, both the Interface Repository and the Trading Service act as information *repositories* in CORBA environments: interface descriptions, in one case, and service descriptions and offers, in the other, are maintained for queries and updates. IDL interfaces for these services are quite complex: the programmer must typically invoke a large set of methods in order to obtain some basic information such as the signature of a given method. LuaRep and LuaTrading create a simplified vision of these services by mirroring the information repositories into local Lua tables which act as proxies for the remote repositories. All objects contained in the repositories appear as fields in the proxy. On the other hand, assigning a new value to a field in one of these tables will insert this value in the remote repository. The libraries and the unifying concept of mirror repositories are described in detail in [9].

2.1.1 LuaTrading

The CORBA Trading Service is often described as a yellow pages system [13]. It contains *service offers* that associate object references with service types. However, because the service types are not as widely known as the ones in the yellow pages (painters, plumbers, etc), the Trading Service also maintains service type descriptions. These descriptions contain IDL interface identifiers and sets of properties. A service offer must contain values for the properties defined for the corresponding service type.

The act of announcing a new service in a Trading Service is called *exporting* a service offer. Symmetrically, we say a program is *importing* service offers when it retrieves service offers from the repository.

One important feature of the Trading Service is the provision of *dynamic* properties. Dynamic properties accomodate information that must be updated frequently: values for dynamic properties are references to objects that can provide the current property value. This mechanism is specially relevant in the context of dynamic configuration and

adaptation, since it can be used to maintain information about resource usage and availability.

To query and update the information on a Trader, the programmer must first obtain a proxy for the appropriate repository. As an example, to obtain a handle for the repository of type descriptions the programmer can write:

```
typesRep = serviceTypesRep()
```

and then "walk through" the descriptions, for instance using Lua's predefined *foreach* function:

```
foreach(typesRep.servicetypes, print)
```

which will invoke function *print* for each field in table `typesRep.servicetypes`. The program code can use `typesRep` to select an appropriate service type for the running application and then import service offers associated to that type.

To import a service offer, the programmer may use the *importServiceOffers* constructor. This constructor may be invoked with no arguments, causing the search to return all offers for this service type, or with arguments specifying property values to be returned or constraints on property values, as in the following code:

```
offers = importServiceOffers{
          type="CertAuth",
          properties = {"Methods",
                        "Price"},
          constraint = "Price < 100",
          pref = "min Price",
          maxret = 3}
```

This invocation will assign to `offers` a table containing at most three (`maxret` argument) service offers of type `"CertAuth"` as well as the values of properties `"Methods"` and `"Price"` for these offers. The `constraint` argument specifies that only offers with prices below 100 should bee returned, and `pref` establishes the desired ordering for `offers`.

LuaTrading offers similar facilities for creating new service types, exporting service offers, updating properties, and de-registering offers.

2.1.2 LuaRep

CORBA requires that any ORB implementation implement the Interface Repository (IR) interface [10, 13]. The Interface Repository is a regular CORBA object whose use is mandatory in both DSI and DII.

The LuaRep library allows the Lua programmer to directly manipulate the information in the Interface Repository through a proxy, adding new interfaces or retrieving information about registered interfaces. This allows LuaOrb clients to issue requests to services they have discovered dynamically and to retrieve the signature of interfaces that have been previously returned by LuaTrading operations.

To obtain a proxy for the Interface Repository, the programmer must use the *InterfaceRep* constructor:

```
ir = InterfaceRep{}
```

Supposing that the programmer wants to know the definition of *book*, which is defined in the repository as a *struct*, he may write:

```
foreach(ir.book, print)
```

This code will print the name and type of each of the fields defined in the Interface Repository for *book*.

2.2 Dynamic Component Selection

LuaSpace builds on the LuaRep and LuaTrading libraries to provide support for higher-level mechanisms that simplify dynamic component selection.

The *search* function dynamically searches for objects that satisfy different search criteria. Possible criteria are: object name, signatures of methods, and properties (non-functional features). This function receives as a parameter a combination of search criteria and returns a list of objects that satisfy the given requirements. Its argument is a Lua table that contains criterion types (*name*, *operation*, or *properties*) and associated expressions. Logical operators (and, or, not) are used to combine the different criterion types and to determine the relationship between different items associated with a same selection criterion. The following code shows an example of the use of *search*.

```
listdoctor = search{
 {NAME="doctor"}, "and",
 {PROPERTIES =
   "hospital='MedicalCenter' and
    speciality =´pneumologist´"}, "or",
 {OPERATION = "urgentcall()"}}
```

The use of the search function allows the programmer to invoke a single function independently of the search criterion, while at a lower level each criterion is related to a different CORBA service. The CORBA Name Service provides access to objects by

their name and the CORBA Trading Service allows the programmer to search for objects according to their properties. The search function provides an uniform and easy-to-use interface.

One important aspect of the search function mechanism is that it provides support for structural compatibility [11]. When looking for a component that will act as a service supplier in a distributed application, it is often the case that the programmer is not interested in the precise interface that the object implements: what he needs is a component that provides a set of methods with the expected signatures. Searching for an object that implements a given set of methods requires using the Trading Service to find out what interface is implemented by each registered object and then accessing the Interface Repository to determine the signatures of the methods in each interface. The search function hides the complexity of this procedure from the programmer, creating an abstraction by means of which he can search for all objects with interfaces that are structurally compatible with the one given as a parameter to *search*.

Another selection mechanism offered by LuaSpace is the *Collection* mechanism. Collections are Lua tables that group sets of components with some common feature. A collection acts as a broker for the objects it contains in that their methods can be invoked directly on the collection object. This invocation triggers the dynamic selection of the instance that will handle the request. This implicit selection is done according the selection strategy associated to the Collection. This strategy is implemented as a Lua function that is called in every method request in order to select the collection member that should handle the request. The selection strategy could favor load balancing, fault tolerance, or a combination of these and other non-functional requirements (A set of tools described in [4] provide support for programming such selection strategies.). The collection's members can be explicitly inserted or can be the result of an invocation of the *search* function.

Combining the search and collection mechanisms, the LuaSpace programmer can create a number of powerful abstractions. One example we have explored is the *generic connector* [1]. The generic connector allows the programmer to dynamically invoke any required method on a *generic* object. This generic object selects an active component whose signature is compatible with the invoked method, and then goes through with the invocation, acting as a generic provider of the requested method. An invocation on a generic connector can also, optionally, provide a set of properties as an argument. The generic connector invokes the *search* function to look for appropriate components, and stores the resulting objects in a collection; finally, it invokes the required method and sends the returned results back to the caller.

3 First Experiments with LuaSpace

To evaluate the dynamic reconfiguration features of LuaSpace and its related tools, we have been applying them to different application scenarios. We have used it to prototype a collaborative CAD environment [5] and to build a multiple display viewing system for virtual environments [6]. But by far the most exciting scenario in which we have applied LuaOrb and LuaSpace was a middleware for ubiquitous computing applications, called Gaia [12]. Gaia is a middleware infrastructure that coordinates software entities and heterogeneous networked devices contained in a physical space. Typically, ubiquitous computing applications strongly demand dynamic adaptation capabilities, since they can be affected by many external factors, such as user mobility, resource availability, and different contextual properties.

Gaia is based on CORBA and its standard services, namely Trading Service, Naming Service, and Event Service. These and other specific Gaia services, such as a context service, a presence service, a context-aware file system, and a component management subsystem, provide a powerful software infrastructure that converts physical spaces and their ubiquitous computing devices into a programmable computing system. However, all these services and the meta-information they provide about their digital and physical environments make the task of developing applications with traditional CORBA programming languages, such as Java and C++, even more difficult.

In order to provide a better programming support, Gaia uses LuaSpace mechanisms to automate management and configuration tasks, describe and create ubiquitous computing scenarios, test components, and prototype new applications.

Currently, we are working on a specialized version of Gaia to support distributed and collaborative sci-

entific applications. This new infrastructure, called GaiaVR, has two new challenges: In the back-end, it has to integrate a grid of computational resources and, in the front-end, it has to allow user interaction through geographically distributed immersive visualization rooms. In GaiaVR, we are using LuaTrading to simplify system management and application development. We are using the search mechanism of LuaSpace to specify *application templates*. Instead of specifying the exact components, an application template provides a more abstract description of the components that should be used to assemble a GaiaVR application. These descriptions are based on functional and non-functional properties of the desired components. LuaSpace's search mechanism uses these descriptions to select the actual components.

4 Final Remarks

In this paper, we presented an environment that offers facilities for dynamic configuration and management of applications. The facilities we described are not novel in themselves: the functionality they offer is also available directly from CORBA's services and features. However, we believe that the complexity of using CORBA's reflective mechanisms directly from a conventional language binding is such that programmers avoid creating applications that make use of them. It is thus important to create new programming interfaces that make reflective features effectively available to programmers. Since there is a consensus that middleware technologies will continue incorporating new reflective features [8], such programming interface should become even more important in a near future.

Our previous work with Lua and LuaOrb has been building up into a series of support tools which now make it easier to explore the presented ideas in the Lua environment. However, the search mechanisms we described could also have been built in other interpreted languages, such as CorbaScript or Python.

References

[1] T. Batista, C. Chavez, and N. Rodriguez. Dynamic Reconfiguration through a Generic Connector. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, volume II, pages 1127 – 1132, Las Vegas - Nevada - USA, June 2000. CSREA Press.

[2] T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-based Applications. In *5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000)*, pages 32–39, Limerick, Ireland, 10-11 June 2000. IEEE, IEEE Computer Society.

[3] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic component gluing across different componentware systems. In *International Symposium on Distributed Objects and Applications (DOA'99)*, pages 362–371, Edinburgh, Scotland, September 1999. OMG, IEEE Press.

[4] A. L. de Moura, C. Ururahy, R. Cerqueira, and N. Rodriguez. Dynamic support for distributed auto-adaptive applications. In *Proceedings of AOPDCS - Workshop on Aspect Oriented Programming for Distributed Computing Systems (held in conjunction with IEEE ICDCS 2002)*, pages 451–456, Vienna, Austria, July 2002.

[5] B. Feijó, P. Rodacki, J. Bento, S. Scheer, and R. Cerqueira. Reactive design agents in solid modelling. In J.S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design'98*, pages 557–577. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[6] A. Ferreira, R. Cerqueira, W. Celes, and Marcelo Gattass. Multiple display viewing architecture for virtual environments over heterogeneous networks. In *Proceedings of SIBGRAPI'99*, pages 83–92, Campinas, Brazil, 1999. SBC, IEEE Computer Society.

[7] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[8] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of The ACM*, 45(6):33–38, 2002.

[9] L. Nogueira. Um ambiente de gerência de aplicações CORBA. Master's thesis, Depto de Informática, PUC-Rio, 2001.

[10] OMG. The Common Object Broker Architecture and Specification. Technical Report Revision 2.2, OMG, 1998.

[11] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *Sigplan Notices*, 28(8):81–89, 1993.

[12] M. Román, C. Hess, R. Cerqueira, Anand Ranganat, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[13] Jon Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.