Dynamic Reconfiguration through a Generic Connector

Thaís Batista*
Informatics Department
Federal University of Rio Grande do Norte
Natal, RN, Brazil

Christina von Flach Chavez*
Computer Science Department
Distributed Systems Laboratory
Federal University of Bahia
Salvador, BA, Brazil

Noemi Rodriguez
Computer Science Department
Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, RJ, Brazil

Abstract *In component-based programming, it is often desirable that applications can be modeled as sets of services executed by components selected at runtime. In order to support this requirement, we present the Generic Connector, a mechanism that searches for components that provide the services required in the application configuration program, activates each service and returns the results to the client. The Generic Connector explores Lua's and LuaOrb's intrinsic characteristics. Lua is a dynamically typed, interpreted language and LuaOrb is a binding between CORBA and Lua that provides dynamic access to CORBA objects. The Generic Connector allows the configuration of applications as sets of services without previous knowledge about the components that will actually provide them. As components are selected on-the-fly, the Generic Connector turns out to be a powerful mechanism for dynamic reconfiguration.*

Keywords: dynamic reconfiguration, component-based programming, client-server style, software connector

1 Introduction

Component-based programming has been the subject of recent research [1, 2, 3], especially for its support for component reuse, reducing time and costs of software development. In this work, we use the term *components* to describe separately compiled software entities that provide some functionality described by their interfaces and may interact with other components to create an application.

In component-based programming, it is often desirable that applications can be modeled as sets of services executed by components that are selected at runtime. In order to support this requirement, we present the Generic Connector, a mechanism that searches for components that provide the services required in the application configuration program, activates each service and returns the results to the client.

The Generic Connector is proposed in the context of LuaSpace [4], an environment for dynamic reconfiguration of component-based applications centered on Lua [5], a dynamically typed, interpreted programming language. Lua is used as a configuration language since it provides a flexible type system that supports dynamic access to components. The dynamic access to CORBA objects is provided by LuaOrb [6], a binding between CORBA and Lua, based on the Dynamic Invocation Interface (DII). Lua and LuaOrb's intrinsic characteristics are extensively used by the Generic Connector.

The purpose of the Connector presented in

*PhD Student at PUC-Rio

this work is different from that of connectors used in Software Architecture [7, 8, 9], which describe the kinds of components that take part in an interaction and the ways they relate to each other, imposing restrictions on how the messages flow between components. Instead, our concept of "connector" derives from the idea of an executable entity that controls the interaction among components [10], providing a way to isolate the implementation of components from the interconnection aspects. The Generic Connector supports only the client-server pattern of interaction. The "generic" designation is used here to specify that it is not supposed to bind specific, previously defined components, but instead, to look for any component that is able to provide the required service, regardless of its component model (CORBA, COM, JAVA, etc.), as long as a binding between the configuration language and the component model exists.

Through the Generic Connector, it is possible to configure an application as a set of services without previous knowledge about the components that provide them. These components will be selected on-the-fly. Thus, this is a powerful mechanism for dynamic reconfiguration.

In addition, with the Generic Connector facility, Lua's configuration environment offers two-level support for the configuration of applications. In the first level, the programmer defines, explicitly, the components that provide the desired services. In the other level, the programmer specifies the desired services, no matter which component will provide them, and explores the flexibility offered by the Generic Connector, that, in turn, abstracts away from the complexity related to the search of components.

This work is structured as follows. Section 2 briefly introduces some Lua and LuaOrb concepts. Section 3 details the Generic Connector and some examples of its use. Finally, in Section 4, some conclusions are discussed.

2 Lua and LuaOrb

Lua [5] is an interpreted extension language developed at PUC-Rio. Lua is dynamically typed: variables are not bound to types, although values are. Lua includes conventional aspects, such as syntax and control structures similar to those of Pascal. However, it also has several non-conventional features, such as functions as first class values, *tables*, which implement associative arrays, and a set of reflexive facilities. Among Lua's reflexive facilities, *fallbacks* 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 fallback. Examples of such situations are calls to nonexistent functions and indexing a value that is not a table.

LuaOrb [6] is a binding between Lua and CORBA that, as usual, defines mappings between Lua and IDL types. The unusual aspect here is that, unlike most systems that use the static approach to define a mapping to CORBA, requiring recompilation whenever a change takes place either at the client or at the server, LuaOrb's mapping is based on Lua's dynamic typing and on CORBA's Dynamic Invocation Interface (DII) [11], which allows the use of arbitrary CORBA objects at runtime (without compiled stubs). On the server's side, LuaOrb uses the CORBA Dynamic Skeleton Interface (DSI) [11] to permit dynamic installation of implementations written in Lua.

On the client's side, LuaOrb explores the dynamic features of Lua to offer a way to access CORBA objects available at remote servers exactly like any other Lua object. This is done transparently and at runtime. To use a CORBA object, a Lua proxy to the CORBA object must be created, invoking the `createproxy` function with the required interface name and the object reference as arguments. Function `createproxy` returns a Lua object that represents a CORBA object. Using Lua's fallback mechanism, LuaOrb detects the operations applied to the proxy and transforms them into remote operations. The imple-

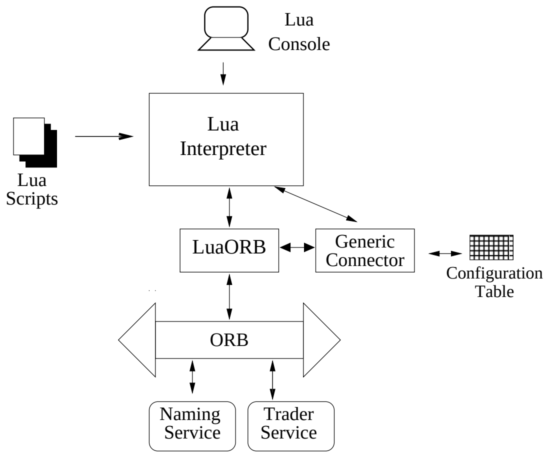

Figure 1: The Generic Connector

mentation of `createproxy` is basically a direct mapping to its equivalent function in the ORB API.

On the server's side, LuaOrb uses the facilities provided by DSI to allow the extension of CORBA objects via Lua [12]. Lua objects may play the role of CORBA servers and clients can access them via stubs or DII. A proxy to the Lua object is created and this object can be dynamically installed on the server. The interface `ServerLua` has been defined to allow the instantiation of a new object and the definition of its implementation with Lua code.

More detailed information about Lua and LuaOrb can be found in:

`http://www.tecgraf.puc-rio.br/lua/`.

3 The Generic Connector

The Generic Connector is an abstraction that supports the configuration of an application without determining in advance the components that will provide the services specified in a configuration program: these components will be selected on-the-fly. In the configuration program, an "orphan" service is invoked in the same way as a service of a known component is invoked, but with a variable that represents the Generic Connector in the place of the name of a component, suggesting that the service will be provided by the Generic Connector. At execution time any component may be selected to perform the same service previously provided by other component, thus promoting a way of dynamically reconfiguring the application. Figure 1 shows the context of use of the Generic Connector.

A configuration program is a Lua script that, besides ordinary Lua statements, may include statements to access CORBA components and to execute their services. These services are directly activated through some proxy returned after a LuaOrb's `createproxy` invocation or are intercepted by the Generic Connector. Upon intercepting a service request, the Connector searches for the service in some local repository (constructed on top of Trader or Naming services), gets a list of components that implement it, and, after a component is selected, activates the service and returns the results. A *configuration table* stores the required service and the component that was dynamically selected to provide it. This table provides, at anytime, a snapshot of the available components introduced on-the-fly by the Connector.

We present a very simple example to illustrate the Generic Connector mechanism. Suppose that a print service is provided by some servers that implement the interface `Printer` (Figure 2). Each server manages a single printer and if the printing job is completed with success, a value other than 0 is returned.

```
interface Printer {
  short print_service(in file);
}
```

Figure 2: Printer interface

Figure 3 shows a configuration program in Lua that uses `print_service`. A Lua proxy to a component with interface type `Printer` and IOR file `stylus.ref` is returned after a call to `createproxy` and is stored in `p`. Every subsequent reference to p:print_service will select the method print_service of the specific component represented by p. When this component is not available, the print service fails.

```
p = createproxy{interface="Printer",
                IOR_file="stylus.ref"}

status = p:print_service(theFile)
if status == 0 then
  write("print failed")
end
```

Figure 3: print_service only with LuaOrb

Figure 4 presents a configuration program that uses print_service through the Generic Connector, i.e., without specifying the component that will provide the service.

```
p = generic_createproxy()

status = p:print_service(theFile)
while status == 0 do
   status = p:print_service(theFile)
end
```

Figure 4: print_service with the Generic Connector

Function generic_createproxy returns a proxy that is not bound to a specific component – this proxy refers to a Lua object, the Generic Connector itself. The Generic Connector's implementation intercepts the call p:print_service(theFile) (using Lua's fallback mechanism), and implicitly invokes an internal function with the required service name (print_service), the sequence of parameters (theFile) and the client name. As a first step, this function searches in a standard repository for some component that offers print_service with the same signature provided by the parameter. Once a set of components that provide print_service(theFile) is available, one of them is selected (sometimes requiring user participation). Eventually, the Generic Connector builds the *service solicitation*, a sequence of commands written in Lua to create the proxy for the selected component (using LuaOrb) and to activate the service. Finally, the service is executed and the results are returned.

For each established connection, a record is stored in the *configuration table*, with the following information: (i) a pair (service_name, parameters_signature), (ii) some identification of the actual executing component that offered the service and (iii) the name of the client that required it. Before searching for service_name(parameters) in the default repository, the Generic Connector queries its *configuration table* to check if the same service has already been used previously, avoiding unnecessary access to the default repository. Alternatively, it is possible to define that all searches must be done upon the repository and the *configuration table* is not used. As component selection is done at execution time and, each time, any component may be selected to perform the same service, this mechanism promotes a way of dynamically reconfiguring the application.

The potential for reconfiguration is an intrinsic aspect of the Generic Connector's philosophy. Considering the same example as before, suppose that some problem occurs during the printing job, and a status with value 0 is returned. Other component that provides print_service can be selected without demanding previous knowledge about its identity. This is illustrated inside the while loop in Figure 4. Finally, if the last component selected is not available in a subsequent execution of the application, the Generic Connector will select another component. This scenario illustrates how reconfiguration can be done automatically, without explicit reconfiguration points defined by the programmer. This issue introduces a great flexibility to the application.

The flexibility provided by the Generic Connector is well-suited for the configuration of CORBA-based applications, which consist of a set of services offered by several components. Besides, it is compatible with the idea of a repository of components providing several services and the Generic Connector allowing the programmer to abstract himself from the task of searching for components that offer the desired services.

This approach increases the level of component reusability, since it allows the use of components as they were designed (without changes), and the components are unaware about the interactions in which they will take part. It differs from other approaches where the component interfaces are sometimes modified to participate in new interactions [13] or where the bindings among components are determined inside the definition of a composite component [3].

The Generic Connector's functionality goes beyond that provided by Naming and Trader Services, that only provide repositories with some well-defined ways to query about component information stored there. The Generic Connector provides to the application the ability for reconfiguration as well as an uniform way to use services from previously selected components or from unknown components. Besides, it also activates the required service from the selected component and returns the results.

4 Conclusions

This work has presented an important tool for structuring and reconfiguring component-based applications: the Generic Connector. This tool offers a flexible way to configure applications, by declaring sets of services without previous knowledge of the components that implement them.

One of the main advantages of this approach is to offer an implicit way of selecting components and interconnecting them to build an application. This support frees the programmer from the responsibilities of searching for the right components. Another interesting aspect refers to the potential of automatic reconfiguration that the Generic Connector introduces in the application, since in each call for a service, distinct components can be selected to perform it. As a consequence, Lua's configuration environment offers an additional level of support for application reconfiguration. In another level, traditionally approached by reconfigurable systems, reconfiguration is explicitly established.

Although the mechanism has been implemented using Lua and a binding to CORBA, the same philosophy can be adopted by similar approaches using programming languages with dynamic features similar to those provided by Lua, and dynamic bindings to other component models. These aspects allow that applications can be configured without relying on previous declaration of components that will only be found at runtime.

References

[1] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[2] J. Schneider and O. Nierstrasz. Components, scripts and glue. Software Architectures - Advances an Applications, Springer 1999. available at http://www.iam.unibe.ch/cgi-bin/.

[3] J. Magee, A. Tseng, and J. Kramer. Composing distributed objects in corba. In *Third International Symposium on Autonomous Decentralized Systems - ISADS 97*, pages 9–11, Berlin, Germany, April 9-11 1997.

[4] T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-based Applications. *Proc. PDSE-2000 (5th International Symposium on Software Engi-*

neering for Parallel and Distributed Systems), Limerick, Ireland, June 2000.

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

[6] 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.

[7] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*, pages 2–10, Annapolis, Maryland, May 6-8 1996.

[8] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie-Mellon University, 1997.

[9] J. Bishop and R. Faria. Connectors in configuration programming languages: are they necessary? In *Third International Conference on Configurable Distributed Systems*, pages 11–18, Annapolis, Maryland, May 6-8 1996.

[10] S. Ducasse and T. Richner. Executable connectors: Towards reusable design elements. In *Proceedings of ESEC/FSE'97, LNCS 1301*, pages 483–500, 1997.

[11] J. Siegel. OMG overview: Corba and the OMA in enterprise computing. *Communications of the ACM*, 41(10):37–43, October 1998.

[12] M. Catunda, N. Rodriguez, and R. Ierusalimschy. Dynamic extension of CORBA servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, August 1999. Springer-Verlag. LNCS 1685.

[13] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. *Object-Based Distributed Programming (ECOOP'93 Workshop), LNCs 791*, pages 152–184, 1994.