Dynamic Component Gluing Across Different Componentware Systems

Renato Cerqueira
Carlos Cassino
Roberto Ierusalimschy

Departamento de Informática — PUC-Rio, Brazil
rcerq,cassino,roberto@inf.puc-rio.br

Abstract

In this paper, we present a model to support dynamic gluing of components from different componentware systems, such as CORBA, COM and Java. This model adopts a dynamic composition approach combined to the interactive facilities offered by an interpreted language. Based on this model, we define a common design for the implementation of bindings between glue languages and component systems. The proposed design provides a natural way to create dynamic bridges between different component systems. We also describe the LuaOrb system, an implementation of the proposed model that binds the interpreted language Lua to Java, CORBA and COM. With LuaOrb, Lua acts as a unifying glue language, wherein we can write code that freely use and mix components from these three component systems.

Keywords: Componentware, Glue Languages, Interoperability.

1. Introduction

A very important aspect for component-based software development is the mechanism used to connect components. Too often, the component connections are defined in the same language used to implement the components, and architecture aspects are mixed with the component implementations. However, there is an increasing development of new mechanisms oriented to link components. With these mechanisms, the application architecture and the component implementations are disconnected. This separation allows a better comprehension of the application architecture and also simplifies maintenance tasks, such as changes in the software architecture and substitutions of components that do not affect the system architecture.

Some of these composition mechanisms are based on configuration languages, such as Darwin [31, 30], UniCon [49, 3], Prescription [53], and Aster [23]. Other mechanisms are based on visual composition tools, such as graphical environments to support Darwin [27, 36, 38, 37] and JavaBeans [24], and the Vista environment [9]. Finally, scripting languages, such as Tcl [44], Python [29], Lua [22], JavaScript [11], and Visual Basic [6], are also popular tools for gluing components [43, 40, 7, 41, 26, 15].

A relevant feature of a componentware system, which is directly related to composition mechanisms, is its capacity to perform dynamic changes in the application composition. This feature is very important in some cases that need extra flexibility, such as desktop and operating systems [50], network management tools [2, 46, 50], cooperative applications [32, 17, 12], and critical systems in general [51, 19, 20, 55, 35]. All those kinds of systems can take advantage of the dynamic composition feature to replace or add components without stopping the system execution.

Another challenge to composition mechanisms arises when we need to integrate components from different componentware systems. For instance, CORBA is strongly focused on heterogeneous distributed systems, while COM offers a better support to compound documents and desktop applications [48]. Without some mechanism that allows the interoperability between components from different componentware systems, the reuse of those components is restricted.

In this paper, we propose a model to support dynamic application composition, that uses an interpreted language as its gluing element among components from different systems. This model adopts a dynamic composition approach combined with the interpreted language. Applying this model to different componentware systems, we can connect heterogeneous components through bridges created dynamically with the help of the glue language.

Interpreted languages have an important role in the software development process: They are very useful for rapid prototyping, mainly as a tool for requirement acquisition [8, 43]. An interactive console is useful during system prototyping and development, allowing new modules to be added and tested without test stubs. Another feature of interpreted languages is that they do not need re-compilation and relinking after each change. This feature is useful for building little programs (specially for end user programming), and it is a main ingredient for rapid prototyping, where several design options can be implemented and tested in a short period of time.

The integration between interpreted languages and componentware systems that is used in our model is a symbiotic relationship: From the prototyping point of view, this union offers an important flexibility for an exploratory programming approach; from the distributed system point of view, it provides a powerful support to dynamic reconfiguration [45].

To allow the component gluing with an interpreted language, we need an interface between the language and the component systems. This kind of interface is usually called *language binding*. As we are interested in dynamic composition, we need a language binding that allows, at run-time, the identification of new component types and the integration of their instances with a specific application. To fulfill these needs, we define a common design for the implementation of bindings between glue languages and component systems. Our design is strongly based on reflexive mechanisms that should be provided by the glue language and the component system. It not only offers access to available components, through the use of *generic proxies*, but also provides support to dynamic implementation of components using the glue language, through the use of *generic adaptors*.

We can apply our design to different component systems and glue languages, as long as they offer the required reflexive mechanisms. As our language bindings follow that common design, we can achieve dynamic bridges between different componentware systems through the connection of generic proxies and adaptors.

In this paper, we also describe the LuaOrb system, an implementation of the proposed model that allows the use and mix of Java, CORBA and COM components to build applications. To support dynamic application composition, LuaOrb uses the interpreted language Lua [22, 13] as its glue language. As in other language bindings for componentware, this binding allows the components of those systems to be used in the same way as other native objects. The main reasons for choosing Lua were its simplicity, portability (useful for running implementations both in Windows and Unix), reflexive facilities, and a small memory footprint when embedded in an application.

The next section discusses the dynamic mechanisms provided by the component systems that we need for our language binding design. The dynamic approach proposed in this paper, including the language binding design and the dynamic bridges between componentware systems, is described in Section 3. The LuaOrb system is presented in Section 4, which also presents some examples. The last section is reserved for some final remarks.

2. Dynamic Mechanisms of Component Systems

In the last few years, CORBA [42, 50], COM [4, 47] and Java [1, 18, 24] have been gaining momentum as the main platforms for construction of componentware. Each one of these architectures defines a *component system*, with a particular set of mechanisms to allow the construction and connection of its components.

Besides the traditional mechanisms to build applications using static typing, CORBA, COM and Java offer mechanisms that allow dynamic integration of new components to applications. These mechanisms are the basic support to dynamic composition, since programs can incorporate new components, and even new interfaces, without changes or recompilations. These mechanisms are based on techniques of reflexivity and meta-programming [25].

The language binding design we describe in Section 3 requires some of these meta-programming mechanisms from a component system. Basically, the required mechanisms are:

- Introspection: more specifically, the ability of inspecting at run-time the complete component type descriptions;
- Dynamic calls: the ability of building method calls at run-time;
- Dynamic implementations: the ability of building new component implementations at run-time.

These mechanisms are presented in those three componentware systems in different forms. CORBA is strongly based on the notion of object interfaces, described in *IDL* (Interface Description Language). Information about object interfaces may be obtained dynamically through queries to the *Interface Repository* (IR), which maintains a description of interfaces of available object servers in machine readable form. To support dynamic calls, CORBA supplies a specific interface, called *Dynamic Invocation Interface* (DII): Using DII, a program can dynamically build method calls and invoke them. The CORBA's *Dynamic Skeleton Interface* (DSI) is the CORBA interface that allows the building of new object implementations at run-time.

The approach taken by COM is quite similar. A COM component can supply a *type library*, and the COM API supports dynamic inspection of all interfaces stored in those libraries. To allow dynamic calls, a COM component must offer a special interface, called *dispatch interface*. The dispatch interface has a main operation called `Invoke`. This operation takes an operation selector and a generic variant structure with the desired operation arguments.[1] To implement new components at run-time, the COM dispatch interface can also be used. For this implementation, it is necessary to provide only an object adaptor that offers the dispatch interface and can receive a generic `Invoke` request and re-dispatch the selected operation call for the real component implementation, provided at run-time.

The meta-programming support provided by Java is based on its Core Reflection API. This API allows the inspection of types and the construction of dynamic method calls, using specific classes that model types, fields and methods. In contrast with CORBA and COM, Java objects always carry their own type information. Java does not supply a mechanism for dynamic implementations. This drawback can be circumvented with the Class Loader facility, plus some extra work to generate binary code for the new class at run-time.

3. A Unifying Model for Dynamic Composition

This section presents our design for language bindings that allows the implementation of bindings between different component systems and a single unifying gluing language. Our model for component integration relies on the reflexive facilities of these systems and the dynamic nature of interpreted languages. This section also presents how that kind of language binding can provide dynamic bridges between different componentware systems.

A mechanism to support dynamic composition should be interactive, that is, it should allow an operator to interact directly with the system to extend or adapt an application. Moreover, the mechanism should offer to the operator a simple way to express her intentions. Typically, interpreted languages present these both characteristics: interactivity and support to end-user programming. Therefore, interpreted languages are an interesting technique to be used as a gluing element between components [43, 33, 5].

Besides providing applications with a high level of flexibility in their architectures, a composition mechanism based on an interpreted language is also useful for rapid prototyping, as available components can be used during the prototyping process [8, 43]. With such mechanism, we can achieve a final application with a high level of flexibility from successive refinements of its prototype.

[1] In contrast with CORBA and Java, the generic variant structure offered by COM to define the operation arguments cannot represent all data types supported by the COM model. So, not all component interfaces can be accessed with the COM dispatch interface, but only *OLE Automation* compliant interfaces [4].

3.1. Dynamic Language Binding

As we saw in Section 2, component systems typically support both introspection and dynamic construction of method calls. Our language binding design uses these features for the creation of bindings where scripts written in a gluing language can incorporate and make effective use of new component interfaces in run time; moreover, as in other bindings, the language should be able to use external components transparently, in the same way it uses native objects. We name this design *dynamic language binding*.

The first step in the solution adopted by the binding design is the creation of *proxies* for each external component to be handled by a program, following the design pattern *proxy* [14].

A proxy is a native object created to represent an external one; its basic functionality is to delegate to the external object it represents any operation performed over itself. To implement this interface, the design prescribes the use of the reflexive mechanisms of both the component system and the scripting language.

The interpreted language must have a mechanism such that operations applied to the proxy, made through the regular language syntax, can be detected and handled by the binding that created the proxy. Moreover, when the binding is evoked, it must be able to access any information concerning the original operation (the “receiver” proxy, the operation name, eventual arguments, etc). Typically, this can be provided through overloading, in static typed languages (such as C++), or through an exception handling mechanism, in dynamic typed languages (such as Lua).

When an operation is applied to the proxy, the binding runs the following algorithm:

1. using introspection, it looks in the external object interface for a method with the given name;
2. if it cannot find such method, raises an exception;
3. otherwise, it gets the complete description of the method signature (formal parameters and return types);
4. for each formal parameter, it checks whether the corresponding argument has a compatible type (the compatibility rules are defined by the binding, through a set of direct translations plus possible coercions); if any argument is not compatible, raises an exception.
5. With the method found and the arguments converted, it dynamically builds the method call;
6. it does the actual call;
7. it converts back any results (or “out” parameters), again using the method description and the compatibility rules.

Other kinds of operations over a proxy, such as access to a field or array indexing, follow a similar pattern.

At a first sight, it seems that this kind of binding can be one-way: The glue language must be able to act as a client of external components, but it does not need to implement new *object servers*, to be called by the component system. However, to support callback objects—such as *listeners* in JavaBeans— the binding must work in both directions, since those callback objects must be called from the external components. Because the use of callback objects is a pervasive technique in object-oriented programming (for instance, most patterns presented in [14] use callback objects), a language without such support would have quite a limited use.

To provide dynamic object implementations with glue languages, our pattern prescribes *generic adaptors*. A generic adaptor is registered in the component system as an object that provides a specific interface, and delegates the real method implementations to an object defined in the glue language, in a way similar to proxies: Any operation performed over the adaptor is re-applied over the real implementation. Because these implementations are written in an interpreted language, they can be dynamically created and linked to an adaptor. The linking between the adaptor and the object uses the same reflexive mechanisms used by our client proxy, but this time these mechanisms are provided by the language interface. Therefore, the language API must also allow the dynamic construction of method calls.

The key point in this design is its dynamic nature. Every step, from type checking to method identification and invocation, is done at run time. Therefore, a program can incorporate new components, from different component systems, without any additional declarations and without stopping for recompilation.

3.2. Dynamic Bridges

Our dynamic language binding can also offer the dynamic creation of bridges between componentware systems. For a better comprehension of how this is achieved, consider the interface `Dispatcher`, defined in Figure 1 with the OMG IDL notation. This interface offers a single operation, `invoke`, that performs dynamic calls. The first parameter of `invoke` (`method_name`) names the operation to be requested, and `params_in` has the parameter list to be passed to the selected operation.[2] If the selected operation is performed with success, `invoke` returns `true` and the parameter `params_out` has the list of results returned by the operation. Otherwise, the method `invoke` returns `false` and the parameter `excep` has the description of the raised exception.

[2]The OMG IDL type `any` is similar to a *tagged union* that can represent any valid OMG IDL type.

```
typedef sequence<any> ListOfAny;

struct NamedValue {
   string name;
   any value;
};

typedef sequence<NamedValue> NamedValueList;

struct ExceptDescr {
   string name;
   NamedValueList fields;
};

interface Dispatcher {
  boolean invoke(in string method_name,
                 in ListOfAny params_in,
                 out ListOfAny params_out,
                 out ExceptDescr excep);
};
```

Figure 1. The interface `Dispatcher`.

The interface `Dispatcher`, with different implementations and abstraction levels, is pervasive in our design:

- The dynamic call mechanism of a componentware system (such as CORBA DII) is an implementation of the interface `Dispatcher`.
- The glue language API must offer such interface as well, so that a generic adaptor can build calls to functions written in the glue language.
- The proxies used in our design implement the interface `Dispatcher`. Those components implement the method `invoke` in accordance with the dispatching algorithm we previously described.
- A generic adaptor also implements the interface `Dispatcher`. We can see proxies and adaptors as translators between different dispatchers: While a proxy offers this interface to the glue language and uses the componentware system to delegate the calls, a generic adaptor offers the interface to the componentware system, and delegates the calls to code written in the glue language.

Figure 2 presents the basic configuration for the use of a generic proxy. The generic proxy exports the interface `Dispatcher`, so that clients can have dynamic access to the target component. The proxy itself is implemented over the interface `Dispatcher` provided by the componentware system. The target component, on the other hand, may

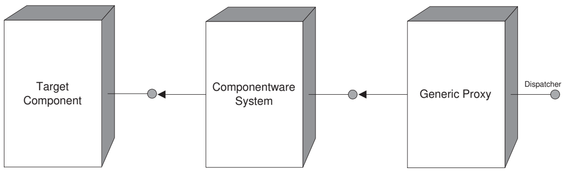

Figure 2. Basic architecture.

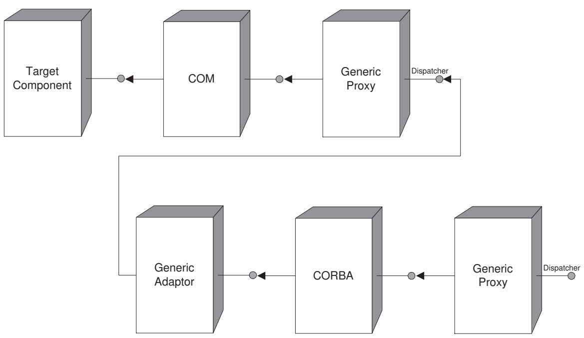

Figure 3. A dynamic bridge example.

be implemented through a static binding with the middleware.

Now, to create a dynamic bridge between components from different systems, we only need to connect a generic adaptor of a specific componentware system to a generic proxy of another system. That is, using the glue language, we instantiate a proxy for the target service, and then use this proxy as the implementation for a generic adaptor in the other componentware system. Figure 3 shows a sample of this integration architecture. In this example, a COM component is exported to a CORBA environment, where method requests will trigger a sequence of mappings:

$$\text{CORBA} \Rightarrow \text{glue language} \Rightarrow \text{COM}$$

4. The LuaOrb System

LuaOrb is an implementation of the previous binding design that integrates the scripting language Lua [22, 13] to CORBA, COM, and Java. As we will see in this section, with LuaOrb, Lua acts as a unifying glue language: We can write Lua code that freely use and mix components from those three component systems.

A common use of Lua is as a tool to customize applications [28]. Its extension facilities allow it to adapt to diverse application domains, from CGI [21] to astronomical data analysis [39]. Its integration with componentware systems has already been explored in some CORBA based applications for network management [46], cooperative work [17, 12] and configuration management [45].

Lua integrates in its design data-description facilities, reflexive facilities, and familiar imperative constructs. On the "traditional" side, Lua is a procedural language with usual control structures (*whiles*, *ifs*, etc.), function definitions with parameters and local variables, and the like. On the less traditional side, Lua provides functions as first order values, closures, and dynamically created associative arrays (called *tables* in Lua) as a single, unifying data-structuring mechanism.

Through its reflexive facilities, Lua fulfills the requirements for the glue language of our dynamic language binding. Its reflexive feature that is fundamental for us is the *tag* mechanism. Each object in Lua can have an assigned tag. The use of tags allows changing the semantics of some operations. The new semantics can be specified by *tag methods*: Functions that are executed when events occur over a value with a specific tag. Lua defines many kinds of events: arithmetic operations, accesses to global variables, accesses to objects, illegal function calls, etc. For our binding, tag methods allow to redefine the meaning of method calls (and other operations) over proxies, and so the binding can be called to handle such operations.

As an example of the binding between Lua and Java, let us use the following Java class:

```
public class Calc {
  public
  float add (float a, float b)
  {
    return a+b;
  }
}
```

A proxy to an instance of this class can be created, in LuaOrb, with the following code:

```
-- calc is an instance of Calc.
calc = LuaOrb:javaNew("Calc")
```

Once you have created a proxy, the method `add` can be called exactly like any method invocation over Lua native objects:

```
print(calc:add(2,3))
```

Figure 4 gives a description of what really happens in the previous example. The method call `calc:add` is in the

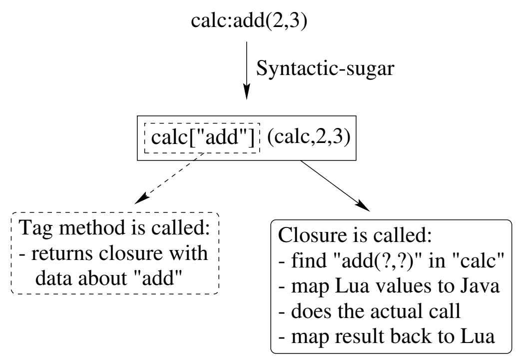

Figure 4. Calling a method through a proxy.

```
dispinterface IPasswd {
 methods:
 [id(0x00000001)]
 VARIANT_BOOL validate ([in] BSTR user,
                        [in] BSTR passwd);
};
```

```
interface login {
   boolean validate (in string user,
                     in string passwd);
};
```

Figure 5. Interfaces `IPasswd` and `login`.

top; then it is rewritten according to Lua's syntactic sugar. The first step, represented in the dashed box, occurs when the expression `calc["add"]` is evaluated. Because all proxy objects have a distinct tag, this expression creates a `"gettable"`[3] event, which calls the corresponding tag method. The tag method, which is part of the binding, executes the first two steps of the pattern algorithm previously described: First it gets the external component that the proxy represents; then it uses introspection to check whether this component has the desired method (or methods, in case of overloading). The final result is a closure encapsulating the information about the found methods.

The second step, represented in the solid box, happens when the closure is called with the given arguments; it executes the pattern algorithm from the third step to its end. In case of overloading, the closure selects, among the candidate methods, the best fitted to the given arguments. Then it converts the arguments to values in the component system, and builds a dynamic call. It does the actual call, and finally converts back any result, returning this converted result.

Although we have presented our example in Java, exactly the same pattern is used for CORBA and COM. A similar algorithm, also using tag methods and introspection, is used for field accesses.

Now let us see how LuaOrb handles the dynamic implementation of object servers, that is, Lua objects which can be called from the component system. This mechanism has one kind of implementation for CORBA and COM, and a different one for Java.

As we have seen in Section 2, CORBA provides an interface exactly for this purpose, called DSI (Dynamic Skeleton Interface). This interface has a single method, which is called to handle any operation performed over the object. In our binding, this method does the reverse of a Lua proxy: It translates CORBA arguments to Lua values, calls the corresponding Lua method, and translates back any results. Because the interface is fixed, we need only one class for this implementation; any Lua object, when exported to CORBA, uses an instance of this class for its interface.

COM classes also can be implemented through a unique interface, called `IDispatch`, with a method `invoke` that handles all operations over an object. Therefore, its binding is similar to the CORBA binding described above.

Java, on the other hand, does not provide such interface for dynamic implementations. We must use regular Java classes as proxies. Because of Java static type system, we must create a new class for each interface we want to export from Lua to Java. As we want a dynamic binding, such classes must be created on the fly. Whenever a new interface is needed, LuaOrb dynamically generates a class binary definition, and uses the `ClassLoader` to load it. All these generated classes have the same functionality: They delegate all its operations to their corresponding Lua objects, which are kept in an instance variable.

Note that the three solutions to implement our generic adaptors are based on delegating the method executions to Lua objects. Particularly, these Lua objects can be proxies to objects from a different component system. Therefore, we have bridges between different component systems as a direct result from our approach.

The following scripts give a better example of the integration facilities of LuaOrb. This example uses the Java package AWT to create a console, a COM object to validate logins in this console, and CORBA to link the Java client with the COM object across a distributed environment. The example also shows how Lua native objects, CORBA, COM and Java objects are all accessed in exactly the same way, after their creation.

The COM IDL interface to the COM object that validate logins is shown in Fig. 5. The figure also shows a OMG IDL interface compatible with that COM interface.

The first script of our example (Fig. 6) comprises the creation of a proxy for the COM object and the dynamic creation of a CORBA server linked to that proxy.

[3] Lua native objects are called *tables*.

```
-- create COM proxy for target service
com_obj=LuaOrb:COMproxy("Passwd.Passwd.1")
-- create CORBA server with 'com_obj'
-- as its implementation
server=LuaOrb:DSIServer("login",com_obj)
-- save server's IOR in file 'login.ref'
writeto("login.ref");
write(server:_get_ior());
writeto()
```

Figure 6. A script to bridge COM and CORBA.

The second script implements the client side of the example; it is shown in Fig. 7. The first lines create a proxy to the CORBA server that will do the login validation; the second block creates proxies to Java AWT beans needed for composing the user interface. The last block creates a new Lua object (`login_cb`) to implement the login button listener. When the method `addActionListener` is called, LuaOrb automatically creates a Java generic adaptor to link the Lua object `login_cb` with the related AWT button,

LuaOrb is not only a kind of protocol converter between different componentware systems, but it can also be used to adapt incompatible interfaces. For example, consider a different OMG IDL interface `login` (Fig. 8). Now, the interface `login` is incompatible with `IPasswd`. Nevertheless, we can easily change the server side script to adapt these interfaces, as we show in Fig. 9. The trick is to define a new `validate` method in Lua, which calls the old version, in COM, to do the real job.

5. Final Remarks

This work has addressed some issues related to the dynamic gluing of components. We have presented a design pattern for the implementation of bindings between glue languages and component systems. The key point in our design is its dynamic nature: Every step, from type checking to method identification and invocation, is done at run time. Therefore, a program can incorporate and integrate new components, from different component systems, without any additional declarations and without stopping for recompilation. Besides its dynamic characteristic, our binding design also allows the connection between components from different componentware systems.

We have also described the LuaOrb system, an implementation of the proposed language binding. With LuaOrb, Lua plays the role of a unifying glue language for Java, CORBA and COM. Although we have used Lua in our implementation, our design pattern can be used with other languages that offer similar reflexive facilities, such as

```
-- create CORBA proxy
login_serv=LuaOrb:CORBAproxy("login",
                             "login.ref")

--create other proxies to Java AWT beans
--needed for the user interface
t_user=LuaOrb:javaNew("java.awt.TextField")
t_pass=LuaOrb:javaNew("java.awt.TextField")
b_login=LuaOrb:javaNew("java.awt.Button",
                       "Login")
...

-- create a listener for the login button
login_cb = {}
function login_cb:actionPerformed(ev)
  local user=t_user.getText()
  local pass=t_pass.getText()
  local key=login_serv:validate(user,pass)
  if key then
    LuaOrb:javaNew("Application")
    frame:dispose()
  end
end

b_login:addActionListener(login_cb)
```

Figure 7. Lua script for the client side.

```
struct LoginInfo {
  string user;
  string password;
};

interface login {
  boolean validate(in LoginInfo data);
};
```

Figure 8. Another definition for `login`.

```
-- create COM proxy
com_obj=LuaOrb:COMproxy("Passwd.Passwd.1")
-- redefine method 'validate'
com_obj.old_validate = com_obj.validate
function com_obj:validate(info)
   return self:old_validate(info.user,
                            info.password)
end
-- create CORBA server
server=LuaOrb:DSIServer("login",com_obj)
```

Figure 9. Lua script for new `login` Interface.

Smalltalk and Tcl.

There are many available solutions for componentware scripting [4, 15, 32, 33, 10, 34, 16, 52]. However, few scripting languages support more than one component system, and even those languages do not provide a mechanism to mix components from different systems dynamically.

One of the main solutions to provide bridges between componentware systems is the OMG's COM/CORBA Interworking specification [42, 48]. This specification is similar to our solution in many aspects, such as the use of the COM `IDispatch` interface and the CORBA dynamic interfaces (DII and DSI) to implement the bridges. However, the OMG specification addresses the interworking between CORBA and both the static and dynamic mechanisms of COM (*Virtual Tables* and Automation, respectively), while our mechanism only deals with the dynamic mechanism (Automation). On the other hand, LuaOrb is not only a bridge mechanism; it is a gluing tool that can be used to adapt incompatible interfaces.

Although our bridge mechanism may create many indirection levels between the target and source component systems, it requires a minimal effort from the developer. From the prototyping point of view, this is a very important feature. Moreover, as usual with prototypes, we can replace the dynamic bridges by static ones where performance is critical.

Another important aspect of LuaOrb is that the bindings are external to the language, that is, the Lua package does not need any change to support the binding implementations: The bindings do all communication with Lua through Lua's standard API. This is not the case for some other solutions; for instance, the binding of Python to Java [54] was made through the reimplementation of Python in Java, while CorbaScript [33] was specifically implemented to bind to CORBA.

Currently, LuaOrb has its bindings in different stages of progress: Lua-CORBA and Lua-Java are fully implemented, with both the client proxies and server adaptors implemented. The client proxy of Lua-COM is working, although without a complete mapping between Lua types and COM types, and its server adaptor is very incipient. We are also working on the implementation of a type checking algorithm to verify the type compatibility between interfaces of those three componentware systems.

We are already using LuaOrb, mainly the CORBA binding, in some application domains. As a particular example, we are using LuaOrb as the basis for a network management platform [46]. The goal is to allow advanced users to extend existing management tools with new functionality, such as the management of new devices and the graphical display of statistical summaries of monitored data. The dynamic characteristic of LuaOrb ensures that accesses to new servers can be dynamically incorporated into applications.

In another application, we use Lua scripts to define and modify, at run-time, the relationships (*constraints*) among the objects of a cooperative CAD system [17, 12]. With this mechanism, the insertion of a new CAD object in the system and the dynamic constraint configuration are more flexible. Besides, each component in the cooperative system becomes less sensitive to changes in other components, since the application do not need to be recompiled neither stopped when a change is made in a Lua configuration script.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, second edition, 1998.

[2] B. Ban. *A Generic Management Model for CORBA, CMIP and SNMP*. PhD thesis, University of Zurich and IBM Zurich Research Laboratory, Rueschlikon, Switzerland, 1997.

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

[4] K. Brockschmidt. *Inside OLE*. Microsoft Press, second edition, 1995.

[5] R. Cerqueira, R. Ierusalimschy, and N. Rodriguez. A dynamic approach for composing CORBA objects. Monografias em Ciência da Computação 43/97, PUC-Rio, Rio de Janeiro, Brazil, 1997.

[6] D. Clark, editor. *OLE Automation Programmer's Reference*. Microsoft Press, 1996.

[7] I. Corporation, N. C. Corporation, O. Corporation, and Sunsoft. CORBA Component Imperatives. Technical Report orbos/97-05-25, OMG, 1997. `http://www.omg.org`.

[8] D. Cowan, R. Ierusalimschy, and T. Stepien. Programming environments for end-users. In *12th World Computer Congress*, volume 3, pages 54–60, Madrid, Sept. 1992. IFIP.

[9] V. de Mey. Visual composition of software applications. In O. Nierstrasz and D. Tsichritzis, editors, *Object-oriented Software Composition*. Prentice-Hall, 1995.

[10] L. d'Informatique Fondamentale de Lille and O.-O. Concepts. OMG CORBA Scripting Language RFP Initial Submission. Technical Report orbos/97-11-25, OMG, 1997. `http://www.omg.org`.

[11] ECMA. *EMCAScript: A general-purpose, cross-platform programming language*, June 1997. `http://www.omg.org`.

[12] B. Feijó, P. Rodacki, J. Bento, S. Scheer, and R. Cerqueira. Distributed agents supporting event-driven design processes. In *5th International Conference on Artificial Intelligence in Design'98*, Lisbon, Portugal, July 1998.

[13] L. H. Figueiredo, R. Ierusalimschy, and W. Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, Dec. 1996.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[15] B. Gates. VBA and COM. *Byte*, 23(3):70–72, Mar. 1998.

[16] GMD-FOKUS. CORBA Scripting Language OMG RFP Submission. Technical Report orbos/97-11-30, OMG, 1997. `http://www.omg.org`.

[17] P. Gomes, B. Feijó, R. Cerqueira, and R. Ierusalimschy. Reactivity and pro-activeness in virtual prototyping. In *2th International Symposium on Tools and Methods for Concurrent Engineering*, pages 242–253, Manchester, U.K., Apr. 1998.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[19] K. M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 62–69, Annapolis, Maryland, May 1996.

[20] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 70–80, Annapolis, Maryland, May 1996.

[21] A. Hester, R. Borges, and R. Ierusalimschy. Building flexible and extensible Web applications with Lua. *Journal of Universal Computer Science*, 4(9):748–762, 1998. http://medoc.springer.de:8000/.

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

[23] V. Issarny and C. Bidan. Aster: A CORBA-based software interconnection system supporting distributed system customization. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 194–201, Annapolis, Maryland, May 1996.

[24] JavaSoft. *JavaBeans, Version 1.00-A*, Dec. 1996. `http://java.sun.com/beans`.

[25] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[26] M. Kirtland. *Designing Component-Based Applications*. Microsoft Press, 1999.

[27] J. Kramer, J. Magee, and K. Ng. Graphical configuration programming. *IEEE Computer*, 22(10):53–65, Oct. 1989. (`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[28] Projects using Lua. http://www.tecgraf.puc-rio.br/lua/uses.html.

[29] M. Lutz. *Programming Python: Object-Oriented Scripting*. O'Reilly & Associates, 1996.

[30] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceeddings of the Fifth European Software Engineering Conference*, Barcelona, Spain, Sept. 1995.

[31] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 1(5):304–312, Sept. 1994.

[32] P. Merle, C. Gransart, and J.-M. Geib. CorbaScript and CorbaWeb: A generic object-oriented dynamic environment upon CORBA. In *Proceedings of TOOLS Europe'96*, Paris, France, Feb. 1996. Prentice-Hall.

[33] P. Merle, C. Gransart, and J.-M. Geib. Using and implementing CORBA objects with CorbaScript. Technical report, Universite de Lille, 1997.

[34] S. Microsystems and I. Technologies. Response to the CORBA Scripting Language Proposal. Technical Report orbos/97-11-16, OMG, 1997. `http://www.omg.org`.

[35] N. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building reconfiguration primitives into the law of a system. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 89–97, Annapolis, Maryland, May 1996.

[36] K. Ng and J. Kramer. Automated support for distributed software design. In *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering*, pages 381–390, Toronto, July 1995. (`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[37] K. Ng, J. Kramer, and J. Magee. A CASE tool for software architecture design. *Journal of Automated Software Engineering*, 3(3):261–284, Aug. 1996. (`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[38] K. Ng, J. Kramer, J. Magee, and N. Dulay. A visual approach to distributed programming. In A. Zaky and T. Lewis, editors, *Tools and Environments for Parallel and Distributed Systems*. Kluwer Academic Publishers, Feb. 1996. (`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[39] D. Nguyen, T. Gaetz, D. Jerius, and I. Stern. Modeling axaf obstructions with the generalized aperture program. *Astronomical Data Analysis Software and Systems VI, ASP Conference Series*, 125, 1997.

[40] OMG. CORBA Component Model — Request For Proposal. Technical Report orbos/96-06-12, OMG, 1996. `http://www.omg.org`.

[41] OMG. CORBA Scripting Language — Request For Proposal. Technical Report orbos/96-06-13, OMG, 1996. `http://www.omg.org`.

[42] OMG, Framingham, MA. *The Common Object Request Broker Architecture and Specification; Revision 2.2*, Feb. 1998. `http://www.omg.org`.

[43] J. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Mar. 1998.

[44] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[45] N. Rodriguez, R. Ierusalimschy, and R. Cerqueira. Dynamic configuration with CORBA components. In *4th International Conference on Configurable Distributed Systems*, Annapolis, 1998.

[46] N. Rodriguez, A. Moura, R. Ierusalimschy, and R. Cerqueira. Aplicações de gerência com comportamento dinâmico. In *Anais do SFBSID'97*, pages 476–487, Fortaleza, Brasil, 1997.

[47] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[48] M. Rosen and D. Curtis. *Integrating CORBA and COM Applications*. John Wiley & Sons, 1998.

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

[50] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.

[51] M. Stadel. Object oriented programming techniques to replace software components on the fly in a running program. *ACM SIGPLAN Notices*, 26(1):99–108, 1991.

[52] S. Stanton. Tclblend: Blending tcl and java. *Dr. Dobb's Journal*, pages 50–54, Feb. 1998.

[53] J. Thornton. Practical description of configurations for distributed systems management. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 36–43, Annapolis, Maryland, May 1996.

[54] G. van Rossum. Java and Python: a perfect couple. *developer.com*, Aug. 1998. http://www.developer.com/journal/techfocus/081798_jpython.html.

[55] I. Warren and I. Sommerville. A model for dynamic configuration which preserves application integrity. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 81–88, Annapolis, Maryland, May 1996.