Binding An Interpreted Language to CORBA

Renato Cerqueira Noemi Rodriguez
Roberto Ierusalimschy
Departamento de Informática — PUC/Rio
Rua M. S. Vicente 225, Gávea
22453-900 — Rio de Janeiro — RJ
rcerq,noemi,roberto@inf.puc-rio.br

Abstract

This paper describes LuaOrb, a binding between the extension language Lua and CORBA. Like other CORBA bindings, this binding allows a program to manipulate CORBA objects in the same way it manipulates local objects. Unlike conventional bindings, however, LuaOrb is based on the CORBA Dynamic Invocation Interface, mapping its dynamic character to the dynamic type system of Lua. This binding brings together CORBA reuse facilities with the flexibility provided by an interpreted language, fulfilling the "command center" concept proposed by Siegel.

Sumário

Este artigo descreve LuaOrb, uma interface entre a linguagem de extensão Lua e CORBA. Essa, como outras interfaces de CORBA, possibilita que um programa manipule objetos CORBA da mesma forma que manipula objetos locais. Entretanto, diferentemente de outras interfaces, LuaOrb é baseada na Interface Dinâmica de Chamadas de CORBA, mapeando seu carater dinâmico no sistema de tipos dinâmico de Lua. Essa interface une as facilidades de reuso de CORBA com a flexibilidade de uma linguagem interpretada, implementando o conceito de um "centro de comando", proposto por Siegel.

1 Introduction

Interpreted languages have an important role in the software development process. These languages are very useful for rapid prototyping, a technique which is being increasingly adopted in the development of new systems, mainly as a tool for requirement acquisition. Besides, the large demand for customizable applications has changed the typical structure of programs. Nowadays, many programs are split in two parts, a *kernel* and a *configuration*, written in two different languages. The kernel implements the basic classes and objects of the system, and is usually written in a compiled, statically typed language, like C or Modula-2. The configuration part, usually written in an interpreted, flexible language, connects these classes and objects to give the final shape to the application [CIS92, ?]. Such configuration languages are also called *extension languages*.

The concept of reusable modules or "componentware" is also becoming increasingly important in the development of software. It seems interesting, then, to explore the possibility of using componentware when prototyping with an extension language, thus saving time in the development of readily available program parts. It is also interesting to explore the use of configuration languages as a *gluing* factor in the construction of new programs from available modules. For this, it is necessary to have some kind of interface between interpreted languages and componentware facilities.

In the last few years, CORBA ([OMG95, Sie96]) has been gaining momentum as one of the main platforms for construction of componentware. Several implementations are now available, many of which include support for distributed applications. The CORBA proposal is strongly based on the notion of object interfaces, described in *IDL* (Interface Description Language). The development of mappings (called *bindings*) from IDL to different programming languages allows each client and server to be implemented in the most convenient language.

CORBA supports two types of client/server invocations: static and dynamic. Static invocations use stubs generated at compile time, from information in the IDL description of objects to be invoked. Dynamic invocations use the Dynamic Invocation Interface (DII), which allows a client to build and invoke requests to servers at run time.

Information about objects and their interfaces may be obtained dynamically through calls to the *interface repository* (IR), which maintains a description of interfaces of available servers in machine readable form. The commonly discussed use of the interface repository (see, for instance, [OHE96]) is in tools like class browsers, application generators, or compilers. Such tools use the interface repository to create method invocations through stubs which are then compiled and linked to the target application. In this context, interpreted languages can add a great deal of flexibility. Using the Interface Repository and DII, one can dynamically compose method calls, with dynamic typing. This makes interpreted languages specially attractive for the gluing function mentioned above, since it means that applications can be dynamically and transparently configured and reconfigured. It also fulfills the "command center" concept proposed by Siegel in [Sie96]: A console where a user would be able to freely interact with any CORBA object server. Siegel also states that there will be a new standard for state-of-the-art desktop software when client applications find out and incorporate new objects via DII [Sie96].

In this paper, we describe LuaOrb, a binding between the extension language Lua [IdFC96, FIC96] and CORBA. As usual, this binding defines mappings between Lua and IDL types. Unlike other bindings, this mapping is based on the dynamic typing of Lua and the Dynamic Invocation Interface of CORBA, without the use of stubs. With this system, when a programmer wants to invoke a method on a CORBA Object, she just writes the call, using the regular syntax for method calls in Lua. LuaOrb intercepts the call, dynamically maps parameter types from Lua to IDL, does the actual invocation, and maps any results back to Lua. With LuaOrb, the user has transparent access to all available CORBA server objects, as if they were local objects. There is no need to predefine IDL headers and stubs for each

```
function map (a, f)
  local b = {}  -- creates a new table
  local i = 1
  while a[i] do
    b[i] = f(a[i])
    i = i+1
  end
  return b
end
```

Figure 1: An example of Lua code.

object.

As of this moment, standards have been established for bindings between CORBA and C, C++, and Smalltalk. It is not at all clear, at this point, whether the mapping of different languages to the CORBA IDL is really viable for a wide range of programming languages [CI96]. The definition of a binding for a language such as Lua, in many ways very different from the three languages mentioned above, is an important step towards the evaluation of the effective integrating potential of CORBA.

2 The Extension Language Lua

Lua is a general purpose configuration language that arose from the need of the TeCGraf Group[1] to use a single extension language to customize industrial applications [IdFC96]. Currently, Lua is being used in dozens of products and prototypes, many of which developed outside TeCGraf. 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, and dynamically created associative arrays (called *tables* in Lua) as a single, unifying data-structuring mechanism.

As a simple example of Lua code, Figure 1 shows the implementation of the `map` function in Lua; this function receives a list $a = (a_1, a_2, \ldots, a_n)$ (represented by a table with indexes 1, 2, ..., n) and a function f, and returns a new list $b = (f(a_1), f(a_2), \ldots, f(a_n))$.

There is no notion of a "main" program in Lua; being an embedded language, it only works embedded in a host client. Lua is provided as a library of C functions to be linked to host applications. The host can invoke functions in the library to execute a piece of code in Lua, write and read Lua variables, and register C functions to be called by Lua code. Moreover, *fallbacks* can be specified to be called whenever Lua does not know how to proceed. In this way, Lua can be augmented to cope with

[1]TeCGraf is a research and development group at PUC-Rio, with industrial partners like PETROBRAS and CEPEL.

rather different domains, thus creating customized programming languages sharing a single syntactical framework [Bec91].

Functions in Lua are *first class* values, that is, functions can be manipulated like any other value in the language. A function definition creates a value of type `function`, and assigns this value to a global variable (`map`, in Figure 1). Like any other value, function values can be stored in variables, passed as arguments to other functions, or returned as results. This feature greatly simplifies the implementation of object-oriented facilities.

Lua is dynamically typed. Variables can handle values of any type. Whenever an operation is performed, it checks the correctness of its argument types. Besides the basic types `number` (floats) and `string`, and the type `function`, Lua provides three other data types: `nil`, `userdata`, and `table`.

The type `nil` has a single value, also called `nil`, whose main property is to be different from any other value. The type `userdata` is provided to allow arbitrary host data to be stored in Lua variables. The only valid operations on values of this type are assignment and equality test. Typically, userdata values are used to store C pointers created by C functions, which can be passed back to other C functions, behaving as an abstract data type in Lua.

Finally, the type `table` implements associative arrays, that is, arrays that can be indexed not only by integers, but by strings, reals, tables, and function values. Associative arrays are a powerful language construct; many algorithms are simplified to the point of triviality because the required data structures and algorithms for searching them are implicitly provided by the language [Ben88]. Most typical data containers, like ordinary arrays, sets, bags, and symbol tables, can be directly implemented by tables. Tables can also implement records by simply using field names as indices. Lua supports this representation by providing `a.name` as syntactic sugar for `a["name"]`.

Tables are created with special expressions, called *constructors*. The simplest constructor is the expression `{}`, which returns a new empty table. An expression like `{n1 = exp1, n2 = exp2, ...}` creates a new table, and stores in each field `ni` the result of `expi`. A typical example is the creation of a point:

```
point1 = {x = 10, y = 30}
```

Constructors can also be used to built lists, with the syntax `{exp1, exp2, ...}`. This expression creates a new table, and stores in each field `i` the result of `expi`. Therefore, after the assignment:

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"}
```

the expression `days[3]` will result in the string `"Tuesday"`.

Because functions are first class values, table fields can refer to functions. This property allows the implementation of some interesting object-oriented facilities, which are made easier by syntactic sugar. First, method definitions can be written like

```
function object:foo (params)
  ...
end
```

which is equivalent to

```
function object.foo (self, params)
  ...
end
```

That is, a function is created and stored in a table field[2]; moreover, this function has a hidden parameter called `self`. Second, a method call can be written as

```
receiver:foo(params)
```

which is translated to

```
receiver.foo(receiver, params)
```

In other words, the receiver of the method is passed as its first argument, giving the expected meaning to the parameter `self`. A complete treatment of Lua's object-oriented facilities is outside the scope of this paper; the interested reader should refer to [IdFC96].

Several features in Lua have a row in its reflexive character. For instance, since records and objects are represented by tables, it is easy to "browse" an object at run-time, traversing all of its fields or simply checking the presence of particular fields. Whenever explicit type checking is needed, the primitive function `type` may be used; it returns a string describing the type of its argument.

Part of the reflexive nature of Lua is provided by *fallbacks*. Fallbacks allow programmers to set their own functions to handle what Lua considers error conditions. Fallbacks are also used to provide hooks to handle other situations that are not strictly error conditions, such as accessing an absent field in a table and signaling garbage collection. Although the semantics of fallbacks are quite similar to exception handling with resumption [YB85], the dynamic nature of Lua allows their use in many cases where a statically typed language would issue an error at compile time. Therefore, despite the semantic similarities, the use of fallbacks in Lua is quite different from the common use of exception handling. Fallbacks have an efficient implementation, and are frequently used to emulate other facilities, like inheritance and overloading [IdFC96].

The use of fallbacks in an otherwise "type error" construction is the key to some interesting facilities. As a typical example, consider the following statement:

```
obj:foo(a, b)
```

As explained before, this is equivalent to:

```
obj["foo"](obj, a, b)
```

Usually, `obj` would be a table, and so `obj["foo"]` would be the field `"foo"` of this table. However, `obj` does not need to be a table; it can be a userdata, for instance. In such cases, a user defined fallback will be called, and this fallback will be responsible for providing a sensible value to the expression `obj["foo"]` (or for raising an error). If this value is a function, this function will then be called with the parameters `obj`, `a` and `b`. Otherwise, another fallback will be called, to handle the "invocation" of this non function value.

[2]Remember that `object.foo` is equivalent to `object["foo"]`.

3 CORBA and Its Dynamic Invocation Interface

The CORBA standard [OMG95] provides communication facilities to applications in a distributed environment. All communication between CORBA objects is mediated by the *Object Request Broker* (ORB) [OMG95, Sie96]. A client can interact with the broker through *IDL stubs* or through its *Dynamic Invocation Interface*.

OMG IDL is a technology-independent syntax for describing object interfaces. Specifications written in IDL can be compiled into stubs (in the client side) and skeleton programs (in the server side). A client program links directly to the IDL stub. From the client's perspective, the stub acts as a local object. Transparently, the stub provides an interface to the ORB, which performs marshaling to encode and decode the operation parameters into communication formats suitable for transmission. The IDL skeleton program is the corresponding server-side implementation of the IDL interface. When the ORB receives a request, the skeleton provides a callback to a server-supplied method implementation. When the server completes the process of the request, the skeleton passes the results back to the client stub, which then returns the results to the client program, along with any exception information. It is worth noting the similarities between this model and the RPC model [Sun88].

The Dynamic Invocation Interface (DII) is a generic facility for invoking any operation with a runtime-defined parameter list. A dynamic invocation performs four steps:

- identify the desired object;
- retrieve its interface (type);
- construct the invocation;
- invoke the request, and receive the results.

Figure 2 shows an example of interaction among CORBA objects in a dynamic invocation. Each of these steps is analyzed in more details below.

In a dynamic distributed object environment, users can consult a *Trading Object Server* or a *Naming Server* [Sie96] to locate object implementations.

The Naming Service, as its name suggests, links names to CORBA object references in a distributed scope. The IDL interface that defines the operations of this service is called `NamingContext`. The main methods of this interface are `bind` and `resolve`. Method `bind` associates a name to an object reference, while `resolve` returns the object that is associated to an specific name.

The Trading Object Service supports the offer and discovery of instances of objects of particular types. This service is implemented by objects called *traders*. Traders can be viewed as objects through which other objects can advertise their capabilities and match their needs against advertised capabilities.

Advertising a capability or offering a service is called an *export*. Matching against needs or discovering services is called an *import*. To export, an object gives the trader a description of a service and the location of an interface where that service is available. To import, an object asks the trader for a service having certain characteristics. The trader checks against the service descriptions it holds and responds to the importer with the location of the selected object's interface.

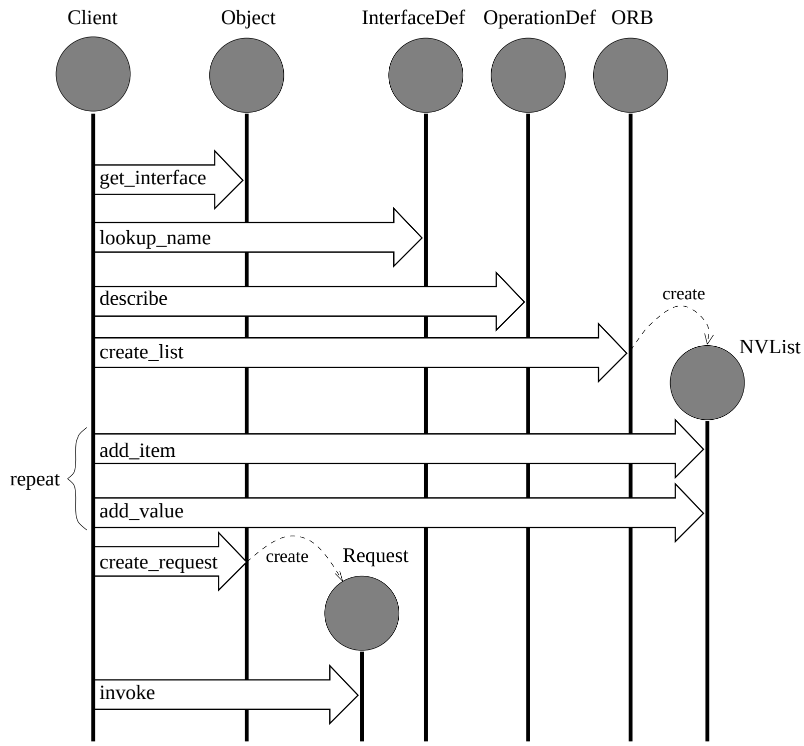

Figure 2: An Example of Dynamic Invocation.

Examples of the additional information that may describe object services are features, location, cost, etc. Since the Trading Object Service has been standardized recently, there are few implementations of it.

Once the reference for a desired object is known, it is possible to acquire the meta data that describe the object interface. CORBA offers many reflexive mechanisms, where objects provide information about themselves. In the example of Figure 2, method `get_interface`, which is offered by every CORBA object, is used to access an object interface description. This description is represented by an `InterfaceDef` object, returned by method `get_interface`. The `InterfaceDef` object is an entry-point for navigating the Interface Repository. In this manner, it is possible to obtain detailed information about an object interface. CORBA specifies about ten calls for navigating the Interface Repository and describing the objects it contains.

With a complete method definition, it is possible to create the argument list to be used in the method call. CORBA specifies a data structure for passing parameters, called *Named Value List*, which is a list of pairs of names and values. Values, in turn, are unions containing a type description an the actual value.

A request in DII is represented by a CORBA *pseudo-object*[3] that contains the name of the method, the argument named value list, and a place for the return value. A `Request` object is created by invoking the ORB method `create_request`.

The invocation of a `Request` object can be performed in three different ways:

- the `invoke` method call sends the request and obtains the results (*synchronous* mode);

[3]In OMG terminology, a *pseudo-object* is an object that is created directly by the ORB.

- the `send_oneway` method call sends a datagram. In this case, no response is needed (*asynchronous* mode);
- the `send_deferred` method call sends the request and returns the control to the program, which must then poll for the response by calling `poll_response` or `get_response` (*deferred synchronous* mode).

From the server's point of view, the use of DII instead of an IDL static interface is completely transparent.

Dynamic Invocation versus Static Stubs

There are a number of important differences between static interfaces and DII. The major one is that DII allows the discovery of the object type and methods at run time, while static interfaces require the definition of the object type to be made at compile time. In other words, DII allows *dynamic typing*.

There are additional differences. As a consequence of dynamic typing, DII can not check argument type correctness at compile time, while static interfaces can. Structurally, the ORB requires a separate stub for each static interface, but only one DII interface is provided by ORB itself. While static interface invocations are generally synchronous, or at most asynchronous, DII may be invoked in either synchronous, asynchronous, or deferred synchronous modes.

With traditional bindings, programming with static interfaces is much simpler and results in more robust code for the developer. However, DII provides a level of flexibility that is necessary in some applications, such as desktops and operating systems [MZ95]. In fact, the stubs used in the static interface use a low-level variant of DII to do their jobs [Vis96].

Some CORBA application builders prefer DII because, unlike static interfaces, it does not create a new stub for each interface a client uses. This allows a smaller ORB memory footprint, although the potential exists for a run-time performance penalty since at least first-time invocations of each interface are, in effect, interpreted at run time by the client [Sie96].

In practice, it is not so easy to make applications incorporate new objects via DII. DII offers the support for applications to access new object types, and the Naming and Trader services address the problem of finding out new objects on the system. However, there is still the difficulty of building the method calls. Using DII, this activity involves querying and constructing complex structures.

4 The Binding between CORBA and Lua

The definition of the mapping between Lua and CORBA has tried to respect the common use of Lua. That is, CORBA objects are to be used in the same way as other Lua objects. To reach this goal, the reflexive characteristics of Lua were fully applied.

Since Lua works as an embedded language, its binding to CORBA is more relevant for clients (applications) than for servers. Therefore, this Lua binding addresses only the issues related to the client side. The reason for this approach is that the

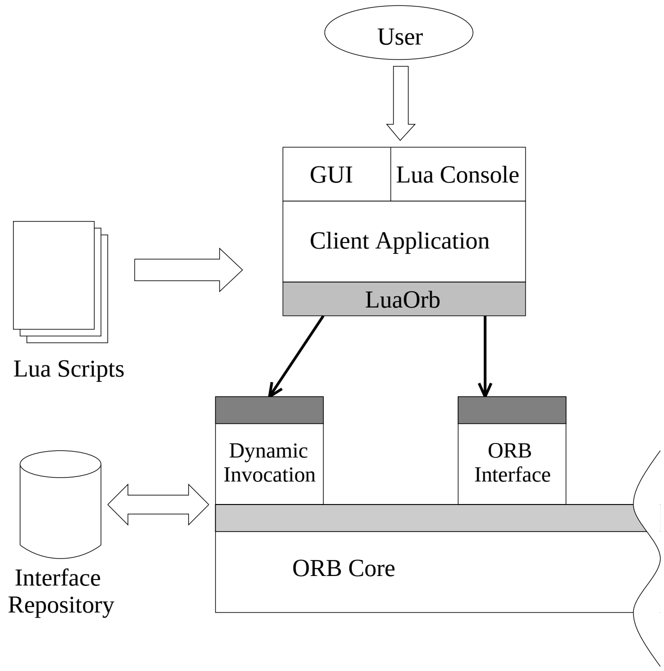

Figure 3: The architecture of a LuaOrb client application.

generation of CORBA skeletons in Lua makes little sense (although the use of Lua to implement the methods of a server object is another interesting topic).

Since CORBA objects are to be accessed like any other Lua object, the generation of stubs was neither necessary nor interesting. Instead, CORBA objects should be accessible from Lua with no need of previous declarations, and with dynamic typing. To achieve this goal, the binding was built upon DII.

The Lua binding intercepts a method call, dynamically maps parameters types from Lua to IDL, does the actual invocation, and maps any results back to Lua. The mapping of parameter types is done by trying to coerce Lua values into IDL types, and vice versa for result types. The binding specifies which coercions can be performed. It is interesting to notice that this mapping is done between two dynamic type descriptions: The actual types of Lua arguments, accessed through Lua API, and the formal types of the method parameters, accessed through the Interface Repository.

The Lua binding is implemented in a library called `LuaOrb`. The resulting architecture for integrating LuaOrb library with the application and with the ORB is illustrated by Figure 3.

The next two subsections give more details about the LuaOrb implementation.

4.1 How to Use CORBA Objects

To explain the way Lua uses CORBA objects, the example of an IDL interface in Figure 4 will be used. The methods of this example were chosen so as to illustrate the mapping between some IDL data types and Lua types.

To create a proxy of a distributed object that implements the `foo` interface, the function `createproxy` is used in the following way:

```
struct book
{
   string author;
   string title;
};

interface foo
{
   boolean add_book(in book book_info);
   boolean test();
   long div(in long x, in long y);
};
```

Figure 4: An example of an IDL interface.

```
a_foo = createproxy("foo")
other_foo = createproxy("foo","Master Foo")
```

As the above example shows, `createproxy` can receive two parameters: the first one (required) is the name of the interface that should be created; the second one (optional) is the name of a specific instance of the interface specified in the first parameter. If the second parameter is not specified, the ORB tries to find some server of the desired interface that has been previously activated. If it does not find any active server, it tries to activate one (in order to do this, the ORB must have a registered server for the interface). If, in some way, the ORB finds an object for the desired interface, then a proxy for this object is created, and an identifier of this object is returned to the application. Otherwise, the function returns `nil`.

When the second parameter is specified, `createproxy` will only succeed if there is a server object with the same name of this parameter.

After a proxy has been created, the services of the related object can be requested. For example, the methods of the `foo` interface can be called as following:

```
a_book = {author = "Mr. X", title = "A New Book"}
a_foo:add_book(a_book)
if a_foo:test() then
  x = other_foo:div(y,z)
end
```

The above example shows how a Lua user handles CORBA objects in the same way she handles Lua objects. Notice that the Lua table `a_book` is dynamically mapped to the IDL structure `book`, when `add_book` is called. So, the Lua table must have, at least, the same fields that the IDL structure.

4.2 The Binding Implementation

Two mechanisms must be supplied by the binding implementation: The creation of proxy objects, and the invocation of methods of CORBA objects.

As explained before, proxies are created by calls to function `createproxy`. The implementation of this function is basically a direct mapping to its equivalent function in the ORB API. The returned value of `createproxy` is a Lua `userdata`, which is a C++ reference to a proxy.

The following code fragment will be used to describe the implementation of method calls:

```
a_book = {author = "Mr. X", title = "A New Book"}
a_foo = createproxy("foo")
result = a_foo:add_book(a_book)
```

As explained in Section 2, the expression `a_foo:add_book(a_book)` is equivalent to:

```
a_foo["add_book"](a_foo, a_book)
```

Since `a_foo` is not a table, the expression `a_foo["add_book"]` triggers a fallback. The fallback function, defined by LuaOrb, simply returns the string `"add_book"`, which will be used by Lua as the "function" to be called. When Lua tries to call this string, another fallback is called, as explained in Section 2. This second fallback receives as parameters the string `"add_book"` (the method name), the userdata `a_foo` (the proxy reference), and the table `a_book`.

With this information, the LuaOrb fallback asks the Interface Repository whether the IDL interface `foo` has a method named `add_book`. If this is the case, the description of `add_book` (parameters, context information, result and exceptions) is retrieved from the IR.

With the Lua parameters and the method description at hand, the parameter list for the dynamic call is built. The Lua parameters are adjusted to the types of the IDL description (if this conversion is not possible, a run-time error is generated). In our example, the Lua table `a_book` is converted into a `NVList` that describes the structure `book`.

After that, the method `add_book` is invoked, in accordance with its invocation semantics (whether it is `oneway` or not). After the method execution, LuaOrb checks whether some exception was raised. If there are no exceptions, the method's returned value and result parameters are converted to Lua values and returned to the client application. Otherwise, the exception is raised.

A more detailed description of the binding can be found in [CRI97].

5 Related Work

To the best of our knowledge, currently no CORBA binding adopts the solution proposed by LuaOrb, of mapping the dynamic character of an interpreted, dynamic typed language to the dynamic characteristics of the Dynamic Invocation Interface.

Most CORBA language bindings, including the bindings for C, C++, Smalltalk [OMG95], Java [Jav96] and School [Cer96], are strongly based on the static invocation interface. Although some of them also support access to DII, this support is in a very low abstraction level. Indeed, since most of these languages are statically typed (the only exception being Smalltalk), it is inherently difficult to bind DII facilities to higher level language constructs.

The Smalltalk binding makes use of the dynamic characteristics of the language [GR83] to generate stubs at run time. This feature makes Smalltalk clients more flexible than clients based in static compiled stubs. Nevertheless, the binding is still based on static IDL descriptions and stubs.

Another CORBA binding to a non statically typed language is TclDii [ASG+94]. Like LuaOrb, TclDii is built on top of DII, and it extends the functionality of TCL by enabling TCL scripts to send requests to CORBA objects. Distinctly from LuaOrb, however, TclDii obliges the client programmer to know many details of DII, as the use of DII is not transparent in TclDii. For instance, the structures that are used by DII must be explicitly built in TclDii, in the same way that in the C binding for DII.

6 Final Remarks

The LuaOrb library is fully implemented over *Visibroker for C++* [Vis96], a CORBA 2.0 compliant implementation. At this moment, LuaOrb has been tested in some small applications.

From the work done so far, some results stand out. The first result is a new approach to the implementation of language bindings to the CORBA object model, using the Dynamic Invocation Interface to construct the binding. Our model can be adopted by other languages that support reflexivity and dynamic typing.

The Lua binding can be seen as an evaluation of the effective integrating potential of CORBA, since it is in many ways quite different from the known bindings. From this point of view, this work is a continuation of the work started in [Cer96].

On the other hand, this work has addressed some problems related to DII. As discussed in [Sie96, MM97], the adoption of DII can give rise to a new generation of distributed applications, but there are some problems with the use of DII. One of them is the complexity of building requests. LuaOrb solves this problem, by raising the abstraction level of DII to the same level of common method calls. Orfali and Harkey [OH97] state other problems related to DII, such as a weak type checking and a run-time performance penalty. The weak type checking problem is solved by LuaOrb by its automatic mapping between Lua and IDL types. Of course, there is still the risk of run-time type errors, since we use dynamic typing.

The run-time performance penalty can be greatly reduced with the minimization of the number of queries to the Interface Repository. This can be achieved through several mechanisms, for example caching the most recently queries to the Interface Repository, or bringing the complete interface description from the IR when the first instance of a specific class (IDL interface) is created. In the current implementation, the first time a method is called its interface description is loaded by the client, and used in any subsequent call. If the method interface changes in the server, the call generates an error, and LuaOrb automatically reloads the new interface. This implementation gives a performance penalty of less than 50% for a remote call, compared to a call using a static pre-compiled stub.

With the support offered by LuaOrb, it has been possible to fulfill *the command center* concept proposed by Siegel [Sie96]. In accordance with this author, the client applications will become generic command centers that proceed to configure themselves to invoke all the actions their users want via DII. This is exactly what is provided by a Lua console when embedded in an application using LuaOrb. Siegel also states that this application architecture is a necessary step in the transition from monolithic applications to user-centric distributed computing.

A most important continuation of this work is the development of some case studies about the use of LuaOrb. These case studies will then be used to evaluate the power of the mechanism. We believe that LuaOrb can be used basically in two scenarios. In the first one, new applications are assembled from the reuse of available components, with Lua performing the role of a *Composition Language* [NM94, NT95] (in this way, the binding of Lua to OLE automation [Cla96] and DCOM [Bro95] technologies is another important issue). In the second scenario, the client application downloads a Lua program (like an Internet browser) that accesses server objects unknown by the client.

Acknowledgements

This work has been partially supported by PETROBRAS, CNPq and CAPES.

References

[ASG+94] G. Almasi, A. Suvaiala, C. Goina, C. Cascaval, and V. Jagannathan. TclDii: A TCL interface to the Orbix Dynamic Invocation Interface. http://www.cerc.wvu.edu/dice/iss/TclDii/TclDii.html, 1994.

[Bec91] B. Beckman. A scheme for little languages in interactive graphics. *Software, Practice & Experience*, 21:187–207, 1991.

[Ben88] J. Bentley. *More programming pearls.* Addison-Wesley, 1988.

[Bro95] Kraig Brockschmidt. *Inside OLE.* Microsoft Press, second edition, 1995.

[Cer96] Renato Cerqueira. Um estudo sobre interoperabilidade entre linguagens orientadas a objetos. Master's thesis, Dep. Informática, PUC-Rio, Rio de Janeiro, Brazil, 1996.

[CI96] Renato Cerqueira and Roberto Ierusalimschy. Uma avaliação das arquiteturas para interoperabilidade entre objetos. In *X Simpósio Brasileiro de Engenharia de Software*, pages 371–386, 1996.

[CIS92] D. Cowan, R. Ierusalimschy, and T. Stepien. Programming environments for end-users. In *12th World Computer Congress*, volume 3, pages 54–60, Madrid, Sep 1992. IFIP.

[Cla96] David Clark, editor. *OLE Automation Programmer's Reference.* Microsoft Press, 1996.

[CRI97] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. Using Lua to access CORBA objects. Monografias em ciência da computação, PUC-Rio, Rio de Janeiro, Brazil, 1997.

[FIC96] Luiz H. Figueiredo, Roberto Ierusalimschy, and Waldemar Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, December 1996.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and its Implementation.* Addison-Wesley, 1983.

[IdFC96] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Jav96] JavaSoft Inc. *Mapping IDL to Java*, feb 1996.

[MM97] Thomas Mowbray and Raphael Malveau. *CORBA Design Patterns.* John Wiley & Sons, 1997.

[MZ95] Thomas Mowbray and Ron Zahavi. *The Essential CORBA.* John Wiley & Sons, 1995.

[NM94] Oscar Nierstrasz and Theo Meijler. Requirements for a composition language. In *Proceedings of the ECOOP 94 workshop on Models and Languages for Coordination of Parallelism and Distribution*, 1994.

[NT95] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-oriented Software Composition.* Prentice-Hall, 1995.

[OH97] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA.* John Wiley & Sons, 1997.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide.* Wiley, 1996.

[OMG95] Object Management Group, Inc., Framingham, MA. *The Common Object Request Broker Architecture and Specification; Revision 2.0*, jul 1995.

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley & Sons, 1996.

[Sun88] Sun Microsystems. *RPC: Remote Procedure Call, Protocol Specification, Version 2*, jun 1988. RFC 1057.

[Vis96] Visigenic Software, Inc. *VisiBroker for C++ Programmer's Guide, Version 2.0*, 1996. http://www.visigenic.com/techpubs/vbrokerc++/vbcpgm2.pdf.

[YB85] S. Yemini and D. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), April 1985.