Using Reflexivity to Interface with CORBA

Roberto Ierusalimschy Renato Cerqueira Noemi Rodriguez

Departamento de Informática — PUC/Rio
22453-900 — Rio de Janeiro — Brazil
roberto,rcerq,noemi@inf.puc-rio.br

Abstract

Most bindings to CORBA are based on the construction of stubs, which translate a language call into a CORBA invocation. This paper shows an alternative way to build a binding, using the reflexive facilities of an interpreted language. 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, it is based on the CORBA Dynamic Invocation Interface, mapping its dynamic character to the dynamic type system of the language. In this way, a program has immediate access to any CORBA component, without the need of stubs or pre-defined IDL headers.

1 Introduction

The concept of reusable components (or "componentware") is becoming increasingly important in the development of software. In the last few years, CORBA ([15, 18]) has been gaining momentum as one of the main standards to interface components. Several CORBA 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.

Currently, all standard bindings between CORBA and programming languages (C, C++, Java and Smalltalk) are based on *stubs*. In this approach, each object interface a program uses must be compiled into a stub, which is then linked to the program. Whenever the program "calls" a method from a CORBA object, it in fact calls a stub function. This function marshals the parameters, does the actual CORBA invocation, and then unmarshals the results, converting them to appropriate data types in the host language.

In this paper, we describe LuaOrb, a binding between the extension language Lua [10, 6] and CORBA. As usual, this binding defines mappings between Lua and IDL types. Unlike other bindings, however, this mapping is based on the dynamic nature of Lua and the Dynamic Invocation Interface of CORBA, without the use of stubs. With this binding, 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 object. Also, nothing needs to be done when new object types are added to the system, or when an object interface changes in some compatible way.

To achieve this flexibility, LuaOrb uses two special features of CORBA: the Interface Repository (IR), and the Dynamic Invocation Interface (DII). The Interface Repository, which maintains a description of interfaces of available server objects in machine readable form, is automatically queried by LuaOrb for each new method invocation. With this information, the system checks type compatibility, and then uses the Dynamic Invocation Interface to build and execute the actual call.

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. It seems interesting, then, to explore the possibility of using components when prototyping with an interpreted language, thus saving time in the development of readily available program parts. It is also interesting to explore the use of interpreted languages as a *gluing* factor in the construction of new programs from available modules. LuaOrb is specially attractive for this gluing function, since it allows applications to be dynamically and transparently configured and reconfigured. It also

fulfills the "command center" concept proposed by Siegel in [18]: A console where a user would be able to freely interact with any CORBA object server.

The next section gives an overview of the programming language Lua, with some emphasis on its reflexive facilities. Section 3 covers some relevant aspects of CORBA. Then, Section 4 explains how the reflexive facilities of Lua have been used to implement LuaOrb. Related work is discussed in Section 5, and the last section is reserved for some final remarks.

2 The extension language Lua

Lua is a generic configuration language that integrates strong data description facilities and reflexivity with a simple syntax. Currently, Lua is being used in more than a hundred products and prototypes, in many academic institutions and companies. On the "traditional" side, Lua is a procedural language with the usual control structures (*whiles*, *ifs*, etc.), function definitions with parameters and local variables, and the like. On the less conventional side, Lua provides functions as first order values, and dynamically created associative arrays (called *tables* in Lua) as a single, unifying data-structuring mechanism.

There is no notion of a "main" program in Lua; Lua is an *embedded* language, and it only works embedded in a host client. Lua is provided as a library of C functions to be linked to host applications. The library API provides functions 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 rather different domains.

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`, Lua provides four other data types: `nil`, `userdata`, `functions` 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.

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. Like any other value, function values can be stored in variables, passed as arguments to other functions, or returned as results.

Finally, the type `table` implements associative arrays,

```
1  function clone (t)
2    local new = {}   -- new table
3    local i, v = next(t, nil)
4    while i do
5      new[i] = v
6      i, v = next(t, i)
7    end
8    return new
9  end
```

Figure 1. An example of Lua code.

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 [3]. 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"]`.

As an example of Lua code, Figure 1 shows the implementation of a `clone` function in Lua; this function receives a table `t` and returns a shallow clone of `t`. This piece of code shows some interesting features of Lua. Comments are written like in Ada, starting with `--`. The expression `{}`, which appears in line 2, creates a new empty table. In line 3 (and again in line 6), function `next` is used to traverse the indices of table `t`. This function takes two arguments: a table to be traversed and a "previous" index on this table. It then returns a "next" index, in an arbitrary order, and its value (it is worth noticing that functions in Lua may return multiple values). Therefore, in the loop, the local variable `i` runs over the indices of `t`, and `v` runs over their values. These values are then assigned to the new table at their respective indices (line 5).

Since records and other structures are represented in Lua by tables, function `next` provides an easy way to browse over these structures, traversing all its fields. Whenever explicit type checking is needed, the primitive function `type` may be used; it returns a string describing the type of its argument.

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. Method definitions can be written like

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

which is equivalent to

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

That is, a function is created and stored in table `object`, indexed by the string `"foo"`. Moreover, this function has a hidden parameter called `self`. A method call can be written as

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

which is translated to

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

In other words, the function is fetched from the table, and the table itself is passed as the first argument to the function, giving the expected meaning to the parameter `self`. A complete treatment of Lua's object-oriented facilities is beyond the scope of this paper; the interested reader should refer to [10].

Part of the reflexive nature of Lua is provided by *fallbacks*. Fallbacks allow programmers to change the "normal" behavior of Lua, especially in abnormal conditions, such as accessing an absent field in a table or calling a non-function value. Although the semantics of fallbacks are quite similar to exception handling with resumption [9], 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 [10].

Figure 2 shows how inheritance can be emulated with fallbacks. In this form of inheritance (or delegation), whenever a field is not present in an object `o`, its value is inherited

```
1  function Inherit (o, field)
2    if field == "parent" then
3      return nil
4    end
5    local p = o.parent
6    if type(p) == "table" then
7      return p[field]
8    else
9      return nil
10   end
11 end
12
13 setfallback("index", Inherit)
```

Figure 2. Using fallbacks to implement inheritance.

from another object `p`, the *parent* of `o`. The link between an object and its parent is stored in a field `parent` in the object itself.

Referring back to Figure 2, the last line sets the `"index"` fallback to be function `Inherit`. After that, whenever Lua attempts to access a field that is absent in a table, the fallback mechanism will call `Inherit`. This function then checks whether the object has a parent which is a table (line 6). If so, it accesses the desired field in that table (line 7), which may invoke a recursive `index` fallback, until a value for the field is found or the parent chain ends. (The need for the test in line 2 is that, when the parent field is accessed in line 5, it may be absent, and this may cause another fallback call.)

The above scheme allows endless variations. For instance, only methods could be inherited, or only fields starting with an underscore. Many forms of multiple inheritance can also be implemented.

The use of fallbacks in an otherwise "type error" construction is the key to other 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 may be called, and this fallback will be responsible for providing a sensible value for 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).

3 CORBA

The CORBA standard [15] provides communication facilities for applications in a distributed environment. All communication between CORBA objects is mediated by the *Object Request Broker* (ORB) [15, 18]. A client can interact with the broker through *IDL stubs* or through its *Dynamic Invocation Interface*.

IDL (Interface Description Language) is a technology-independent syntax for describing object interfaces. Specifications written in IDL can be compiled into client stubs (called simply stubs) and server stubs (called skeleton programs). The client program is directly linked to the IDL stub. From the client's perspective, the stub acts as a local object. Transparently, the stub marshals and unmarshals the operation parameters (converts them to and from formats suitable for transmission), acting as an interface to the ORB. 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 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 [4, 19].

One of the main points of using IDL is that the same interface can be compiled to different languages. Therefore, objects written in different languages may coexist in the same system, interacting and exchanging information among them. To allow such interoperability, each target language must define a mapping between IDL interfaces and its own types.

The Dynamic Invocation Interface (DII) is a generic facility for invoking any operation with a runtime-defined parameter list. A dynamic invocation performs the following steps: identifies the receiver object; retrieves its interface (type); builds the invocation request; invokes the request, and then receives the results. Figure 3 shows an example of interaction among CORBA objects in a dynamic invocation. Each of these steps is analyzed in more details below, using C++ as the working language.

Once the reference for an object is known, it is possible to acquire the meta data that describes the object interface. CORBA offers many reflexive mechanisms, where objects provide information about themselves. The source of this meta data is the Interface Repository (IR). The IR is a CORBA object that stores interface descriptions of other CORBA servers, in machine readable form. This object can be accessed like any other CORBA object, through a standard IDL interface.

In the example of Figure 3, method `get_interface`,

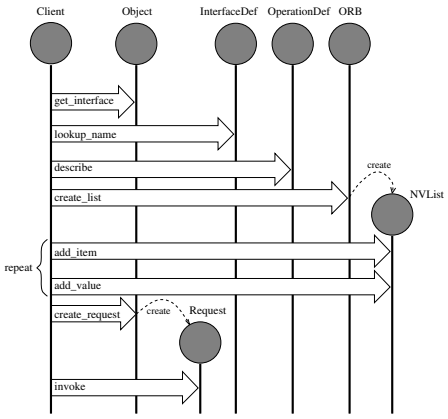

Figure 3. An Example of Dynamic Invocation.

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 the entry-point for navigating the Interface Repository.

With a complete method definition, it is possible to create an argument list to be used in the method call. This argument list is composed using a data structure called *Named Value List*, defined by CORBA. This structure is a list of pairs of names and values. Values, in turn, are tagged unions containing a type description (the tag) and the actual value. In this way, such lists can represent any method signature, supporting a form of dynamic typing.

A request in DII is represented by a CORBA *pseudo-object*[1] that contains the name of the method, the argument named value list, and a place for the return value (also a tagged union). A `Request` object is created by invoking the ORB method `create_request`.

The actual invocation of a `Request` object can be performed in three different ways [15]:

- the `invoke` method call sends the request and obtains the results;
- the `send_oneway` method call sends a datagram. In this case, no response is needed;
- the `send_deferred` method call sends the request and returns the control to the program, which must then poll for the response by calling `get_response` or `poll_response`.

From the server's point of view, the use of DII instead of an IDL static interface is completely transparent. Particularly, any service available through an IDL can be accessed

[1] A pseudo-object is an object whose interface can be defined in IDL, but which is not necessarily implemented as an ORB object. Pseudo-objects cannot be manipulated through DII.

through a dynamic invocation. Moreover, DII provides an extra level of flexibility that is necessary in some applications, such as desktops and operating systems [14].

On the other hand, some authors [16, 18] state some problems related to DII, such as a weak type checking and a run-time performance penalty. But the former problem exists with the stub mechanism too. If a CORBA object has its interface changed and its clients are not updated, run-time type errors can arise. In fact, the stubs used in the static interface use a low-level variant of DII to do their jobs [20]. The run-time performance penalty can be greatly reduced with the minimization of the number of queries to the Interface Repository, for instance by caching the most recent queries to it.

In practice, it is not an easy task to make applications incorporate new objects via DII. Although DII offers the support for applications to access new object types, there is still the difficulty of building the method calls. Using DII, this activity involves querying and constructing complex structures, particularly when there are structured arguments or results in the method call, like sequences of structures or structures with structured fields. With traditional bindings, programming with static interfaces is much simpler and results in more robust code for the developer.

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 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.

Using fallbacks, 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 coercing 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.

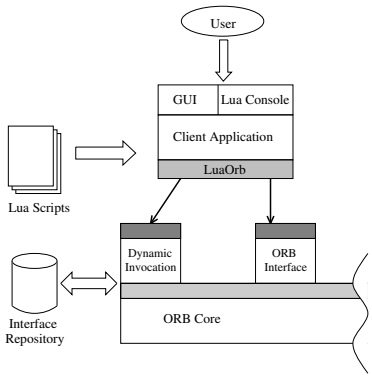

Figure 4. The architecture of a LuaOrb client application.

The Lua binding is implemented in a library called `LuaOrb`. The resulting architecture for integrating the LuaOrb library with an application and the ORB is illustrated in Figure 4.

4.1 How to use CORBA objects

To explain the use of CORBA objects in Lua, the example of an IDL interface in Figure 5 will be used.

The first step for using a distributed object is to create a local representation for it, called a *proxy*. To create a proxy for an object that implements a `nameServer`, function `createproxy` is used:

```
a_server = createproxy("nameServer")
```

When `createproxy` is called, 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`.

After a proxy has been created, the services of the related object can be requested. For example, the code in Figure 6 will print the IP address of a mail server in a given domain. This example shows how a Lua user handles CORBA objects in the same way she handles Lua objects. Notice that the Lua table `host` is dynamically mapped to the IDL structure `host`, when `hostAddress` is called. Such coercion is allowed as long as the Lua table has, at least, the same fields that the IDL structure. This coercion works re-

```
interface nameServer {
  typedef sequence<string> String_seq;

  struct host {
    string domain;
    string subdomain;
    string host;
  };

  String_seq whoDoes(in string service,
                     in string domain,
                     in string subdomain);

  string hostAccess(in host h);

  string hostAddress(in host h);

  ...
};
```

Figure 5. An example of an IDL interface.

```
domain_name = "puc-rio.br"
subdomain_name = "inf"
host_names = a_server:whoDoes("mail",
                        domain_name,
                      subdomain_name)
if host_names.n == 0 then
  error("no server in given domain")
end
host = {domain = domain_name,
        subdomain = subdomain_name,
        host = host_names[1]}
print(a_server:hostAddress(host))
```

Figure 6. Using a remote object.

cursively, so a list of Lua tables can be automatically converted to an array of records, for instance. Because of the dynamic nature of this coercion, small changes in an IDL interface, such as reordering and removing of struct fields, and changes between `short` and `float` or `array` and `sequence`, do not affect its uses in Lua.

It is interesting to note that the same mechanisms provided by LuaOrb also allow a use of CORBA in a lower level, with direct access to the Interface Repository and dynamic construction of calls. The Repository Interface is directly accessed by LuaOrb, since it is a standard CORBA object. Dynamic calls can be built with the Lua function `call`, which calls a function with a dynamic parameter list described in a table (similar to the Lisp function `apply`).

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 reference to a proxy in the host language (C++).

The following code fragment will be used to describe the execution of a method call:

```
print(a_server:hostAddress(host))
```

As explained in Section 2, the expression

```
a_server:hostAddress(host)
```

is equivalent to:

```
a_server["hostAddress"](a_server, host)
```

Since `a_server` is not a table, but a userdata, the index operation in the above expression triggers a fallback. The fallback handler, defined by LuaOrb, returns an internal function with the string `"hostAddress"` associated to it (in a *closure*). Then Lua calls this function, passing as arguments the userdata `a_server` (the proxy reference), and the table `host` (the argument to the faulty call).

With this information, LuaOrb asks the Interface Repository whether the IDL interface `nameServer` has a method called `hostAddress`. If this is the case, the description of this method (parameters, context information, result and exceptions) is retrieved from the IR. Otherwise, an error is signalled.

With the actual Lua parameters and the method description at hand, the parameter list for the dynamic call is built. The Lua parameters are coerced 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 `host` is converted into a `NVList` that describes the structure `host`.

After that, the method `hostAddress` is invoked. and then 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.

5 Related Work

Most CORBA language bindings, including the bindings for C, C++, Smalltalk [15], Java [11] and School [5], 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 [7] 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 [1]. 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 forces 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. Moreover, TclDii does not query the Interface Repository, so the programmer must explicitly specify the types to be used in each invocation.

Recently, other works have proposed special languages for dynamic interaction with CORBA. *GOMscript* [2] offers an abstraction to integrate three different models for network management: CORBA, SNMP and CMIP. *CorbaScript* [12] has been designed to be a generic object shell interpreter. Its main functionality is operation invocation of CORBA objects. As LuaOrb, CorbaScript is based on the same standard CORBA components: the Interface Repository and the Dynamic Invocation Interface. CorbaScript also supports the implementation of servers, which is still a future work for LuaOrb.

It is important to note that, unlike Lua, both GOMscript and CorbaScript have been specifically designed to interact with CORBA. LuaOrb, on the other hand, is a binding between CORBA and a pre-existing interpreted language, Lua. Nevertheless, Lua's reflexive mechanisms allow a binding that integrates CORBA objects into the language in a natural way. Moreover, its type mechanism has been shown to be flexible enough to represent all IDL types in a natural way.

6 Final Remarks

The LuaOrb library is fully implemented over *Visibroker for C++* [20], a CORBA 2.0 compliant implementation. At this moment, LuaOrb has been tested in two applications, a network management system [17] and a cooperative CAD system [8] .

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 type descriptions to build the binding. Our model can be adopted by other languages that support reflexivity and dynamic typing.

The Lua binding also 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.

This work has also addressed some problems related to DII. As discussed in [18, 13], the adoption of DII can give rise to a new generation of distributed applications, but there are some problems with the use of DII, namely the complexity of building requests, a weak type checking, and the run-time performance penalty. LuaOrb solves the first problem by raising the abstraction level of DII to the same level of common method calls. The weak type checking problem is solved by LuaOrb by its automatic mapping between Lua and IDL types. Finally, the run-time performance penalty has been greatly reduced by caching the most recently queries to the Interface Repository. Preliminary tests have given 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 [18]. 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.

Acknowledgments

This work has been partially supported by PETROBRAS, CNPq (the Brazilian Research Council) and CAPES.

References

[1] 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.

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

[4] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1), Feb. 1984.

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

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

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

[8] P. R. Gomes, B. Feijó, R. Cerqueira, and R. Ierusalimschy. Reactivity and pro-activeness in virtual prototyping. In *Tools and Methods for Concurrent Engineering'98*, 1998. to appear.

[9] J. Goodenough. Exeption handling: issues and a proposed notation. *Comm. ACM*, 18(12), Dec. 1975.

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

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

[12] P. Marle, C. Gransart, and J.-M. Geib. CorbaScript and CorbaWeb: A generic object-oriented dynamic environment upon corba. Technical report, Universite de Lille, 1996.

[13] T. Mowbray and R. Malveau. *CORBA Design Patterns*. John Wiley & Sons, 1997.

[14] T. Mowbray and R. Zahavi. *The Essential CORBA*. John Wiley & Sons, 1995.

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

[16] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 1997.

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

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

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

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