A Dynamic Approach for Composing CORBA Objects

Renato Cerqueira
rcerq@inf.puc-rio.br

Roberto Ierusalimschy
roberto@inf.puc-rio.br

Noemi Rodriguez
noemi@inf.puc-rio.br

PUC-RioInf.MCC43/97 November, 1997

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 a dynamic approach, mapping the dynamic character of CORBA 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.

Keywords: Componentware, Distributed Systems, CORBA.

Resumo: 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 em uma abordagem dinâmica, mapeando o carater dinâmico de CORBA 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.

Palavras-chave: Desenvolvimento de Software baseado em Componentes, Sistemas Distribuídos, CORBA.

This work has been sponsored by CNPq and Petrobrás/CENPES.

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 [CIS92, Ous97].

The concept of "interpreted language" is not very precise, at least for two reasons: Often, a language can be implemented both through a compiler and through an interpreter; and implementations that combine both techniques are very common. Nevertheless, the term "interpreted language" is useful to designate a group of languages with distinct characteristics of those called "conventional", such as FORTRAN and C. To make this concept more clear, the following definition is used in this paper: A language will be classified as interpreted if it has some mechanism to execute pieces of code created dynamically (e.g. the *eval* function in Lisp); in other words, if the interpreter is directly available to the language. According to this definition, languages like Lisp, Lua [IdFC96, FIC96], Mumps and Tcl [Ous94] are interpreted, while Pascal, C, C++ and Java do not belong to this category.

Many properties of interpreted languages derive directly from the above definition, such as *interactivity*, support for *end user programming*, and *reflexivity*. Other properties are not direct consequences of the definition, but make easier the design of interpreted languages, such as absence of declarations and dynamic typing.

The interactivity provided by an interpreted language allows a programmer to enter language commands one by one, for immediate execution, as in a console with a shell language. With an interactive console, an operator has direct access to all facilities and options of the system, without the need of specific tools. A console is also 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 for rapid prototyping, where several design options can be implemented and tested in a short period of time.

The concept of reusable modules, or "componentware", is also becoming increasingly important in software development. One ideal to be reached by Software Engineering, in the reuse area, is the construction of new applications from existing and reliable software components, like in other engineerings [CN91].

One of the goals of the componentware approach is to support the extension of existing systems by the simple replacement or addition of new components. Moreover, it is also important to allow dynamic changes in an application composition, for better flexibility [Sie96, Sta91].

In this context, it seems important to explore the use of componentware 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* element in the construction of new programs from available modules. For this, it is necessary some kind of interface between interpreted languages and componentware facilities.

In the last few years, CORBA [Obj95, Sie96] has been gaining momentum as one of the main platforms for construction of componentware. Several implementations are now available, with many including 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 or server to be implemented in the most convenient language.

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. Using the Interface Repository and the Dynamic Invocation Interface (DII), a program can dynamically compose method calls, with dynamic typing. Therefore, programs can incorporate new components, and even new interfaces, without changes and/or recompilations. This makes interpreted languages specially attractive for the gluing function mentioned above, since it means that applications can be dynamically and transparently configured or reconfigured. Such languages fulfill 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 dynamically [Sie96].

In this paper, we describe LuaOrb, a binding between the interpreted language Lua [IdFC96, FIC96] and CORBA. As usual, this binding defines mappings between Lua and IDL types. However, unlike other bindings, this mapping is based on the dynamic typing of Lua and the dynamic character of CORBA. To do this, LuaOrb accesses the Object Request Broker (ORB) through the Dynamic Invocation Interface, instead of using IDL stubs. Using LuaOrb, 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, query the Repository Interface to get the method signature, 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 any available CORBA server object, as if they were local objects.

The next section gives an overview of the language Lua. Section 3 discusses CORBA, with some emphasis to its Dynamic Invocation Interface, and then the binding between Lua and CORBA is discussed in Section 4. Section 5 shows some current applications of LuaOrb, and the last section is reserved for some final remarks.

2 The Extension Language Lua

Lua is a general purpose configuration language that arose from the need of our group to use a single extension language to customize industrial applications [IdFC96]. Currently, Lua is being used in more than a hundred products and prototypes, in many academic institutions and companies. 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 rather different domains, thus creating customized programming languages sharing a single syntactical framework [Bec91].

Functions in Lua are *first class* values. A function definition creates a value of type `function`, and assigns this value to a global variable (like `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.

```
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.

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`, with a single value, also called `nil`, whose main property is to be different from any other value; `userdata`, that is provided to allow arbitrary host data (typically C pointers) to be stored in Lua variables; and `table`.

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={"Sun","Mon","Tue","Wed",
      "Thu","Fri","Sat"}
```

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

Several features in Lua have a role 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 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.

Because functions are first class values, table fields can contain functions. This property allows the implementation of some interesting object-oriented facilities, which are made easier by some syntactic sugar. For instance, the expression:

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

is syntactic sugar for

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

That is, the function stored in field `foo` from object `receiver` is called, with `receiver` as its first argument (which plays the role of `self`). A more complete treatment of Lua's object oriented facilities may be found in [IdFC96].

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. For instance, consider again the statement:

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

Usually, this expression signals an error whenever `receiver` is not a Lua object and/or `foo` is not a method of `receiver`. With fallbacks, the program can catch this error, and give this expression a meaningful semantics in this context. This kind of facility is the key to the implementation of LuaOrb, as will be discussed in Section 4.

3 CORBA

The CORBA standard [Obj95] provides communication facilities to applications in a distributed environment. All communication between CORBA objects is mediated by the *Object Request Broker* (ORB) [Obj95, 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 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 un-marshals 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 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.

The Dynamic Invocation Interface (DII) is a generic facility for invoking any operation with a runtime-defined parameter list. To perform an invocation using DII, a program executes four steps:

- The identification of the desired object;
- The retrieval of its interface (type);
- The invocation construction;
- The actual request invocation, receiving its results.

Since DII allows the discovery of the object type and methods at run time, it is possible to implement mechanisms of *dynamic typing* using DII.

The DII and other CORBA services, like the Trading and Naming Services [Sie96], offer the basic mechanisms to support a dynamic distributed object environment [Sie96, MM97]. In dynamic environments, applications can find out and incorporate new components at run-time. Besides, applications can be adapted on the fly to component changes. This level of flexibility is very important to some applications, such as desktops and operating systems [Sie96, MZ95], network management tools [Ban97], and cooperative applications [MGG96].

To build dynamic requests through the DII, it is necessary to acquire the meta data that describe the object interface. CORBA offers many reflexive mechanisms, where objects provide information about themselves. Particularly, every CORBA object offers a method, named `get_interface`, to access its interface description. The meta data that describe object interfaces are provided by the Interface Repository. CORBA specifies approximately ten calls for navigating the Interface Repository.

DII offers the support for applications to access new object types, and the Naming and Trader services offered by CORBA can be used to 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. In practice, it is not an easy task to make applications incorporate new objects via DII. With traditional bindings, programming with static interfaces is much simpler and results in more robust code for the developer.

Therefore, mechanisms that offer a better support for dynamic composition of CORBA objects become a very important topic [Sie96]. The next section presents LuaOrb, a binding of the interpreted language Lua to CORBA. LuaOrb offers a more suitable support to develop open applications, which allows the dynamic composition of CORBA objects.

4 The LuaOrb Binding

The definition of the mapping between Lua and CORBA has tried to respect the common use of Lua. In that way, it should be possible to use CORBA objects in the same way as other Lua objects. To reach this goal, the reflexive characteristics of Lua and CORBA were fully explored.

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 2.

The next two subsections give more details about how LuaOrb works.

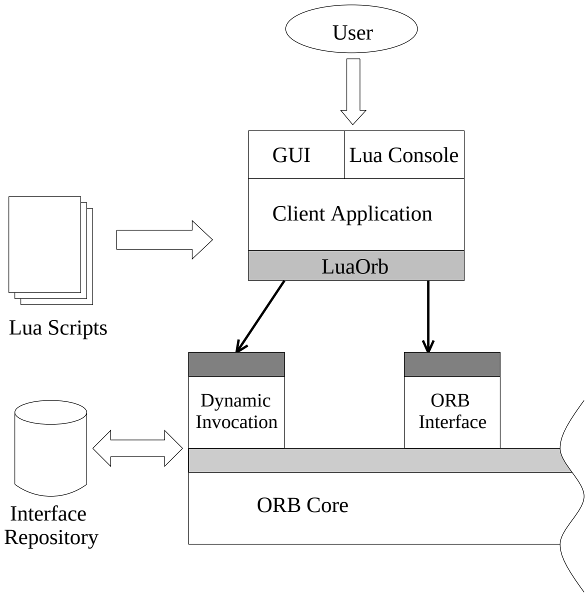

Figure 2: The architecture of a LuaOrb client application.

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

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

Figure 3: An IDL interface.

4.1 How to Use CORBA Objects

To explain the way Lua uses CORBA objects, the example of an IDL interface in Figure 3 will be used.

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

```
a_foo = createproxy("foo")
foo2 = createproxy("foo","Master")
```

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. When the second parameter is specified, `createproxy` will only succeed if there is a server object with the same name of this parameter. This function is basically a direct mapping to its equivalent function in the ORB API.

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="New Book"}
a_foo:add_book(a_book)
if a_foo:test() then
  x = foo2:div(y,z)
end
```

The above example shows how a program handles CORBA objects in the same way it handles Lua objects.

It is worth noticing how the Lua table a_book is dynamically mapped to the IDL structure book, when add_book 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 recursively, so a list of Lua tables can be automatically converted to an array of records, for instance. Because of its dynamic nature, this coercion allows that many changes in an IDL interface do not affect its uses in Lua, such as reordering and removing of struct fields; and changes between IDL types which have similar representations in Lua, such as short and long, or array and sequence.

Since Lua works as an embedded language, its binding to CORBA is more relevant for clients (applications) than for servers. Therefore, at this moment, 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, using the CORBA Dynamic Skeleton Interface (DSI), is another interesting topic.

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 returned value of createproxy is a Lua userdata, which is a C++ reference to a proxy.

To describe the implementation of method calls, the following code fragment will be used:

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

As explained in Section 2, when the expression a_foo:add_book is executed, a fallback handler is called, since a_foo is not a real object. The LuaOrb fallback handler receives as parameters the string "add_book" (the method name), the userdata a_foo (the proxy reference), and the table a_book (the parameters of the method call). 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, LuaOrb builds the parameter list for the dynamic call. For this, LuaOrb coerces the Lua parameters into the types of the IDL description (if this conversion is not possible, LuaOrb issues a run-time error).

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

A possible run-time performance penalty has been greatly reduced with the minimization of the number of queries to the Interface Repository. This has been achieved simply by caching the most recent queries. A more detailed description of the binding can be found in [CRI97b, CRI97a].

5 Sample Applications

This section discusses some example applications, currently under development, which exploit the dynamic characteristics of LuaOrb. The first example is a network management system that allows access to both CORBA and more traditional SNMP [CFSD90] servers. The second one is a cooperative CAD system.

SNMP, the TCP/IP family network management protocol, supports a model where *agents* act as servers, providing information about the managed devices to *managers* or management applications, which act as their clients. The information provided by each specific type of agent is described by a *MIB* (management information base), which contains the description of variable names and types of values.

In order for a management application to provide relevant services, it must be programmed with some knowledge about the agent´s MIB. A systems administrator typically needs more than a browser to manage a device: She needs summarized information and signalling of important events. This need has traditionally been answered by specific management applications supplied along with each new device. This typically introduces portability and integration problems.

We are currently using Lua as the basis for a management platform [RCIM97]. The goal is to allow technically advanced users the possibility of extending existing management applications with new functionality, possibly for the management of new devices, or for graphical or textual display of statistical summaries of monitored data. End-users will also be able to use Lua facilities for simple customization tasks. A new library, LuaMan, was developed, offering SNMP-related functions and other facilities which may be useful in network administration, such as ICMP communication and access to DNS. Access to CORBA objects is extremely interesting in this context, since there is a strong trend for network-related services to be offered as CORBA servers. The dynamic characteristic of LuaOrb means that access to new servers may be dynamically incorporated into management applications.

For instance, a name server, implemented as a CORBA object, can be used to help a network manager to get a specific host reference. After that, SNMP queries can be performed over this host. Figure 4 presents an IDL specification for this name server, and Figure 5 shows an example of the use of LuaOrb in combination with SNMP. In this example, the name server is accessed to find a reference to a host which is a mail server, and then the SNMP protocol is used to get some information about it.

The user of this network management system has available a Lua command console, where she can perform any operation available in the Lua environment (for instance, user interface and graphic facilities, CORBA servers, SNMP protocol, etc). In this way, any CORBA component, possibly found through the trading server or even through more traditional means, such as a message to a newsgroup, can be used to help in the management activity.

In the other application, a cooperative CAD system uses CORBA objects to wrap the application objects. The relationships (*constraints*) among the application objects can be defined and modified through Lua scripts, at run-time. With this mechanism, the insertion of a new

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

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

  string hostAccess(in string domain,
                    in string subdomain,
                    in string host);

  string hostAddress(in string domain,
                     in string subdomain,
                     in string host);

  ...
};
```

Figure 4: IDL interface for the name server.

```
  ns = createproxy("nameServer")
  dom = "puc-rio"; subd = "inf"
  hosts = ns:whoDoes("mail",dom,subd)
  if hosts.n == 0 then
    error("no mail server in the given domain")
  end
  p = ns:hostAddress(dom,subd,hosts[1])
  c = ns:hostAccess(dom,subd,hosts[1])
  session = snmp_open{peer=p, community=c}
  varBind = snmp_get(session, "sysDescr.0")
  print(sprint_var(varBind))
```

Figure 5: Accessing the name server via LuaOrb.

CAD object in the system and the dynamic constraint configuration are more flexible. Besides, the applications that compose the cooperative system become less sensitive to changes in other components, since the application should not need to be re-compiled neither stopped; only the Lua configuration scripts may need changes.

6 Related Work

Most CORBA language bindings, including the bindings for C, C++, Smalltalk [Obj95], 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. Moreover, TclDii does not use the Interface Repository, so the programmer must specify the types to be used in each invocation.

Recently, some new works have adopted solutions very similar to the approach adopted by LuaOrb. *GOMscript* [Ban97] offers an abstraction to integrate three different models for network management: CORBA, SNMP and CMIP. *CorbaScript* [MGG96] was designed to be a generic object shell interpreter. Its main functionality is operation invocation onto CORBA objects. As LuaOrb, CorbaScript is based on the same standard CORBA components: Interface Repository, DII and ORB Interface.

It is important to note that, while GOMscript is oriented to the application domain of network management and CorbaScript is limited to integrate CORBA objects, LuaOrb is based on a general purpose language that can be used in different domains and can integrate different component models.

7 Final Remarks

This work has addressed some problems related to dynamic composition of distributed CORBA objects. As discussed in [Sie96, MM97], the adoption of dynamic mechanisms can give rise to a new generation of distributed applications. But there are some problems with the use of DII, the "official" CORBA dynamic mechanism. 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. But this 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 [Vis96].

The run-time performance penalty was greatly reduced with the minimization of the number of queries to the Interface Repository, by caching the most recent queries to it.

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, 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, but using a more abstract level over DII. Siegel also states that this application architecture is a necessary step in the transition from monolithic applications to user-centric distributed computing.

As of this moment, standards have been established for bindings between CORBA and C, C++, Java, and Smalltalk. It is not at all clear, at this point, whether a binding to the CORBA IDL is really feasible 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 other languages mentioned above, is also an important step towards the evaluation of the effective integrating potential of CORBA. Our model can be adopted by other languages that support reflexivity and dynamic typing.

The LuaOrb library has been implemented over *Visibroker for C++* [Vis96], a CORBA 2.0 compliant implementation. But, since it only depends on standard CORBA components, it can be moved to other CORBA implementations.

An important continuation of this work is the development of other case studies that take more advantage of the use of LuaOrb, like the network management system and the cooperative CAD presented in section 5.

We believe that LuaOrb can be used basically in a scenario where new applications are assembled from the reuse of available components, with Lua performing the role of a *Composition Language* [NM94, NT95]. Therefore, we start with the prototype of an application and finish with a highly customizable one.

In this way, the binding of Lua to OLE automation [Cla96] and DCOM [Bro95] technologies is another important issue. With this new binding, LuaOrb will be able to integrate components of different models in the same application, yet using these components in a transparent way.

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.

[Ban97] Bela Ban. *A Generic Management Model for CORBA, CMIP and SNMP.* PhD thesis, University of Zurich and IBM Zurich Research Laboratory, Rueschlikon, Switzerland, 1997.

[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.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol, 1990. RFC 1157.

[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.

[CN91] Brad Cox and Andrew Novobilski. *Object-oriented Programming: an evolutionary approach*. Addison Wesley, 1991.

[CRI97a] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. Using Lua to access CORBA objects. Monografias em Ciência da Computação ??/97, PUC-Rio, Rio de Janeiro, Brazil, 1997.

[CRI97b] Renato Cerqueira, Noemi Rodriguez, and Roberto Ierusalimschy. Binding an interpreted language to CORBA. In *II Simpósio Brasileiro de Linguagens de Programação*, pages 23–36, 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.

[MGG96] Philippe Marle, Christoph Gransart, and Jean-Marc Geib. CorbaScript and CorbaWeb: A generic object-oriented dynamic environment upon corba. Technical report, Universite de Lille, 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.

[Obj95] Object Management Group, Inc., Framingham, MA. *The Common Object Request Broker Architecture and Specification; Revision 2.0*, jul 1995.

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

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Ous97] John Ousterhout. Scripting: Higher level programming for the 21st century. Technical report, Sun Microsystems Laboratories, mar 1997.

[RCIM97] N. Rodriguez, R. Cerqueira, R. Ierusalimschy, and A. Moura. Aplicações de gerência com comportamento dinâmico. In *Anais do SFBSID'97*, Fortaleza, Brasil, 1997.

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

[Sta91] Manfred Stadel. Object oriented programming techniques to replace software components on the fly in a running program. *ACM SIGPLAN Notices*, 26(1):99–108, 1991.

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