Dynamic Configuration with CORBA Components

Noemi Rodriguez Roberto Ierusalimschy Renato Cerqueira

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

Abstract

Most existing support for dynamic reconfiguration assumes that component interfaces specify input and output channels. The CORBA model, however, supports specifically a client-server architecture, with component interfaces describing only the provided services. Besides, the existing bindings for CORBA, based on static stubs, impose considerable difficulties on dynamic configuration. This paper investigates the use of the interpreted language Lua as a tool for dynamic configuration of distributed applications using CORBA components.

1. Introduction

In most systems that support the idea of configuration there is a clear distinction between the programming of *components* and the programming of applications. The configuration of an application is based on the idea of "joining" existing components, frequently using a language different from the one which is used for the coding of each component. This could be described as a distinction between "programming in the large" (for building complete applications) and "programming in the small" (for building the components themselves). The use of different languages for these two programming levels reflects the existence of different concerns.

In many cases, the "meeting point" between these two levels is given by some kind of component interface description. The different components in a distributed application typically interact through exchange of information. Interface descriptions must contain information about the kind of exchange expected by each component.

In languages such as Conic [7] and Darwin [6], the interface of each component is described in terms of input and output channels. In Darwin, for example, a component is described in terms of the communication objects it *provides* (roughly comparable to input channels) and the communication objects it *requires* (roughly comparable to output channels). This approach allows the specification of different paradigms of interaction between processes [1]. Filtering structures can be easily built from components such as the one shown in Figure 1.

```
component filter {
  provide left <port,int>;
  require right <port,int>;
          output <port,int>;
}
```

Figure 1. example of component specification in Darwin.

More recently much attention has been given to the use of components in the CORBA model [14], where the interface of each component is described in terms of the method calls which it accepts, through an *Interface Description Language* (IDL). Figure 2 shows an example of IDL use. This approach is clearly aimed at supporting the client-server interaction paradigm.

Both approaches have in common an emphasis on typed communication; in both of them, interfaces define the type and number of objects passed to and from a component. This is essential to ensure that applications built from these components will use them correctly. Both approaches also have as one of their goals the support for component reusability. However, they differ largely as to the level of abstraction which is being supported. As stated in [4], the required and provided communication objects of Darwin can be seen as required and provided services. In CORBA, only provided services are declared in the interface of a component, and nothing is known about the methods which will in turn be called by this component. On one hand, this gives the programmer much less control over the construction of a complete application. On the other hand, for the

```
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 2. example of component specification in CORBA.

specific case of client-server applications, this mechanism supports the use of existing services as black boxes, about whose implementation the programmer needs not worry. Rather than coupling service requirements to service provisions, the building of an application in this case consists mainly of decisions about which services to use. The new application, in this view, is the only one *requiring* any services.

The configuration, and specially reconfiguration, of distributed applications has been the focus of much work in the first, more traditional, approach. However, in the CORBA approach little has been discussed about this subject. The need for dynamically reconfigurable applications using CORBA is becoming increasingly apparent, specially in areas such as network management and other real time control systems. Fault tolerance and the dynamic integration of newly available services are important goals in these areas. However, most of the existing bindings for CORBA are based on compiled stubs, making dynamic incorporation of new facilities a complex operation. In this work we investigate the use of Lua [5], an interpreted extension language, as a configuration tool for CORBA-based applications. The discussion is organized as follows. We first review some common features of interpreted languages, with special attention to their role in the configuration of applications. Next we make a brief presentation of Lua. We then describe LuaOrb, a binding based on the CORBA dynamic invocation interface (DII), which allows CORBA components to be dynamically incorporated to an application. Finally, we present some examples of the use of Lua as a configuration language for CORBA-based applications.

2. Interpreted Languages

To say that a language is interpreted is not a very precise statement, for at least two reasons. In the first place, the same language can usually be either interpreted or compiled. Secondly, most language implementations use a combination of the two techniques. In order to try to be more precise, we will adopt the following definition: a language will be said to be *interpreted* if it offers some mechanism for execution of chunks of code created dynamically; in other words, if the interpreter is directly accessible to the language. According to this definition, languages such as Lisp, Lua, Mumps and Tcl are interpreted, while Pascal, C, C++ and Java are not.

Many important features of interpreted languages, such as *interactivity*, *end user programming*, and *reflexivity*, follow directly from the preceding definition. Other properties, such as dynamic typing and the absence of declarations, are not a direct consequence of this definition, but make the design of interpreted languages easier. Finally, powerful features frequently associated to interpreted languages, such as garbage collection and dynamic arrays, do not have any direct relation with interpretation, and are in fact present in several compiled languages.

Programs are frequently developed in one environment, and later installed and configured in a different, target environment. Much of this configuration activity relates to setting program variables (such as IP addresses, local directories, server names, etc) to appropriate values. Such configuration may be done through the use of environment variables or simple text files such as the X system "resource" files. However, as programs become more complex, configuration possibilities increase, specially in interface-related issues. Many programs allow menus to be created or modified, and even new operations ('macros') to be defined. To support such flexibility, configuration must many times be controlled by a full fledged programming language. This need fueled the development of *extension* languages such as Tcl [10] and Lua [5].

With the incorporation of an interpreter to the run-time environment, program configuration files can contain much more than a list of data and options. A configuration file can contain initialization routines which use all the expressive power available in a programming language (conditionals, loops, abstractions, etc).

These configuration languages, when expressive enough, can be employed by several classes of users. Roughly, one can distinguish two basic categories. The first one is composed by systems administrators and other technically advanced users, who can use all the expressiveness of the language to adapt a program to a given environment. This can include the definition of security policies, adaptation to local directory structures, and so on. A second class of users are the so called *expert users*, usually final users of the application but with a natural interest in programming. Such users will in general not employ all the expressiveness of the configuration language, but may program basic chores, such as menu configuration conditional to the graphical terminal where the application is running.

In the preceding paragraphs we have been using the term "configuration" in the sense of tailoring an application to specific needs and environments. However, it must be noted that this task is not so different in its goals from the configuration of distributed applications. Rather, these seem to be different points in a continuum of complexity levels. Therefore, it makes sense to explore the utilization of a language such as Lua, that has been proving itself very useful for the extension and configuration of sequential applications, as a tool for the configuration of distributed applications.

Two weaknesses generally associated to interpreted languages are efficiency and robustness. In fact, interpreted languages are normally much slower than their compiled counterparts (a factor of 10 is not uncommon). However, in many application domains, a substantial amount of execution time is spent on primitive operations, such as communication or input/output calls, which have execution times that are independent of the coding of the application. For such programs, the efficiency of compiled languages is of little value. Similarly, robustness in a language must be evaluated in the context of its use. Static verification is certainly an important ally in the development of large software projects. Yet conventional programming languages pay a price for static typing, namely, the loss of polymorphism and flexibility. Besides, if on one hand static typing is not available, on the other hand interpreted languages usually rely on sturdy run-time error-checking mechanisms (for uninitialized variables, dangling references, etc) which may be very useful for program debugging. In the case of CORBA based applications, it is important to emphasize that each CORBA component will typically have been developed with a conventional, statically typed language. The use of an interpreted language as a *glue* between these components may result in a run-time error if an attempt is made to invoke a non-existing method or to call a method using incorrect parameters. As explained in the next section, such situation will generate *fallbacks* in Lua, that are in some ways similar to exceptions, and may be appropriately handled for program robustness.

3. 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 [5]. Lua integrates in its design data-description facilities, reflexive facilities, and familiar imperative constructs. We believe that this integration of simple syntax and powerful programming constructs has an important role in accommodating the needs of many different classes of programmers.

As a simple example of Lua code, Figure 3 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 $b = (f(a_1), f(a_2), \ldots, f(a_n))$.

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

Figure 3. 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. 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"]`. A pre-defined function, `next`, provides iteration through tables, returning an index and a corresponding value each time it is called.

In Lua, functions are first-class values, and can therefore be assigned to table fields. This feature 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 [5].

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 runtime, 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.

Part of the reflexive nature of Lua is also provided by *fallbacks* (*tag methods* in Lua 3.0). Fallbacks allow programmers to change the "normal" behavior of Lua, especially in abnormal conditions. For instance, consider again the statement:

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

Whenever `receiver` is not a Lua object and/or `foo` is not a method of `receiver`, this expression usually signals an error. 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 5.

Although this is not in any way part of the language, Lua is very often used in event-driven environments. It is already traditional to use event-driven programming to deal with graphical interfaces. We have found that this same paradigm can also be useful in many other situations. [3] and [13] describe an event-driven extension to Lua for distributed programming. Section 6.2 discusses the use of an event-driven approach in the configuration of an application.

4. CORBA

The CORBA standard [9] provides communication facilities to applications in a distributed environment. All communication between CORBA objects is mediated by the *Object Request Broker* (ORB) [9, 14]. 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. Using this approach, clients must be recompiled each time a change in the server's interface takes place (or each time a new type of object is to be used as a server).

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 the four following steps: the identification of the desired object, the retrieval of its interface (type), the invocation construction, and 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.

DII and other CORBA services, like the Trading and Naming Services [14], offer the basic mechanisms to support a dynamic distributed object environment [14]. 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 [14], network management tools [2], and cooperative applications [8].

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.

Using DII, the programmer has also access to more method call modes than when using static stubs. CORBA supports three types of calls: *synchronous*, which stands for the traditional RPC semantics, *oneway*, which allows the client to invoke a method and continue its execution without waiting for completion, and *deferred synchronous*, which allows the client to continue its execution immediately after a method call but to later poll the server for a result. This last possibility is available only through the DII.

It is therefore clear that use of the DII can be interesting in many cases, but represents a difficult task with the current existing support. Mechanisms that offer a better support for dynamic composition of CORBA objects thus become an important topic [14]. 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.

5. 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 acces-

```
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 4. An IDL interface.

sible from Lua with no need of previous declarations, and with dynamic typing. To achieve this goal, the binding was built upon DII.

When the Lua program executes a method call, the LuaOrb binding intercepts it (through the use of fallbacks), 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 the 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`. To explain how it works, the example of an IDL interface given previously in Figure 2 will be used. The example is repeated in Figure 4.

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

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

As shown by these examples, `createproxy` can receive two parameters: The first one (mandatory) 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 follows:

```
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
```

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

In the example, 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 many changes in an IDL interface not to affect its uses in Lua, such as reordering and/or removing of struct fields, and changes between IDL types which have similar representations in Lua, such as `short` and `long`, or `array` and `sequence`. Although this is a discussion out of the scope of this paper, it is worth noticing that the type system that emerges from these properties is one in which *structural compatibility* is enforced [12].

As mentioned previously, use of the DII allows a mode of invocation called *deferred synchronous*, where a client triggers a method and later polls for completion. From the server's point of view, it is transparent whether this invocation mode is used or not. To specify a deferred synchronous call in Lua, the programmer must prefix the method name with `deferred_`, as in:

```
a_foo:deferred_test()
```

A deferred method call returns a handler which can be used for retrieving the method result.

Deferred synchronous calls are specially natural in event-driven programming. In this setting, it is interesting to be able to define a function to be called upon the completion of the method. This is supported in LuaOrb through function `completion_event`, which takes as parameters a handler returned by a deferred synchronous method call and a function. Upon method completion, this function is called with the method results as parameters.

6. Configuring CORBA applications

In this section we discuss some examples of application configuration using CORBA for the definition of the components and Lua as the configuration language.

6.1. Interactive Configuration

One very important, although simple, use of Lua as a configuration tool for applications based on CORBA objects is the use of a command console. The Lua interpreter can be used interactively as a console, and, when linked with LuaOrb, this console provides the user with an interactive environment for access to CORBA objects.

Over the last months, we have been using the command console in the specific context of network management [11]. There is a strong trend for network-related services to be offered as CORBA objects, specially in the area of telecommunications. However, most existing network devices provide management information through the SNMP protocol [15]. Another library for Lua, LuaMan, was developed for access to SNMP network management agents. Linking the Lua console to LuaMan and LuaOrb results in a quite flexible environment for experienced network administrators. The dynamic characteristic of LuaOrb means that access to new servers may be dynamically incorporated into management applications.

```
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 5. IDL interface for the name server.

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 5 presents an IDL specification for this name server, and Figure 6 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 mail server, and then SNMP 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 (such as user interface and graphic facilities, access to CORBA servers, SNMP agents, etc). In this way, any CORBA component, possibly discovered through the trading server or even through more alternative means, such as a message to a

```
ns = createproxy("nameServer")
dom = "puc-rio"; subd = "inf"
hosts = ns:whoDoes("mail",dom,subd)
if hosts.n == 0 then
  error("no mail server in this 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(var2string(varBind))
```

Figure 6. Accessing the name server via LuaOrb.

newsgroup, can be used to help in the management activity.

6.2. Sets of Workers

In this subsection we consider the *calcpi* program used in [6] to illustrate a workers-supervisor application. This program computes an approximation to π by calculating the area under the curve $4/(1+x^2)$ between 0 and 1 using numerical integration. The program is structured as a set of *workers*, with each computing a part of the integration, and a *supervisor* combining the results. The main configuration problem in this case is to bind the output generated by the workers (the partial integration result) to the input of the supervisor object that collects them.

One solution, which is useful in many cases, is to specify the supervisor object as a parameter for the worker call. A worker component could have an IDL definition such as:

```
interface worker {
  oneway void work(in long id,
                   in long nw,
                   in long intervals,
                   in supervisor s);
};
```

where parameter `s` indicates the object that should be called upon the completion of the required work. Note the presence of the *oneway* modifier, which indicates that invocation of method *work* is asynchronous. Figure 7 illustrates a possible C++ implementation of the *worker* interface. The supervisor object is also shown in Figure 7. For the sake of simplicity, details related to CORBA usage are ommitted from the example code.

With the preceding definitions, the complete application could be defined by the Lua code shown in Figure 8.

```
class worker ... {
  ...
};
void worker::work(long id, long nw,
          long intervals, supervisor s) {
  double area = 0.0;
  double width = 1.0/intervals;
  for (int i=id; i<intervals; i+=nw) {
    double x = (i+0.5)*width;
    area += width*(4.0/(1.0+x*x));
  }
  s.result(area);
}
class supervisor ... {
  double area;
  long nr; -- number of calls to result
  long nw; -- number of workers
  ...
};
supervisor::supervisor(long n) {
  area = 0.0; nr = 0;
  nw = n;
}
void supervisor::result(double a) {
  area += a;
  if (++nr==nw) {
    printf("approx pi %20.15lf\n", area);
    exit();
  }
}
```

Figure 7. worker and supervisor classes for *calcpi* program.

Another way of structuring this same application would be through the use of an event-driven configuration script. In this case a *supervisor* object would not be necessary, since its role could be performed by the configuration script itself. Supposing the following IDL definition for the worker component:

```
interface worker {
  double work(in long id,
              in long nw,
              in long intervals);
};
```

the configuration script shown in Figure 9 implements this structure. In that code, the deferred synchronous call mode and function `completion_event` are used so that function `combine` is called each time a deferred call is completed. This function then adds to total and checks whether all workers have finished.

```
local s = createproxy("supervisor")
local i = 1
while i <= nw do
  local w = createproxy("worker",
                        "pi_worker"..i)
  w:work(i, nw, intervals, s)
  i = i+1
end
```

Figure 8. Lua code for *calcpi* program, first version.

It is interesting to note that, in both versions of the `calcpi` program, dynamic instantiation is being handled by the `createproxy` function call. The effective creation of a new object instance when `createproxy` is called depends, as explained in section 5, on the non-existence of an active instance with the given name.

7. Final Remarks

The problem of distributed application configuration has traditionally been addressed by supposing that the interface of each component provides information about both provided and required services. In the last years, however, the use of CORBA-style components, about which only provided services are known, is becoming more and more widespread. This paper has discussed the possibilities and limitations associated to the use of an interpreted language as a tool for dynamic configuration of distributed applications based on CORBA components. We believe we have shown that this is a promising approach. However, some points remain open for future study.

In the first place, as mentioned in the previous section, the CORBA framework does not give the programmer much control over dynamic object instantiation. When specifying a distributed application, it is frequently essential to define whether the application will use any instance of a given class of components or if it requires the explicit creation of a new instance. With CORBA, the programmer has much less control over instance creation than using a dedicated runtime system. On the other hand, the ease with which a new application can be bound to existing object instances is one of the advantages of using CORBA. Thus, one issue for future study is the specification of instantiation constraints in a way compatible with the CORBA standard.

A second interesting point is the use of the event-driven approach coupled with deferred synchronous method calls. So far, we have seen that this technique can be successfully applied in small examples, such as the one discussed in sec-

```
function combine (partial)
  area = area + partial
  answers = answers + 1
  if answers == nw then
    print(format
         ("approx pi %20.15f\n", area))
  end
end --combine

nw = 5
intervals = 1000000
answers = 0
area = 0
local i = 1
workers = {}
while i <= nw do
   workers[i] = createproxy("worker",
                             "w"..i)
   i = i+1
end
i = 1
while i <= nw do
   completion_event(
     workers[i]:deferred_work(
                  i, nw, intervals),
     combine)
   i = i+1
end
```

Figure 9. Lua code for *calcpi* program, second version.

tion 6.2. We now intend to evaluate this approach in the more "real-sized" context of network management applications.

We are also interested in integrating the configuration of distributed applications with the language extension for message passing mentioned in section 3. This extension allows Lua programs to incorporate new code dynamically, through received messages [13]. Using this mechanism, an application controlled by a Lua script may be dynamically reconfigured by messages sent to the controlling script.

Lua is implemented in ANSI C and is currently running on a wide range of platforms (for more information, see `http://www.tecgraf.puc-rio.br/lua/`).

The LuaOrb library has been implemented over *Visibroker for C++* [16], a CORBA 2.0 compliant implementation. We believe porting LuaOrb to other platforms will bring no major problems, since the implementation relies only on standard CORBA components. The run-time performance penalty normally associated to the use of DII 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.

Acknowledgements

This work was partially supported by CNPq (the Brazilian Research Council) and by PETROBRAS.

References

[1] G. Andrews. Paradigms for process interaction in distributed programs. *Computing Surveys*, 23(1), 1991.

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

[3] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. Uma experiência em programação distribuída dirigida por eventos. In *Anais do PANEL95*. SBC, 1995.

[4] S. Crane, J. Magee, and N. Pryce. Design patterns for binding in distributed systems. *SIGPLAN Notices*, 29, 1995. OOPSLA'95 Proceedings).

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

[6] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5), 1994.

[7] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in conic. *IEEE Trans. on Software Engineering*, SE-15(6), 1989.

[8] 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.

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

[10] J. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.

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

[12] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *Sigplan Notices*, 28(8), 1993.

[13] N. Rodriguez, C. Ururahy, R. Ierusalimschy, and R. Cerqueira. The use of interpreted languages for implementing parallel algorithms on distributed systems. In *Euro-Par'96 Parallel Processing*, Lyon, France, Aug. 1996. Springer-Verlag. LNCS 1123.

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

[15] W. Stallings. *SNMP, SNMPv2, and CMIP: The Practical Guide to Netowk Management Standards*. Prentice-Hall, 1993.

[16] Visigenic Software, Inc. *VisiBroker for C++ Programmer's Guide, Version 2.0*, 1996.