# Dynamic Reconfiguration of CORBA-based Applications

Noemi Rodriguez and Roberto Ierusalimschy

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

**Abstract.** Most current support for dynamic reconfiguration assumes that component interfaces specify input and output channels. Component models such as CORBA, however, support a client-server architecture, where component interfaces describe only the offered services. This work discusses the use of an interpreted language as a tool for dynamic configuration of distributed applications using CORBA components. We describe LuaOrb, a system based on the CORBA Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI), which provides Lua programs with easy access to CORBA servers and allows these servers to be dynamically modified. Using LuaOrb, the Lua console itself becomes a tool for reconfiguration. LuaOrb uses a structural sub-typing model, so that only correctly typed connections are accepted. We also discuss possible forms for prescribing a reconfiguration, and their relation to LuaOrb.

## 1 Introduction

Component-based programming has been receiving a lot of attention and is frequently considered the successor of object oriented programming. A fundamental point in the concept of component is the separation between interface and implementation. To use a component, it is necessary only to know its interface — implementations need not and, in general, should not be public. This allows client programs to remain valid even when an implementation of an invoked component is substituted for a new version.

CORBA is the most widely accepted component model for open systems, with several commercial and academic implementations [20]. However, not much attention have been given to the problem of managing change in CORBA based applications. The need for dynamically evolving 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, as is the avoidance of service interruption.

In CORBA, interfaces are specified using an *IDL* (Interface Definition Language). Specifications written in IDL are typically compiled into stubs (in the client side) and skeleton programs (in the server side). Modifications in the structure of the application (client-side) or in the implementation of components (server-side) require re-compilation and service interruption.

In this work, we discuss how an interpreted language can be used to add flexibility to CORBA-based programming. We present a set of tools which use alternative, dynamic mechanisms for client and server implementation and discuss how these tools can be used to support dynamically evolving applications. Tools and examples are based on the extension language Lua and its binding to CORBA, called LuaOrb.

In the next section, we define interpreted languages, and discuss their advantages and disadvantages. We describe Lua and LuaOrb in Section 3, and in Section 4 show how they can be used to solve some classic configuration problems. In the last section we draw some conclusions.

## 2 Using an Interpreted Language

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.

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 [16] and Lua [8].

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).

In the preceding paragraphs we have been using the term "configuration" in the sense of tailoring an application to specific needs and environments. However, this task is not so different in its goals from the configuration of distributed applications. Rather, these are different points in a continuum of complexity levels. Therefore, it makes sense to explore the use 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.

In the context of configuration of applications, the fact that interpreted languages allow changes to take place with no need for re-compilation becomes specially interesting, as it allows distributed applications to be dynamically modified with no need for service interruption. The interactivity offered by an interpreted language also fits in well with configuration requirements for testing and prototyping: the programmer can use a console to test components and partial configurations directly, avoiding the need for test stubs.

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 component-based applications, components may be coded in compiled, efficient languages such as C, with the interpreted language acting only as a a flexible connection element. In this role, the performance penalties interpretation impose may be overruled by time spent on communication or input/output calls. 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, which are in some ways similar to exceptions, and may be appropriately handled for program flexibility.

## 3 LuaOrb

LuaOrb is a binding between CORBA and the language Lua, an interpreted language developed at PUC-Rio [8, 5]. Lua is an *extension* language, implemented as a library. With its API, it is very easy to call Lua functions from C code, as it is very easy to register C functions to be called from Lua code.

The CORBA standard [15] provides communication facilities to applications in a distributed environment. All communication between CORBA objects is mediated by the *Object Request Broker* (ORB) [15, 21]. A client can interact with the broker through stubs or through its *Dynamic Invocation Interface*.

OMG IDL is a technology-independent syntax for describing object interfaces. Typically, specifications written in IDL are compiled into client stubs (called simply stubs) and server stubs (called skeleton programs). The client program is directly linked to the stub. The server program must implement the methods declared in the skeleton.

This approach, used in most current language bindings, such as C++, Java, and Smalltalk, requires clients to 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. Servers must be recompiled after any modification to their interface or implementation.

The CORBA architecture offers two mechanisms which allow programs to circumvent this need for re-compilation. On the client side, the Dynamic Invocation Interface (DII) is a generic facility for invoking any operation with a run-time–defined parameter list. On the server side, the Dynamic Skeleton Interface (DSI) is an interface for writing object implementations that do not have compile time knowledge of the type of the object they are implementing [22]. DII, DSI, and other CORBA services, like the Trading and Naming Services [21], offer the basic mechanisms to support a dynamic distributed object environment. In dynamic environments, applications can find out and incorporate new components at run-time. Besides, components can be extended on the fly to incorporate new functionality, and applications can be adapted, also on the fly, to component changes. This level of flexibility is very important to some applications, such as desktops and operating systems [21], network management tools [2], and cooperative applications [12].

Because CORBA allows the discovery of the object type and methods at run time, it is possible to implement mechanisms of *dynamic typing*. The interface repository (IR) offers support for applications to browse 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.

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.

However, using DII and DSI are not trivial tasks, and involve querying and constructing complex structures. Because C and C++ are static typed languages, with no automatic memory management, a program must build a dynamic call step by step, with explicit calls to create the parameter list, to set each parameter type and value, and so on [13].

It is therefore clear that DII and DSI can be interesting in many cases, but their use represents a difficult task with the current existing support. The next sections present LuaOrb, a binding of Lua to CORBA that offers a more suitable support for developing open applications, and allows the management of change in CORBA based application. LuaOrb interacts with Lua only through the official Lua API, and its implementation required no changes to the language.

### 3.1 LuaOrb's Client Binding

Lua is a dynamically typed language, wherein all type checking is done at run time. Variables and functions have no type declarations. Objects in Lua (also called *tables*) have no classes; each object can have any kind of methods and instance variables. The mapping between Lua and CORBA has tried to respect this flexibility of Lua. In that way, it should be possible to use CORBA objects in the same way as other Lua objects.

Because CORBA objects were 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.

LuaOrb uses *proxies* to represent CORBA objects in a Lua program. A proxy is a regular Lua object that uses fallbacks, a reflexive mechanism of Lua, to change its default behavior [7]. When a Lua program calls a method from a proxy, the fallback intercepts the call and redirects it to the LuaOrb binding. Then, LuaOrb 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. 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.

To illustrate the use of LuaOrb, we will use the following IDL interface:

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

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

To create a proxy of a distributed object that implements the `foo` interface, we use the `createproxy` function:

```
a_foo = createproxy("foo")
```

The `createproxy` function has an optional second argument, which 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)
x = 20
if a_foo:test() then
  x = foo2:div(x,2)
end
```

The first line creates an object with two fields, `author` and `title`, initialized with the given strings. The second line calls the `add_book` method from object `a_foo` (the colon is the Lua operator for method calls). When `a_book` is used as argument in this call, LuaOrb automatically tries to convert it to the IDL structure `book`; because `a_book` has the correct fields with the correct types, the conversion succeeds (otherwise, LuaOrb would signal an error). The conversion works recursively, so a list of Lua tables can be automatically converted to an array of records, for instance.

Because of its dynamic nature, the type conversions allow many changes in an IDL interface not to affect its uses in Lua, such as reordering and removing of structure fields, and changes between IDL types with similar representations in Lua, such as `short` and `long`, or `array` and `sequence`. The type system that emerges from these properties is one in which *structural compatibility* is enforced [19].

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 simply prefixes the method name with `deferred`; for instance,

```
a_foo:deferred_test()
```

A deferred method call returns a handler that can be used later 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 the `completion_event` function, 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.

### 3.2 LuaOrb's Server Binding

The server binding of LuaOrb allows Lua objects to be used as CORBA servers. For that, it uses CORBA's Dynamic Skeleton Interface (DSI). The basic idea of the DSI is to implement all calls to a particular object (which we will call a *dynamic server*) by invocations to a single upcall routine, the *Dynamic Implementation Routine* (DIR). This routine is responsible for unmarshalling the arguments and for dispatching the call to the appropriate code. The most frequent application of DSI has been to implement bridges between different ORBs [23]. In this context, the dynamic server acts as a proxy for an object in some other ORB.

Similarly to the client binding, LuaOrb also uses proxies for the server binding, but reverses the roles: while in the client side each CORBA object has a Lua object as its proxy, in the server side each Lua object that acts as a CORBA server is represented by a proxy, which is a C++ object. This object handles all method requests for the original Lua object through the DSI upcall routine. For each call received, the upcall routine unmarshalls the arguments, converts them to appropriate Lua values, calls the corresponding Lua method, and converts back the results from Lua to CORBA. The conversions follow the same rules used by the client binding, but with opposite directions.

To see a very simple example of a server written in Lua, let us consider the following IDL interface:

```
interface listener {
  oneway void put(long i);
}

interface generator {
  void set_listener(listener r);
}
```

Our `generator` object generates numbers (random numbers, for instance), and for each number it generates it calls the `put` method from its listener. We can write and use a listener object in LuaOrb as follows:

```
l = { put = function (self, i)
              print(i)
            end
    }

gen = createproxy("generator")
gen:set_listener(l)
```

In the first assignment, we create a Lua object with a single method, called `put`. Then, we create a proxy for a generator, and set that Lua object as its listener. When LuaOrb detects that the formal parameter type is an IDL `listener`, while the actual argument is a Lua object, it automatically creates a proxy for this Lua object, enabling it to work as a CORBA server.

LuaOrb also offers an IDL interface for remote server update:

```
interface LuaDSIObject {
  readonly attribute Object obj;
  void InstallImplementation (string opname, string luaCode);
}

interface ServerLua {
  LuaDSIObject Instance (CORBA::InterfaceDef intf);
}
```

To create a new instance at a dynamically extensible server, a client first invokes method `Instance`. The single parameter of this method is a reference to an interface definition in the interface repository. This reference is used by `Instance` to retrieve information about the attributes and methods of the new object.

The new object is returned as the attribute `obj` inside a new `LuaDSIObject` (the object in this attribute is, in fact, the proxy for the new Lua object). The manager client can then invoke the method `InstallImplementation` to install or modify each of the objects methods.

To illustrate the use of this interface, we will change our last example, so that the listener is created remotely, in another machine. We assume that the user has already created bindings to the Interface Repository (`IR`) and to a remote `ServerLua` (`Server`):

```
-- creates a new object with type 'listener'
l = Server:Instance(IR:lookup("listener"))
-- creates a new method called 'put'
l:InstallImplementation("put",
                        "function (self, i) print(i) end")
gen = createproxy("generator")
gen:set_listener(l.obj)
```

The presented sequence of commands can be interactively issued from a simple LuaOrb console, which thus becomes a powerful tool for interactive dynamic configuration.

### 3.3 Access to the Interface Repository

The interface repository (IR), defined as a component of the ORB, provides dynamic access to object interfaces. The IR is itself a CORBA object, and can thus be accessed through method invocations. In general, these methods can be used by any program, allowing the user, for instance, to browse through available interfaces. However, the IR is specially important for DII and DSI, the dynamic interfaces of CORBA. DII allows programs to invoke CORBA servers for which they have no precompiled stub. In order to build dynamic invocations, the program must possess information about available methods and their parameters; the interface repository provides this information. On the server side, DSI allows a server to handle requests for which it has no precompiled skeleton. Again, the correct signature of these requests must be obtained from the interface repository.

LuaOrb provides a library, called LuaRep, to simplify access to the IR. LuaRep makes extensive use of Lua data description facilities. Through the use of fallbacks, LuaRep reflects the repository information into Lua objects, so that any operation on these objects is automatically converted to the equivalent operation on the repository. This allows the Lua programmer to manipulate the repository information by accessing regular Lua objects. The reflexivity implemented by LuaRep allows not only queries to the repository, but updates as well.

The possibility of dynamically updating the IR extends the flexibility obtained with LuaOrb. It allows a manager client to install not only new implementations for existing interfaces, but also unforeseen services in the server, by first adding their definitions to the interface repository.

## 4 Configuration of Applications

In languages like Conic [10] and Darwin [11], 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]. For instance, filtering structures can be easily built from components such as the one shown below:

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

In the CORBA model [21], interfaces are described in IDL by method signatures, instead of input and output ports. Method signatures can declare input and output parameters, besides optional return values. Thus, a method signature describes a bidirectional flow of information.

The two approaches work at different levels of abstraction. In Darwin, the interface of a component contains complete information about all the communication in which the component is involved. Part of the communication activity of a component typically relates to offered services, while some of it may relate to a specific implementation. As an example, a component that implements a parser may or may not communicate with a file server, according to the way it stores temporary information. In CORBA, only offered 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 for implementation purposes. On one hand, this gives the programmer much less control over the construction of a complete application. On the other hand, 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.

In some cases, however, the need for objects that offer specific methods may be part of a CORBA object interface. In object oriented programming, an important concept is that of a *listener* or a *callback* object. The specification of an object may define that it must pass on produced data to a consumer object or register the occurrence of certain events at a registrar object. In such cases, since communication is part of specification, and not implementation, the interface of the object must reflect the existence of these communication partners, for instance by providing methods for defining who the partner is.

Recently, languages like Darwin and OLAN [3] are being called *architecture description languages*. This reflects the emphasis on a top-down approach, where support is given for the specification of application structure, and tools are offered for the translation of this specification to module skeletons, which must be filled in by the programmer. In component based programming, a bottom-up approach to software development may be more appropriate. The goal here would be, given the need for a new application, to compose it from existing components keeping new code to a minimum. Since the focus is on using existing components, an important role of the configuration language is to act as an intermediate between components not originally designed to be used together.

In our system, the description of the application is a script written in LuaOrb. The possibility of invoking arbitrary CORBA servers with no need for previous declarations allows the set of CORBA components which compose the application to dynamically evolve. This evolution is implicit: There are no explicit linking or unlinking operations.

Lua and the LuaOrb framework provide an environment that allows the application to evolve in yet another way. Through the `ServerLua` interface, a specific component can be modified or extended from a Lua program running on a different machine. This program may be part of the distributed application or may be executing as an independent activity.

In the next subsections, we discuss some classes of change management in CORBA based applications. Other configuration examples are presented in [18].

### 4.1 Interfaces and Application Configuration

The goal of our first example is to discuss how the concepts of "provisions" and "requirements", which are present in classic configuration languages, can be modeled using Lua and CORBA. The example we use is the primes sieve of Erathostenes discussed for Darwin in [9] and [11]. In this application, a process feeds a stream of numbers to a pipeline of *filter* processes. Each filter prints the first number it receives and subsequently filters out multiples of that number from the stream of numbers.

The interface definition of the filter component used in Regis is basically the one we presented previously. [11] describes two ways in which a pipeline can be created with these components. In a static configuration, a fixed number of filter components is instantiated, and each `right` port is connected to a `left` port of another filter through a `bind` command. In a dynamic configuration, each filter component is dynamically instantiated when its predecessor in the pipeline uses its `left` port. In what follows, we will discuss only the dynamic configuration, since our main focus here is on application reconfiguration.

The IDL for the filter component in CORBA, *primeFilter*, is as follows: The `provider` interface plays the part of an input channel. It gets new items from calls to its single method `put`.

```
interface provider {
  oneway void put(long i);
}
```

The `requirer` interface models an output channel, that is, a component requiring an input channel. Its single method, `setprovider`, connects it to an input channel, so that, for each new item the interface needs to output, it calls the put method of its provider.

```
interface requirer {
  void setprovider(provider r);
}
```

Finally, the primeFilter interface inherits from these two other interfaces. Therefore, it acts as a filter, receiving numbers through its put method, and passing them to its provider.

```
interface primeFilter: provider, requirer {
}
```

The specific behavior of this filter is quite simple: It stores the first number it receives. After that, it filters out all numbers that are multiple of that first number; only non-multiple numbers pass through it. An implementation of primeFilter is sketched in Figure 1.

```
class primeFilter {
  provider prov;
  long myprime;
  primeFilter () { myprime=0; }
  void setprovider (provider n) { prov=n; }
  void put (long c) { if (!myprime) myprime = c;
                      else if (c%myprime) prov.put(c); }
}
```

**Fig. 1.** Implementation of primeFilter.

Figure 2 presents the configuration of the sieve program in LuaOrb. The program prints out primes from 2 to N. The Lua object assigned to variable lnf is responsible for the dynamic creation of new filter components. After the declaration of the lnf object, a generator object is created. This component simply creates a stream of numbers from 2 to N, to be fed to the filter pipeline.

In this example, as in the next, we use an appServer object to create new instances of a component. The appServer object represents an *application server*, a service that registers interface implementations and supports requests both for running implementations and for new instances of a given interface.

When the generator invokes put for the first time, it will activate the Lua object lnf, which is its current provider object. Execution of lnf:put results in the creation of a new primeFilter object, which is then set as the provider for the generator. Subsequent invocations of put in the generator will invoke this new filter object. This new filter object has lnf as its provider component, so

```
-- define listener for creation of new filters
lnf = { put = function (self, p)
                print(p)
                local next = appServer:createCorba("primeFilter")
                next:put(p)
                next:setprovider(self)
                last:setprovider(next)
              end
      }
--  start application
gen = appServer:createCorba("generator")
gen:setprovider(lnf)
gen:createStream(2,N)
```

**Fig. 2.** Configuring the sieve of primes.

the same behavior will be repeated each time a `primeFilter` invokes `put` for the first time.

This example also illustrates how LuaOrb allows Lua and CORBA objects to be manipulated homogeneously. Calls to the `setprovider` method defined in the `requirer` interface receive both kinds of objects as arguments in different moments.

Note that in the proposed scheme, each filter component is aware only of the existence of a `provider` object, to which candidates to primes must be passed on. The Lua script creates these new filters, as it identifies new primes, in a way that is completely transparent to the C++ components.

The paradigm used in this example is data driven. The generator is the main active object; filters act as passive objects, activated when a new datum is available. We could build this same example with a result-driven paradigm, wherein we change the roles of providers and requirers: Providers would have a single method `get`, which returns a new item, and requirers would call this method from their providers whenever they need a new item. The possibility of these two dual approaches illustrates the flexibility offered by CORBA and LuaOrb.

### 4.2 Event-driven applications

An important class of distributed applications are event-driven applications, where actions must be associated to the occurrence of specific events. This section discusses how CORBA-based event-driven applications can be coded in LuaOrb.

To illustrate the proposed programming style, we will get an example used in [17] to present Glish, a language that supports communication through events. In this example, a simple distributed application is composed out of two components: one measures some data and the other displays the data. The monitor component announces it has produced new data by generating an event. The response to this event is to activate the display component. Using Lua and CORBA, we can again resort to passing a listener object as an argument to the monitor; the generation of an event is then modeled by an invocation of this object.

Possible IDL interfaces for these two components are as follows:

```
interface monitor {
  void measure(long intervalsize, supervisor s);
};

interface display {
  void newData(double data);
}
```

As described in Section 3.1, LuaOrb provides deferred invocations, which allow an asynchronous style of programming. The Lua configuration program can invoke `measure` asynchronously; it can then proceed its execution and the Lua supervisor object will be called when needed. The configuration code for this example is as follows:

```
-- supervisor object
s = { newData = function (nd)
                  d::newData(nd)
                end
    }
d = appServer:createCorba("display")
m = appServer:createCorba("monitor")
m:deferred_measure(5,s)
...
```

In this style of component linking, all events pass through the configuration program. As discussed in [17], this increases the cost of communication, but also buys flexibility. Suppose that, depending on the value of the data, there is a need to transform it before passing it on to the display component. This can be trivially programmed in `s::newdata`. According to the kind of transformation that is needed, it could either be done by calling a third component, or in the Lua program itself.

When performance is vital, Glish provides point-to-point links between programs, through an explicit `link` statement. Unlike Glish, LuaOrb does not need extra mechanisms for such links. In LuaOrb, this facility follows directly from the fact that Lua objects and CORBA objects are interchangeable. So, we can pass the display component directly as an argument to function `measure`:

```
m = appServer:createCorba("monitor")
d = appServer:createCorba("display")
m:deferred_measure(5,d)
```

### 4.3 Component Modification

In the previous section we have discussed how new components can be dynamically added to an application and how the flow of data can be dynamically redirected between components. In this section we discuss changes to a component itself.

For this discussion we do not use an example from the literature, since few works discuss runtime changes to components. Instead, we will use an example drawn from network management. Figure 3 shows IDL definitions for a component that provides TCP-related information. The definitions represent information provided by the SNMP standard management information base, MIB-II [14]. The tcpConnTable attribute contains, at any moment, a description of each current TCP connection. Each entry in this table is a tcpConnEntry structure, which contains the local and remote ports and IP addresses, and the state of the connection.

```
struct tcpConnEntry {
        long   tcpConnState;
        string tcpConnLocalAddress;
        long   tcpConnLocalPort;
        string tcpConnRemAddress;
        long   tcpConnRemPort;
};

typedef sequence<tcpConnEntry> tcpConnTable;

interface tcpGroup {
        readonly attribute tcpConnTable connections;
        tcpConnTable FilterConnections();
        ...
};
```

**Fig. 3.** IDL for network management component

A common problem in network management is the availability of enormous quantities of raw data that must be processed by the management application. To circumvent this problem, the `tcpGroup` interface provides a `FilterConnections` method, which returns only the connection table entries that satisfy some criteria.

Using LuaOrb, this interface can be implemented by a component that can be modified dynamically, allowing the administrator to establish different filtering criteria with no need for re-compilation. For instance, first we can create this component with a trivial filter, that returns the whole table of TCP connections:

```
trivialf = "function () return connections end"
```

```
dsiobj = serverlua:Instance(IR:lookup("tcpGroup"))
dsiobj:InstallImplementation("FilterConnections", trivialf)
```

Later, we can redefine this implementation, to select only connections that use local port 79, say:

```
newf = [[
  function ()
    filtertab = {}
    i = 0
    while connections[i] do
      if connections[i].tcpConnLocalPort==79 then
        tinsert(filtertab, connections[i])
      end
      i = i+1
    end
    return filtertab
  end
]]

dsiobj:InstallImplementation("FilterConnections", newf)
```

The `newf` variable holds a string with the code of the new filter function. (Lua uses the double square brackets to delimit literal strings that span several lines.) After installing this new implementation, we can use it to retrieve the table of connections:

```
tcpG = dsiobj.obj
interestingConnections = tcpG:FilterConnections()
```

## 5 Final Remarks

It seems natural here to relate the problem of component-based configuration with the more general problem of software configuration, which deals with customizing software to different needs and environments.

The use of a textual configuration file for software customization reflects a declarative, static style, and can be compared to the use of a declarative architecture description language in component-based applications. Work in architecture description languages, based on a more declarative linking and unlinking style, has given much focus to the problem of checking configuration consistency [6,4] Parsers that check required properties can also be easily built for configuration files.

This is in fact only another application of the properties of static checking. As is always the case, a static solution represents robustness, avoiding many run time errors, but limits the flexibility that can be provided.

With the use of an extension language for software configuration, the configuration file becomes a program, with the associated difficulties for automatic checkers, but also with the flexibility that only a programming language can provide. Besides, execution errors due to eventual inconsistencies can many times be captured and handled in an interpreted language.

In this work, we proposed extending the use of extension languages to the configuration of component-based applications. No explicit linking or unlinking operations are provided: On the one hand, this means no consistency verifications are possible; on the other hand, it means that the evolution of the application is controlled with a full programming language. As shown in the examples discussed in this work, this results in a great deal of flexibility, allowing not only different patterns of component interaction to be dynamically defined but the components themselves to be easily modified. These facilities are specially important if component-based programming is to fulfill its promise in the fields of software reuse and rapid prototyping.

## References


1. G. Andrews. Paradigms for process interaction in distributed programs. *Computing Surveys*, 23(1), 1991.
2. Bela Ban. *A Generic Management Model for CORBA, CMIP and SNMP*. PhD thesis, University of Zurich, 1997.
3. L Bellisard, F. Boyer, Riveill, and J. Vion-Dury. Systems services for distributed application configuration. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 53–60, Annapolis, MD, 1998. IEEE.
4. P. Feiler and J. Li. Consistency in dynamic reconfiguration. In *International Conference on Configurable Distributed Systems*, pages 189–196, Annapolis, MD, 1998. IEEE.
5. Luiz H. Figueiredo, Roberto Ierusalimschy, and Waldemar Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, December 1996.
6. K. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Third International Conference on Configurable Distributed Systems*, pages 62–69, Annapolis, MD, 1996. IEEE.
7. R. Ierusalimschy, R. Cerqueira, and N. Rodriguez. Using reflexivity to interface with CORBA. In *International Conference on Computer Languages 1998*, pages 39–46, Chicago, IL, 1998. IEEE, IEEE.
8. R. Ierusalimschy, L. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6), 1996.
9. J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *IEE Software Engineering Journal*, 8(2):73–82, 1993.
10. J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in Conic. *IEEE Trans. on Software Engineering*, SE-15(6), 1989.
11. Jeff Magee, Naranker Dulay, and Jeff Kramer. A constructive development environment for parallel and distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5), 1994.
12. 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.
13. Marco C. Martins, Noemi Rodriguez, and Roberto Ierusalimschy. Dynamic extension of CORBA servers. In *Euro-Par'99*, Toulouse, France, 1999.

14. K. McCloghrie and M. Rose. Management information base for network management of TCP/IP-based internets: MIB-II, 1991. RFC 1213.
15. Object Management Group, Inc., Framingham, MA. *The Common Object Request Broker Architecture and Specification; Revision 2.0*, jul 1995.
16. J. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.
17. V. Paxson and C. Saltmarsh. Glish: a user-level software bus for loosely-coupled distributed systems. In *1993 Winter USENIX Technical Conference*, 1993.
18. N. Rodriguez, R. Ierusalimschy, and R. Cerqueira. Dynamic configuration with corba components. In *International Conference on Configurable Distributed Systems*, pages 27–34, Annapolis, MD, 1998. IEEE.
19. N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *Sigplan Notices*, 28(8), 1993.
20. D. Schmidt. Corba products, 1999. `http://www.cs.wustl.edu/~schmidt/corba-products.html`.
21. Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.
22. Peter Wayner. Objects on the March. *Byte*, january 1994.
23. N.M.S. Zuquello and E.R.M. Madeira. A mechanism to provide interoperability between ORBs with relocation transparency. In *Proc IEEE Third International Symposium on Autonomous Decentralized Systems (ISADS'97)*, pages 53–60, Berlin, Germany, 1997.