Dynamic Reconfiguration of Component-based Applications

Thaís Batista [1,2]
[1] Informatics Department – Dimap
Federal University of Rio Grande do Norte – UFRN
Campus Universitário – Lagoa Nova
59.072-970 - Natal - RN - Brazil
thais@inf.puc-rio.br

Noemi Rodriguez [2]
[2] Computer Science Department
Catholic University of Rio de Janeiro – PUC-Rio
R. Marquês de São Vicente, 225 – Gávea
22.453-900 - Rio de Janeiro - RJ - Brazil
noemi@inf.puc-rio.br

Abstract

Component based programming is a current trend in the development of software. The application is created using components and binding their interfaces appropriately at the configuration level. This is especially interesting for applications that, for availability reasons, claim for dynamic reconfiguration. This paper describes an approach for dynamic reconfiguration of applications based on CORBA components running in an environment called LuaSpace that is composed by the dynamically typed language Lua and a set of tools based on Lua. Components, scripts and glue code are the elements that form an application expressed in Lua. LuaSpace provides support for both programmed and ad-hoc reconfiguration. Although our work focuses at configuration level, LuaSpace also handles components updating.

1. Introduction

The advantages of constructing systems in two levels - configuration and programming - are well recognized in the software community [2, 13, 21]. According to this approach, an application is created using components and binding their interfaces appropriately at the configuration level [13].This is especially interesting for evolution because the maintenance of the software is favored when distinguishing the application structure from the programming details. It is possible to unplug components and reconfigure the needed parts. Since availability is a key issue for many applications such as electronic commerce, financial systems and critical mission systems, the running systems must be reconfigured without interruption of their execution. It is also interesting to support *automatic reconfiguration*, in which changes occur in an application as the result of some computation performed by the system, without the user performing any explicit configuration management actions.

Frameworks for component interoperability are playing an important role in component based application development because they offer support for working with heterogeneous components despite differences in language and in the execution platform. The main aspects provided by this approach, which make interoperability possible, are interfaces standardization according to an Interface Definition Language (IDL) and standard communication protocols for different platforms. The CORBA model [20] has drawn special attention as a framework for interoperability because it is independent of language and manufacturer and provides easy access to components with transparent distribution. However, CORBA, like other component models, does not have facilities to describe the global organization of an application [5].

In order to plug components together at the configuration level, a *configuration language* may be used [13, 2, 11]. The goal of configuration languages is to allow the specification of the system and the definition of component interaction. Components are divided in an interface and an implementation part, and the configuration program uses only information available in interfaces. Typically, systems based on configuration languages use components whose interface describes *required* and *provided* communication objects [13]; communication objects are similar to channels. The configuration program then basically determines which components will be used in an application and links output to input channels, defining the flow of information. CORBA interfaces are quite different from the ones generally used in configuration systems. In the first place, CORBA is an object oriented architecture, and all communication is based on method invocations. Besides, CORBA interfaces describe strictly a component's behavior, ie, the provided services, and contain no information at all about implementation. As CORBA components are to be used as black boxes, a component's interface contains no information about other components which may be invoked as part of an object's implementation. However, it is many times interesting to introduce some flexibility in such invocations, allowing the required objects to be determined at runtime. Although the configuration paradigm [13] supports component interaction, it fails to address interconnection between components that do not declare the required services.

We believe interpreted languages can be an important tool in this context. Interpreted languages typically include features such as absence of declarations and dynamic typing. Thus, interpreted languages do not impose compiled interconnection between components. If an interpreted language is used to write the configuration program, changing the program does not imply recompilation. It is thus possible to quickly redefine functionality in a running system without shutting it down. Moreover, the interactive style of an interpreted language can offer a console to be used as a tool for reconfiguration purposes.

What we are proposing is to use an interpreted language with a conventional procedural programming model as a configuration tool. In this approach, there are no interconnection declarations: ordinary method invocations make the configuration program act as a glue between components. Conditional structures can thus be used to reconfigure an application, and even the set of participating components, according to the conditions occurring at runtime. The expressive power of a procedural and interpreted language also allows writing glue code to adapt components with incompatible interfaces so that they work together.

In this work, we discuss the flexibility and dynamism that results from the use of an interpreted language in this context. Specifically, we use the programming language Lua [12], an extension language developed at PUC-Rio. Besides the generic advantages of using a procedural interpreted language, Lua offers a set of tools that together provide a powerful environment – *LuaSpace* – for handling dynamic reconfiguration. One of these tools is a binding between Lua and CORBA, called LuaOrb [9], based on CORBA's Dynamic Invocation Interface (DII). This binding allows dynamic access to CORBA objects available at remote servers to be accessed dynamically, as if they were ordinary Lua objects. Moreover, it uses CORBA Dynamic Skeleton Interface (DSI) to permit dynamic installation of new objects, via Lua, in a running server. Automatic reconfiguration is also supported by LuaSpace.

The paper is organized as follows: in section 2, we present a brief overview of the infrastructure which forms the basis of our work, including the Lua language, LuaOrb, and the concept of a generic connector. In section 3, we present the component interconnection model and illustrate it, using a classical example from the literature [13]. In section 4, we outline the reconfiguration tasks used by most systems and discuss how our environment supports these tasks. Finally, in section 5, we present the conclusions.

2. LuaSpace

LuaSpace, our working environment, is composed by a set of tools which offer support for dynamic reconfiguration. Now, we briefly present these tools.

2.1. The configuration Language Lua

Lua [12] is an interpreted extension language developed at PUC-Rio. Lua is dynamically typed: variables are not bound to types, although each value has an associated type. Lua includes conventional aspects, such as syntax and control structures similar to those of Pascal. However, it also has several non-conventional features, such as the following:

- Functions are *first-class* values, which means they can be stored in variables, passed as arguments to functions, and returned as results. Functions may return several values, eliminating the need for passing parameters by reference.
- Lua *tables* implement associative arrays, and are the main data structuring facility in Lua. Tables are dynamically created objects and can be indexed by any value in the language, except `nil`. Many common data structures, such as lists and sets, can be trivially implemented with tables. Tables may grow dynamically, as needed, and are garbage collected.
- Lua offers several reflexive facilities. One simple example is the `type` function, which allows a program to determine the type of a value. *Fallbacks* are Lua's most generic mechanism for reflection. Several situations in which the interpreter would intuitively generate an error can be captured by a programmer-defined function, called a fallback. Examples of such situations are calls to non-existent functions and indexing a value that is not a table.

2.2. LuaOrb

LuaOrb [9] is a binding between the language Lua and CORBA that, as usual, defines mappings between Lua and IDL types. The unusual aspect is that this mapping is based on Lua's dynamic typing and on CORBA's Dynamic Invocation Interface (DII) [20], which allows use of arbitrary CORBA objects at runtime (without compiled stubs). On the server side, LuaOrb uses the CORBA Dynamic Skeleton Interface (DSI) [20] to permit dynamic installation of implementations written in Lua. Most systems use the static approach to define a mapping to CORBA, requiring recompilation when a change takes place either at the client or at the server.

On the client side, LuaOrb explores the dynamic features of Lua to offer a way to access CORBA objects available at remote servers exactly like any other Lua object. This is done transparently and at runtime. To use a CORBA object, a Lua proxy to the CORBA object must be created, using the `createproxy` function and defining the required interface and the object reference. Function `createproxy` returns a Lua object that represents the object. Using Lua's fallback mechanism, LuaOrb detects the operations applied to the proxy and transforms them into remote operations. DII provides different invocation modes and LuaOrb allows the programmer to use them. The deferred synchronous mode is only available via DII. In this mode, the client can call a method and continue its execution without waiting for the method's completion. Later it can poll the server for the result. To use this mode, the prefix `deferred_` must be included before the operation name. The method call returns a handle that can be used to poll the method result. LuaOrb also provides a function for observing the completion of a method called in deferred mode, `completion_event`. This function has two parameters, a handler returned by a deferred synchronous method call and a function that will be invoked upon the method completion with the method results as parameters.

On the server side, LuaOrb explores the facility provided by DSI offering a way to implement and extend CORBA objects using Lua. Lua objects can act as CORBA servers. Clients can access these objects using either stubs or DII. In a strategy symmetric to that of the client side, a proxy to the Lua object is created. The object can be dynamically installed at a server, displaying the behavior of an usual CORBA object. LuaOrb defines an IDL interface, `ServerLua`, that allows a client to remotely instantiate new objects and to define their Lua implementation [8].

2.3. Generic Connector

The generic connector [3] mechanism dynamically selects components for executing services required by an application. The term "Connector" derives from the idea of an executable entity that controls the interaction among components [10], providing a way to isolate the implementation of a component from the interconnection aspects in order to create an application. The "generic" designation is used to specify that such a connector is not supposed to bind components with specific, previously defined characteristics (as do the connectors of Software Architecture works [19, 7, 1]), but instead, it looks for any component that is able to provide the required service. Exploring this approach, an application can be configured using a set of services without being aware of the specific components that offer these services. The components will be selected on-the-fly by the generic connector.

The generic connector is a Lua object represented by a proxy created by function `generic_createproxy()`. In the configuration program, after creating the proxy, the service invocation can be defined as follows: `proxy_name:service(params)`. When this invocation is executed, the generic connector intercepts it and invokes a method whose function is to search, in a standard repository (a catalogue service such as the trader or name service), a component to execute the service required. Then, the generic connector builds the solicitation and activates the service using LuaOrb.

The role of the generic connector goes beyond that of object search services that are simply repositories of information about components and offer functions for querying these repositories. The generic connector allows application to use services of undefined components in the same way they use services of known components.

The generic connector is a tool for interconnecting and reconfiguring running applications since, for each call of the same service, different components can be selected to execute it. As a consequence, with the use of the generic connector, it becomes impossible to distinguish the tasks of configuration and of reconfiguration in the configuration program. The component search process proceeds in a transparent fashion. In this way, the generic connector gives runtime support for automatic insertion of components in an application.

3. Interconnection Model

The global structure of an application is described using a Lua program, which acts as a CORBA client that uses LuaOrb to access CORBA components. In [4] various possible interconnection schemes for a Lua configuration program are discussed. Here, we will focus only on the event-driven programming style to describe component interconnection in a configuration program. To illustrate this model we will use a classical example from the literature [13], the patient monitoring system (PMS). This system is composed by *nurseunit* and *bedmonitor* components. Each *bedmonitor* scans sensors attached to a patient and displays an alarm when there are data outside specified ranges. The *nurseunit* module receives alarms and, sometimes, sends a query to request the status of a patient. Figure 1 shows the IDL interfaces of this system.

LuaSpace provides support for event-driven programming through the `completion_event` function, available with LuaOrb. This function is used to define a handler for the event associated to the completion of a *deferred synchronous* method call. The deferred synchonous invocation

```
struct data{
    string name;
    short pressure_max, pressure_min;
    short temperature, pulse;
}
interface bedmonitor{
   data status();
   bedmonitor monitor();
}
interface nurseunit{
   void alarm(in bedmonitor);
}
```

Figure 1. Patient Monitoring System IDL Interfaces.

mode, provided by CORBA, allows the client to proceed asynchronously, after invoking a method, and later poll for its completion. Deferred method calls can be specified by prefixing a call with the `deferred_` keyword. In PMS, the `monitor` method continuously monitors patient data until it reads any data outside the accepted limits. When this happens, the method returns. Thus, method termination corresponds to an event: the detection of some abnormal state.

In the example configuration programs, 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. The implementation of this service is not in the scope of the current work.

```
nurse = appserver.create("nurseunit")
pat = 1
while pat<5 do
  p = appserver.create("bedmonitor",pat)
  completion_event(p:deferred_monitor(),
                             nurse:alarm)
  pat=pat+1
end
```

Figure 2. An event-driven configuration program.

Using the interfaces described in Figure 1, a Lua configuration program is shown in Figure 2. In this example, a CORBA method is passed as the second argument of function `completion_event`. Method `alarm` of *nurseunit* will be automatically invoked when method `monitor` returns. This illustrates a way of linking components directly.

Figure 3 shows a configuration program using the generic connector to represent a component that offers method `alarm`. In this example, the generic connector is interconnecting components: When method `monitor` terminates, the component whose `alarm` method to be invoked will be determined by the generic connector.

```
pat = 1
while pat<5 do
    p = appserver.create(``bedmonitor'', pat)
    c = generic_createproxy()
    completion_event(p:deferred_monitor(),
                                c:alarm)
    pat = pat + 1
end
```

Figure 3. An event-driven configuration program with the generic connector

4. Reconfiguration

The possibility of modifying a program while it is running, termed *dynamic reconfiguration*, has been the subject of many works in the last years [13, 21, 11]. However, using an interpreted and procedural language for reconfiguring applications, as we do, introduces some important changes in the concept of reconfiguration, which we discuss in this section.

The first aspect is related to the CORBA component model which does not provide informations about required services in the component interface. These information are related to the implementation part. (Sometimes, however, 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, or by declaring the partner as a parameter in the signature of specific methods.)

The second one is related to the use of an interpreted and procedural configuration language, in this work, the Lua language. This introduces a different style of configuring an application where explicit linking and unlinking commands are not necessary. The procedural model is used to describe an application. As a consequence there is no rigid distinction between the configuration and reconfiguration of an application. The simple use of conditional (if) or iteration (while) command in the configuration program implies dynamic reconfiguration. Through an interactive console is possible to build and modify configuration programs. As

Lua is a dynamically typed language, it is not necessary to declare, previously, the component instances that will be used in a program. New components can be selected dynamically according to runtime conditions.

In addition, changes to an application can take place at the configuration level (insertion or removal of component or connections) and at the component level (new implementation, addition of functions, etc). Most work on configuration languages focuses only on modification at the configuration level [21, 15] (One exception is the SOFA/DCUP architecture [17] that handles dynamic updating of components in runtime applications.) In contrast, LuaSpace allows modification in the two levels. Although our work is concerned with the configuration level, the flexibility offered by Lua tools to modify components easily should not be set aside.

In [21, 11, 22], dynamic reconfiguration is classified according to the moment of its definition as either a *programmed* or an *ad-hoc* reconfiguration. In the former, changes are defined at system design time and are triggered by the system when some conditions are satisfied. In the latter (also known as unplanned change [22]), reconfigurations are established by the user at unpredictable time when the application program is in execution. While the former is important for automatic reconfiguration, the latter is suitable for software maintenance. In general, systems address only one kind of reconfiguration, for instance, in [21, 2] only programmed reconfigurations are supported and in [22] the major focus is on unplanned change. LuaSpace offers support to implement both programmed and ad-hoc changes.

Programmed Reconfiguration

For programmed reconfiguration, the Lua conditional commands can be used to establish the conditions that determine reconfiguration in the application source code. In this way, reconfiguration points are explicitly defined by the programmer.

The generic connector introduces the possibility of automatic reconfiguration: in this case, reconfiguration is transparent to the configuration program. This issue is related to the notion of self-organizing architecture [14], in the sense that the generic connector minimizes the explicit management necessary for evolution. The generic connector also satisfies the idea, stated in [11], of implementing generic dynamic reconfiguration. In [11] this is supported by a generic symbol that is a reconfiguration variable whose value is a name of a configuration object found at runtime according to the verification of conditions defined in the configuration program. At runtime, whenever this symbol is referenced, the object currently bound to it is used in its place. The term "generic" is used because no concrete object is bound to this symbol at design time. As discussed in section 2, the generic connector has a similar behaviour.

With the generic connector, LuaSpace offers support for reconfiguration in two levels. In the first one, the programmer determines the components that execute the required services. In the other one, more automatic, the programmer specifies only the services that compose an application and the generic connector hides the complexity of searching for the appropriate components. This is especially useful for accomodating programmers of different technical levels. Advanced programmers can customize the generic connector facility, for instance by programming a strategy for choosing a component in the case of multiple components with the required service having been found. Others programmers can ignore the complexity of such situations, using the Lua console for basic configuration tasks and relying on the generic connector in its default behaviour or as customized by more senior application managers.

Ad-hoc Reconfiguration

Ad-hoc reconfiguration can be programmed interactively through the Lua console. This tool offers the programmer a way to have control over the application by directly interacting with the system. Using the Lua console, it is possible to write an application from scratch and also to modify an existing one.

Unplanned change can yield problems of application consistency. This problem is addressed in some works that, in general, follow the configuration paradigm [13, 22, 6]. The consistency model is based on static type checking and on a clear structural view of the application. These features do not fit in Lua dynamic style that promotes flexibility in lieu of static checking. Static solutions can avoid many runtime errors, but on the other hand limit the flexibility that can be provided [18]. Thus, our work does not support the semantics of reconfiguration to verify the validity of the changes.

4.1. Reconfiguration Tasks

Independently of reconfiguration classes, the operations that are typically provided by systems for reconfiguration are the same [6]: addition and removal of components and links. We next discuss how these tasks are supported in LuaSpace.

The addition of new components in an application can be done: via the Lua console, automatically by the generic connector, or using conditional commands to program when the component must be inserted in the configuration. If the component does not yet exist, it is necessary to use LuaOrb's mechanisms for dynamic installation of an object in a server. After this, the new object can be inserted in the configuration program.

```
interface log{
   void get_data (in long id, in sta-
tus); /* records the patient's data*/
   seqdata history (in long id); /* re-
turns the patient's history */
}
```

Figure 4. IDL Interface of Log component

To illustrate the dynamic installation of an inexistent object in a server we will define a `log` component to periodically register patient data. Figure 4 presents the IDL interface of the component. The code of the dynamic installation of method `history` is shown in Figure 5.

```
-- create a new object with type 'Log'
l = Server:Instance(IR:lookup(''Log''))
-- creates a new method called 'history'
l:InstallImplementation(''history'',
        ''function (number)''
        ''   data = tabpatient[number]''
         ''  return data''
        ''end'');
```

Figure 5. Dynamic Installation of Log Component

The potential for reconfiguring an application is an intrinsic feature of the generic connector. Consider the example illustrated in Figure 3. Suppose that method `alarm` is offered by components *nurseunit* and *medicalunit*. In a first execution, the generic connector will select one of them, for example, the *nurseunit*. In a subsequent execution, if *nurseunit* is not available, the generic connector will select the other component (*medicalunit*). This scenario exemplifies the automatic reconfiguration provided by the generic connector.

Two actions are involved in the task of removing components from a given configuration: removing proxies for CORBA objects and removing CORBA objects themselves. Lua's garbage collection mechanism removes automatically CORBA proxies when they are not being used anymore, so component removal occurs implicitly. To remove CORBA objects it would be necessary that CORBA maintained information about the object use and that a garbage collection mechanism removes unused objects. This is not currently supported by CORBA, and it would have to be developed as an independent service.

Other reconfiguration models include commands to insert and remove links between components. Again, since in Lua we use no explicit linking commands, unlinking is also implicit. As discussed in section 3, a binding between two components can be established either indirectly, by having the Lua program call one of them with data received from the other, or directly, by passing a component as an argument to a method call. In either case, the simple choice of a new component to invoke or to pass as an argument implies in the "unlinking" of the old one. The example we discussed above, about the automatic reconfiguration offered by the generic connector, illustrates that in a first execution the implicit linking will proceed between the *bedmonitor* and the *nurseunit* components, and in a subsequent execution, the linking will be between the *bedmonitor* and the *medicalunit* components.

```
-- modify method 'history'
l:InstallImplementation(''history'',
        ''function (number)''
        ''   data = tabpatient[number]''
        ''   data_pressure = data[pressure]''
         ''  return data_pressure''
        ''end'');
```

Figure 6. Dynamic Modification of a Component Method

The Lua environment also supports updating of component interface and updating of component implementation. In general, configuration languages do not handle the latter because they do not deal with implementation aspects. However, it is interesting to regard the environment as an "application manager" instead of merely as a composer. In this context, LuaOrb provides a great deal of flexibility by allowing the programmer to install new services and to modify the implementation of existing ones.

For example, we can change at runtime the implementation of the method `history` that will only return patient's pressure data. Figure 6 illustrates this issue. Although the implementation of a component was redefined, the configuration program remains unchanged.

Figure 7 illustrates the capability for application reconfiguration present in the Lua program through the conditional command and the generic connector mechanism. The availability of these facilities is well suited for applications such as PMS, where time of response is a critical factor, no matter which component will be selected to execute the required service. In the example, if the condition statement is satisfied, the generic connector will select a component that provides the `urgency` method.

In this example, function `handlealarm` is a handler for the event signaled by the `monitor` method. This function triggers the `alarm` method of a *nurseunit* and then invokes the monitoring method once again; this illustrates a common situation, in which, after handling an event, the program redefines a handler for the next occurrence of the same event.

In addition, in function `handlealarm` the total number of alarms is counted, and, if it exceeds the value LIMIT, a special action is taken (invoking a unit that provides the `urgency` method). This is in contrast to the conventional configuration languages that link components directly with no knowledge about the data which flows between the components. In general, different messages may need different services. In a Lua configuration program, the data can be analyzed to decide the component to be called. As discussed in [16] this adds a great flexibility to the configuration. Comparison of the configuration presented in figures 2 and 7 illustrates well the difference between linking components directly and having control flow through the Lua interpreter. Although the performance gain may be important in some cases, in the second alternative the configuration program loses much of its dynamic character, since, once it has established a connection, it has no knowledge of system behavior.

```
nurse = appserver.create("nurseunit")
pat = 1
while pat<5 do
  p = appserver.create("bedmonitor",pat)
  completion_event(p:deferred_monitor(),
                             handlealarm)
  pat=pat+1
end

alarmcnt = 0

function handlealarm (p)
  nurse:alarm(p)
  alarmcnt = alarmcnt+1
  if alarmcnt>LIMIT then
    c = generic_createproxy()
    c:urgency(p)
  end
  completion_event(p:deferred_monitor(),
                             handlealarm)
end
```

Figure 7. Configuration program with reconfiguration capability.

5. Conclusions

Although the component models offer support to work with heterogeneous and distributed components, they lack facilities to describe the global organization of an application. In this work we have argued that a configuration based environment can offer a good support to overcome this issue. Using an interpreted procedural configuration language, it is also possible to address an important requirement of current applications: the need for dynamic reconfiguration.

The configuration program style we presented in this work is compatible with the CORBA programming model, which does not declare the required services in the component interface. In our model, binding is implicit because there is no explicit linking or unlinking command. In general, works opt for extending the CORBA model in order to define explicit binding. Although we agree with the importance of explicit binding in some contexts, we consider that reusability is favored if we use the CORBA model in its original form. Futhermore, our approach preserves the strict separation of specification and implementation which is one of CORBA's most important concepts.

We discussed how dynamic reconfiguration of component based applications are supported by LuaSpace exploring the advantages of Lua and its tools. One of the main advantage of this environment is to offer support for reconfiguration in two levels. In the first one, the programmer defines the components that offer the required services. In the other, the programmer declares only the services that compose an application and the generic connector selects automatically the proper components to execute them. In addition, programmed and ad-hoc reconfiguration are possible since a change is either a result of some action realized by the system or a result of some external action which modifies the configuration. Our system can be regarded as self-organizing, in the sense defined in [14], since there is no explicit configuration management.

Our environment is based on Lua and on LuaOrb, but the same support it provides for reconfiguration could be offered by other environments, since it is based on an interpreted language with dynamic typing and on a binding between this language and the component model that uses the dynamic interface of the component model.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie-Mellon University, 1997.

[2] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. Durra: a structure description language for developing distributed applications. *Software Engineering Journal*, pages 83–94, March 1993.

[3] T. Batista, C. V. F. Chavez, and N. Rodriguez. Conector genérico: Um Mecanismo para Reconfiguração de Aplicações Baseadas em Componentes. In *Third Ibero-American Workshop on Software Environments and Requirement Engineering (IDEAS'00)*, Cancun - México, April 2000. To be published.

[4] T. Batista and N. Rodriguez. Dynamic Interconnection of Component-Based Applications. Technical Report 15/99, Catholic University of Rio de Janeiro – PUC-Rio, 1999.

[5] L. Bellissard and M. Riveill. Constructions des applications réparties. Ecole Placement Dynamique et Répartition de Charge, Juillet 1996.

[6] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems - ICCDS*, pages 35–42, Annapolis, Maryland, May 4-6 1998.

[7] J. Bishop and R. Faria. Connectors in Configuration Programming Languages: are they necessary? In *Third International Conference on Configurable Distributed Systems*, pages 11–18, Annapolis, Maryland, May 6-8 1996.

[8] M. Catunda, N. Rodriguez, and R. Ierusalimschy. Dynamic Extension of CORBA Servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, Aug 1999. Springer-Verlag. LNCS 1685.

[9] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. In *International Symposium on Distributed Objects and Applications (DOA'99)*, pages 362–371, Edinburgh, Scotland, September 1999. OMG, IEEE Press.

[10] S. Ducasse and T. Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of ESEC/FSE'97, LNCS 1301*, pages 483–500, 1997.

[11] M. Endler. A Language for Generic Dynamic Configurations of Distributed Programs. In *XII Brazilian Symposium of Computer Networks - SBRC*, pages 175–187, Curitiba - PR, 1994.

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

[13] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4):424–435, April 1985.

[14] J. Kramer and J. Magee. Self Organizing Software Architectures. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 35–38. ACM Press, 1996.

[15] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *IEE Software Engineering Journal*, 8(2):73–82, March 1993.

[16] V. Paxson and C. Saltmarsh. Glish: a user-level software bus for loosely-coupled distributed systems. In *1993 Winter USENIX Technical Conference*, 1993.

[17] F. Plásil, D. Bálek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 4-6 1998.

[18] N. Rodriguez and R. Ierusalimschy. Dynamic reconfiguration of CORBA-based applications. In J. Pavelka, G. Tel, and M. Bartošek, editors, *SOFSEM'99: 26th Conference on Current Trends in Theory and Practice of Informatics*, pages 95–111, Milovy, Czech Republic, 1999. Spinger-Verlag. (LNCS 1725).

[19] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*, pages 2–10, Annapolis, Maryland, May 6-8 1996.

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

[21] M. Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 111–118, Annapolis, Maryland, May 4-6 1998.

[22] A. Young and J. Magee. A Flexible Approach to Evolution of Reconfigurable Systems. In *Proceedings of the First International Workshop on Configurable Distributed Systems*, pages 152–163. IEE, 1992.