A Dynamic Approach to Combine Components and Aspects

Fabrício de Alexandria Fernandes, Thais Batista
Federal University of Rio Grande do Norte
Informatics Department
Campus Universitário – Lagoa Nova – 59.072-970 - Natal - RN
e-mail: fabricio@consiste.dimap.ufrn.br, thais@ufrnet.br

Abstract

In this paper we discuss the use of a dynamic approach to combine components and aspects in order to form a CORBA-based application. We use AspectLua, an extension of an interpreted and dynamically typed language – Lua, which supports aspect-oriented programming. We describe the use of AspectLua in the context of a CORBA-based environment. AspectLua explores the reflexive features of Lua and the powerful of its small syntax and allows the dynamic definition of aspects and it also offers support for dynamic weaving. The difference of AspectLua to the traditional aspect-oriented languages is the fact of being interpreted and dynamically typed – this allows the insertion/removal of components and aspects in an application at runtime.

1. Introduction

Component-based software development is a current trend in software engineering because it promotes the idea of reusing components. It focuses on composing applications by assembling prefabricated pieces of software named components [16]. In this context, the development process is split in two levels: at the *component level* there are the components that offer some services. At the *configuration level*, an application is specified through a program that contains the components that provide the application functionality, as well as the interconnection between the components.

Following the idea of separation of concerns, configuration-based programming emphasizes the need of decoupling concerns related to coding computational components from those related to binding components and creating a new application. Thus, configuration based programming separates the composition between components from the component implementation that makes part of this composition.

Another research area that has been gaining popularity as a promising approach to software engineering is Aspect-Oriented Programming (AOP) [8] because it emphasizes separation of concerns and promotes the benefits of modularity. By decoupling concerns related to coding computational components from those related to crosscutting concerns (in several works also named *non-functional aspects*), AOP improves reusability of software.

In this paper we address the integration of component-based software development and aspect-oriented programming by discussing how non-functional aspects of a CORBA application can be dynamically integrated with the CORBA components at the configuration level. In order to handle this integration, we adopt an interpreted-based approach where the aspect weaving occurs at runtime. We aim to handle application dynamic reconfiguration where it is possible to define and redefine aspects at runtime as well as to insert and to remove components in the configuration.

In general, aspect-oriented approaches are static – aspect code and functional modules are mixed at compile time (static weaving). Besides, a special compiler is needed to combine the aspect code with the base code. Although this strategy avoids type mismatches, it imposes many restrictions on application evolution. In contrast, in this work we use a dynamic approach, where crosscutting concerns can be inserted and removed from an application configuration at runtime.

As stated in [4], static weaving is not suitable for components reusability because the aspects to be integrated with the components depend on the usage-context. They will be determined case by case for each individual usage of the component. Besides, it can be influenced by conditions that occur at runtime.

In this work we present AspectLua – an extension of an interpreted and procedural language: Lua [10] – to handle aspect-oriented programming. We also present the integration of AspectLua in an environment [1] for the development of CORBA-based applications that uses Lua to glue CORBA components. In this environment the application configuration is written in Lua and can be composed by components implemented in any language that has a binding to CORBA. The environment, named LuaSpace, offers a set of tools based on Lua that offers strategic functions to facilitate the development of CORBA-based applications. One of the tools is LuaOrb, a binding between Lua and CORBA based on CORBA´s dynamic invocation interface (DII) that provides dynamic access to CORBA components exactly like any other Lua object.

AspectLua allows the dynamic definition of aspects and it also offers support for dynamic weaving. The combination of CORBA components and aspects are defined at the configuration level. This approach follows a common use of a scripting language: as a configuration language to glue components. In this work, it is used to glue components and aspects. We choose the Lua language because it is small, dynamically typed and because it provides facilities for extending its behavior without modification in the underlying interpreter. Such facilities are explored in the definition of AspectLua. We argue that this introduces a different style for aspect-oriented programming where dynamism is a key issue, weaving is done at runtime and both components and aspects can be inserted and removed from the application at runtime. We also choose Lua because the availability of an environment, based on Lua, for CORBA development.

Similarly to our work, there are other works [7][5] that extend scripting languages to support aspect-oriented programming and also support dynamic weaving. In the context of scripting languages, the reason we chose Lua is due to the insertion of this work in a larger project that investigates the flexibility that an interpreted language can add to a component model.

This paper is organized as follows. Section 2 presents the basic concepts of aspect-oriented programming and also the Lua language. Section 3 presents AspectLua and discusses its support for defining applications by combining components (the functional code) with aspects. Section 4 comments about related works. Finally, section 5 contains the final remarks.

2. Basic Concepts

2.1. Aspect-Oriented Programming

Aspect-Oriented Programming emphasizes the need to decouple concerns related to coding computational components from those related to non-functional aspects of an application. This decoupling is often supported by proposing the use of different languages for programming these two types of activities [13]. For instance, Java programmers write the

functional code in Java and the aspect code in a Java extension for aspect-oriented programming such as AspectJ [8]. A special compiler does the integration between Java and AspectJ programs.
Although there is no consensus about the terminology of the elements that makes part of aspect-oriented programming, we refer in this work the terminology used in AspectJ because it is the most traditional aspect-oriented language. *Aspects* are the elements designed to encapsulate crosscutting concerns and take them out of the functional code. *Join Points* are well-defined points in the execution of a program. *Advices* define code that runs at join points. They can run at the moment a joint point is reached and before the method begins running (before), at the moment control returns (after) and during the execution of the joint point (around).

2.2. Lua

Lua is an interpreted extension language developed at PUC-Rio [10]. It is dynamically typed: variables are not bound to types although each value has an associated type. Lua includes conventional features, such as syntax and control structures similar to those of Pascal, and 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. Lua stores all elements in tables generically as *key-value* pairs. 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. Other reflexive facility is the *next* function, which allows retrieving all fields of a table. *Metatables* 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 metatable. Examples of such situations are calls to non-existent functions and indexing a value that is not a table. This mechanism is commonly used to create resources not provided by the language.

3. AspectLua

Lua does not offer support for aspect-oriented programming. In order to include this issue, AspectLua extends the language by including support for aspect-oriented programming. It is developed exploring the Lua extensions facilities. Due to the Lua reflexive features it is not necessary to modify the Lua interpreter to support AspectLua. AspectLua explores the powerful of Lua tables and uses them to the definition of aspects, pointcuts, join points and advices. These elements are defined using the Lua features and no special commands are needed. In the current implementation, AspectLua supports method-intercepting calls.

3.1. Architecture

The development of applications using AspectLua follows the idea presented in Figure 1.

The **component level** contains the application base code (CORBA components). They implement the main functionality of the application regardless of the non-functional issues. They can be written in any language that has a binding to CORBA (C, Lua, Python and so on). Although Lua is much better suited as a glue language, it can also be used to implement components.
The **aspects definition level** contains the non-functional aspects of the application. It is defined using the AspectLua constructions.
The **application configuration level** specifies the components and the aspects that compose the application. This combination is defined in Lua. This level is represented by the *Configuration Program*.

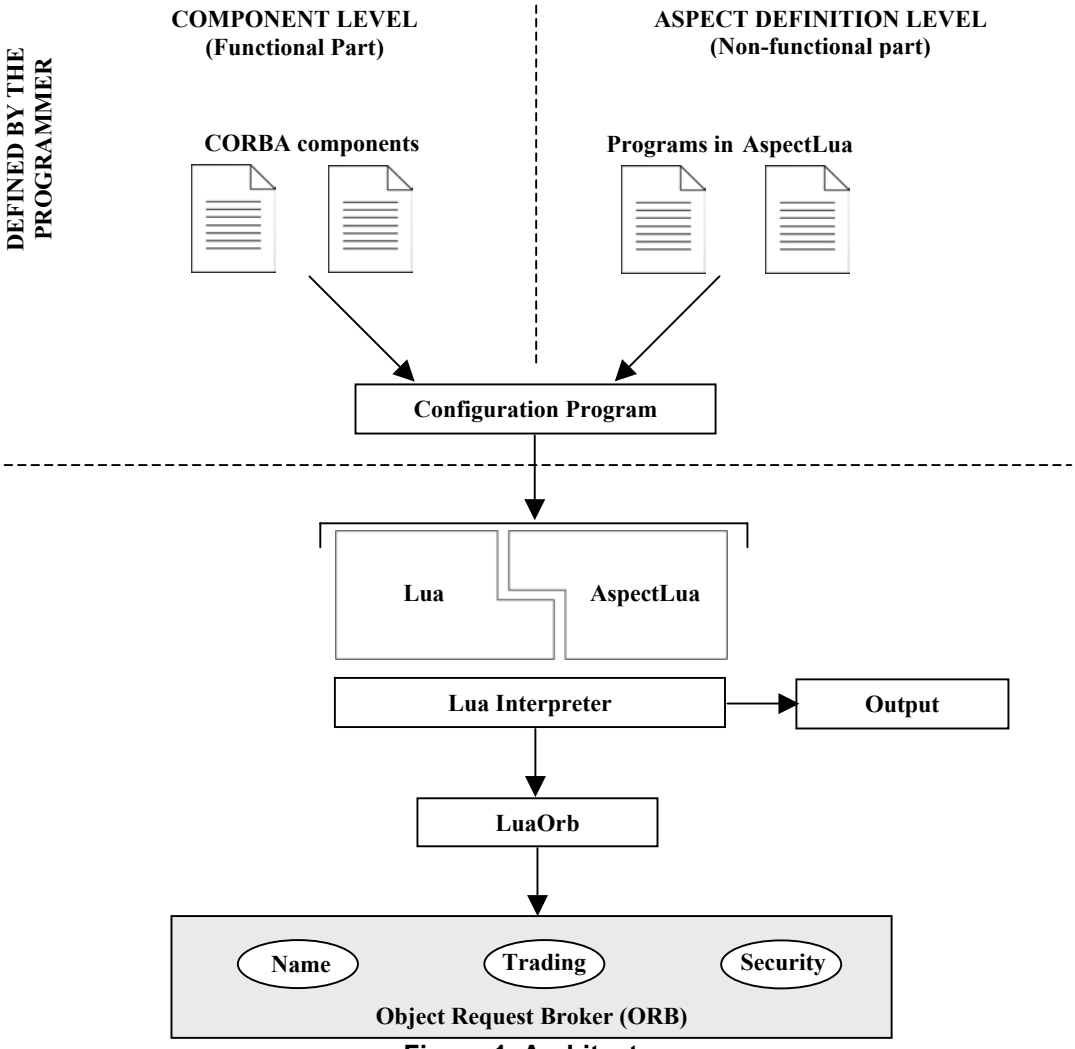

Figure 1. Architecture.

The two codes (components and aspects) are executed by the Lua interpreter that invokes AspectLua when executing commands associated with aspect programming. This is done transparently for the programmer and supported by the reflexive facilities offered by Lua. In contrast to the traditional aspect-oriented programming, in this approach there is no need of a special compiler to join aspect with the functional modules, the interpreter does this mixing at runtime. Thus, it is possible to support dynamic reconfiguration – new components and aspects can be selected dynamically according to runtime conditions.

3.2. AspectLua Elements

Simplicity and flexibility are the main features of the Lua language. AspectLua follows the same idea and preserves these features. No special commands are needed. It offers only two Lua instructions to the definition of aspects, pointcuts, join points and advices. Aspect creation is based on the definition of a Lua table and the elements that compose the aspect are passed as parameters of the table. This avoids that the programmer need to learn a new language in order to work with aspect-oriented programming. AspectLua defines an *Aspect* object that handles all aspects issues. This object defines a function to create a new aspect: the *new()* function.
After creating a new aspect, it is necessary to define a Lua table that contains the aspect elements (name, pointcuts and advices). Figure 2 illustrates the generic code to aspect definition. The first parameter is the aspect name. The second parameter is a Lua table that defines the pointcut elements: its name, its designator and the functions that must be intercepted. The designator defines the pointcut type (the current implementation of AspectLua only allows method-call-interception). The third parameter is a Lua table that defines the advice elements: its type (after, before, around) and the action to be taken when reaching the pointcut.

```
a = Aspect:new()
a:aspect({name = 'value'},
          {pointcutname = 'value',
           designator = ' value ',
           functionsName = value},
         {type = 'value', action = value})
```

Figure 2. Generic code to aspect definition

3.3. AspectLua Implementation

The implementation of AspectLua is divided in two parts. The first one supports the aspect creation. The second part implements the intercepting process that does functions interception to execute the aspect code.
Figure 3 shows the AspectLua internal code used to allow the definition of a type named Aspect.

```
-- Code that handles the definition of an Aspect.
Aspect = settag({}, newtag())

settagmethod(tag(Aspect), "index", function(t, i)
       return rawget(%Aspect, i) end)

-- Aspect Constructor
-- @param a Aspect name
function Aspect:new(name)
       return settag({ aspectName = name or '' }, tag(%Aspect))
end
```

Figure 3. AspectLua internal code that handles aspects definition

After creating an aspect type, the information passed as a parameter of the Lua table that represents the aspect is associated with such type. Figure 4 shows how AspectLua does this association.

```
-- AOP Constructions

-- Aspect definition
-- @param aspect Aspect Name
-- @param pointcut Table with the pointcut properties
```

```
-- @param advice Table with the advice properties
function Aspect:aspect(aspect, pointcut, advice)
       print('Creating the aspect...')
       %Aspect:new(aspect.name)
       %Aspect:pointcut(pointcut.name, pointcut.designator,
pointcut.functionsName)
       %Aspect:advice(advice.type, advice.action)
       interceptList = pointcut.functionsName
       intercept(interceptList, advice.type, advice.action)
end

-- Generic code to create a pointcut
function Aspect:pointcut(name, designator, functionsName)
       print('')
       print('About the pointcut...')
       print('Pointcut name: ' .. name)
       %Aspect.pointcutName = name
       print('Designator: ' .. designator)
       %Aspect.pointcutDesignator = designator
       for _, fn in functionsName do
             print('Function name to be watched: ' .. fn)
       end
       %Aspect.functionsName = functionsName
end

-- Generic code to create an advice
function Aspect:advice(type, action)
       print('')
       print('About the advice...')
       print('Advice type defined: ' .. type)
       %Aspect.adviceType = type
       print('Advice action defined!')
       %Aspect.adviceAction = action
       print('')
end
```

Figure 4. AspectLua internal code that implements information about the aspect

The interception is done via a function, named **intercept**, which redefines the functions determined in the aspect pointcut. Figure 5 illustrates the **intercept** function. This function deals with the parameters passed when creating an aspect. It receives the list of functions passed in the *functionsName* parameter. This list is compared with the set of functions that compose the component program. When a function present in the list is found in the component program, the **intercept** function defines a new function that contains the original function and the function passed as the *action* parameter of the aspect (it represents the code that should be executed by the aspect – the advice itself). This advice code is inserted in the modified function at the local (after or before) determined by the *type* parameter of the advice. Then, the modified function is inserted in the program replacing the original one.

```
-- Function to intercept functions with specified actions.
--@param list List (table) of functions names
--@param adType Advice type (after or before)
--@param adAction Advice action to be executed
function intercept(list, adType, adAction)
       table = globals()

       for _,fn_name in list do
             local name = fn_name
             print(name)
             local old_fn = table[name]
             if adType == 'before' then
                   table[name] = function( ... )
                                 %adAction()
                                 return call(%old_fn, arg)
                                 end
             elseif adType == 'after' then
                   table[name] = function( ... )
                                 temp = call(%old_fn, arg)
                                 %adAction()
                                 return temp
```

```
                                    end
                end
        end
end
```

Figure 5. AspectLua internal code of the intercept function

4. Using AspectLua in a CORBA-based Application

To illustrate the use of AspectLua, a simple example of a CORBA banking application was developed. The main purpose of the application is to offer functions to access a banking account. As an additional feature, security functions are used by the application. Figure 6 shows the code of the CORBA server, written in Lua, mixed with the security code.

```
-- Application example.
bal = 0
source_hello = {

        hello = function()
                print("Hello")
        end,

        gettotal = function()
                return bal
        end,

        settotal = function(self,amount)
                bal = bal+amount
        end
}

source_server = lo_createservant(source_hello,"IDL:HelloWorld:1.0")
writeto("hello.ref")
write(source_server:_get_ior())
writeto()

print("Server created")
```

```
LOrbSecLevel2:create_domain("/")
LOrbSecLevel2:create_domain("/Dimap")
LOrbSecLevel2:create_domain("/CCET")


LOrbSecLevel2:set_required_rights("/Dimap","IDL:HelloWorld:1.0","hello","SecAllRi
ghts","ug");

LOrbSecLevel2:set_required_rights("/CCET","IDL:HelloWorld:1.0","hello","SecAllRig
hts","m");

LOrbSecLevel2:set_required_rights("/CCET","IDL:HelloWorld:1.0","gettotal","SecAll
Rights","m");
LOrbSecLevel2:grant_rights("PrimaryGroupId","room","gsm","/Dimap")

print("Attributes set")
```

Figure 6. CORBA banking application server

Figure 7 shows the client code of the banking application.

```
if not LuaOrbCfg.getIFR() or LuaOrbCfg.getIFR():_non_existent() then
   error("The Interface Repository is not available.")
end


seccur_proxy = lo_createproxy(readIOR("./hello.ref"),"IDL:HelloWorld:1.0")
print("Server created")

seccur_proxy:hello()
seccur_proxy:settotal(5)
print(seccur_proxy:gettotal())
```

Figure 7. CORBA banking application client

The server code, illustrated in Figure 6, contains the security functions that implement the access rights. These functions should be implemented as an aspect because they do not take part of the application main requirement.
Exploring the AspectLua support, the code can be separated and the additional function can be implemented as an aspect. Figure 8 illustrates the server code with the basic functionalities of a banking application. Figure 9 shows the aspect code with the security function. The client code remains the same illustrated in Figure 7.

```
source_hello = {
       hello = function()
              print("Hello")
       end,

       gettotal = function()
              return bal
       end,

       settotal = function(self,amount)
              bal = bal+amount
       end
}

bal = 0

source_server = lo_createservant(source_hello,"IDL:HelloWorld:1.0")
writeto("hello.ref")
write(source_server:_get_ior())
writeto()

print("Server created")
```

Figure 8.Server Code

```
-- Code for creating an Aspect.
-- A definition of an aspect is based on the code described in aspectLua.lua
-- The functions represents the code for the advice.

-- Function that creates the required rights
function rights()
       LOrbSecLevel2:create_domain("/")
       LOrbSecLevel2:create_domain("/Dimap")
       LOrbSecLevel2:create_domain("/CCET")

       LOrbSecLevel2:set_required_rights("/Dimap","IDL:HelloWorld:1.0","hell
          o","SecAllRights","ug");
       LOrbSecLevel2:set_required_rights("/CCET","IDL:HelloWorld:1.0","hello
          ","SecAllRights","m");
       LOrbSecLevel2:set_required_rights("/CCET","IDL:HelloWorld:1.0","getto
          tal","SecAllRights","m");
       LOrbSecLevel2:grant_rights("PrimaryGroupId","room","gsm","/Dimap")

       print("Attributes set")
end

-- Some tests
a = Aspect:new()
a:aspect({name = 'SecurityAspect'} ,
         {name = 'securityFunctions', designator = 'call',  functionsName =
           { 'lo_createservant' } },
         {type ='after', action = rights})
```

Figure 9. Aspect Code – Security Functions

The code of Figure 9 creates an aspect named *SecurityAspect*, with a pointcut named *securityFunctions* that determines the *method-call* interception of the *lo_createservant* function. The *rights* function must be executed *after* the invocation of *lo_createservant*.
Since functions are first-class values in Lua, they can be passed as a parameter of a Lua table. This introduces a great flexibility in defining the actions to be taken by aspects.
Although we adopted the traditional linguistic structure of aspect-oriented programming, this is quite different using Lua. It is not necessary to introduce special commands in

AspectLua to express the aspect elements - we explore the Lua data structuring facility (tables). Since Lua is interpreted and dynamically typed, the aspect elements are not static, they can be easily inserted, removed and modified via the Lua interactive console.
A Lua configuration file defines the files that contain the application base code and the aspect code. Figure 10 shows the configuration file that defines the banking application file and the server aspect file. Again, due to the dynamic style of Lua, the configuration program can also contain dynamic decisions. For instance, a different aspect code can be used according to conditions expressed at the configuration file and verified at runtime. This adds a great deal of flexibility to the configuration.

```
dofile("AspectLua.lua")
dofile("serverAspect.lua")
```

Figure 10. Configuration File

Finally, the Lua interpreter receives the configuration file and executes the application producing the output illustrated in Figure 11. This output shows that the *gettotal* method could not be invoked because the required rights to execute it, determined in the aspect code, were not satisfied by the client that invoked the method.

```
Server created
<LuaOrb> Error calling the method gettotal
<LuaOrb> IDL:omg.org/CORBA/MARSHAL:1.0 (0, maybe-completed)
```

Figure 11. Output

5. Related Works

There are some works that propose an aspect language that is built on top of a scripting language. AOPy [7] is an aspect-oriented language building on top of Python (Rossum 2003) that does not introduce any new language constructs to support aspect-orientation. The language used is Python itself. AOPy implements method-interception via a dynamic proxy-like system. AspectR [5] is an aspect-oriented language building on top of Ruby [17]. It supports aspect-orientation allowing wrapping code around existing methods in classes. Like AOPy, AspectR does not introduce any new language constructors to support aspect-orientation. It uses Ruby itself.
There are many similarities among these aspect-oriented languages and AspectLua: both are built on top of a scripting language, no new language constructs are needed and aspect weaving occurs at runtime. The main difference rely on the language itself: Lua is easier than Ruby and Python; Ruby and Python syntax are very similar while Lua syntax is smaller without loosing the flexibility commonly offered by scripting languages. Comparison between Python and Ruby are available at [11]. Another difference is the availability of Lua tools to support component-based applications and also to support dynamic reconfiguration of such applications. The discussion of the role of a scripting language to dynamic reconfiguration of component-based applications has been focus of previous research and it is described in [1].
LAC – Lua Aspectual Component [9] – is another Lua extension for aspect-oriented programming whose main goal is to support the idea of Aspectual Components [12] and combine the ideas of AspectJ and HyperJ. In order to support this combination, it defined a template with the following elements to support aspect-oriented programming: (1) a *class* represents a base component*;* (2) a *Collaboration class* represents an aspect*;* (3) the composition of classes and collaborations is defined by a *Connector*. LAC is quite different of AspectLua because its elements are defined in order to support the idea of Aspectual Components while AspectLua elements are defined following the traditional concepts of

aspect-oriented programming proposed by AspectJ. LAC imposes a template where components and aspects are defined by different styles of classes. AspectLua does not impose a template to components. Besides, it uses simple tables to represent aspects and its elements. The focus of LAC is the model that implements the idea of Aspectual Components. After defining this model, Lua was chosen to implement it because it is a lightweight language. In contrast, the focus of AspectLua is to explore Lua as an aspect-oriented programming language without introducing new commands or structure.
There are other works, also related to this work, which aim to combine components models and aspects via a configuration language. [14] focus on .NET platform and identify component configuration as an aspect. A XML-based description language is used to handle the configuration aspect. In contrast, we use a procedural programming language – Lua – to define aspects and the combination of aspects and components. This introduces more flexibility regarding to the aspect code. AspectWerkz [2] is a Java framework to support dynamic aspect oriented programming. It can be used in the development of J2EE applications. In order to support dynamic aspect weaving, the framework modified the loading mechanism of Java classes. Aspects definitions are done via a XML file or runtime attributes. There are some differences of this work to our work. First, it modifies the Java original way of loading classes in order to handle aspect weaving. Second, it can only be used in Java or J2EE applications. In contrast, AspectLua is integrated with a CORBA environment in which applications can be written in any language that has a binding to CORBA. Thus, our work is applied in a broader context than those of AspectWerkz. There is a project, named Enterprise Component Broker [3] whose main goal is to integrate EJB and AOP. It suggests the insertion of aspects in an EJB container. Since there is no one available publication in the home page of the Project, we do not have more details in order to give more comments about its strategy.

6. Final Remarks

In this paper we discussed the use of aspect-oriented programming in the context of an environment to the development of CORBA-based applications in order to compose applications using CORBA components and aspects. We proposed the use of an interpreted and dynamically typed language to support dynamic aspect weaving. Aspects are defined using AspectLua – a simple extension of Lua – and the integration of CORBA components and aspects are defined at a configuration level via a Lua program. The reflexive features of Lua allows that the Lua interpreter combine the two parts of the application. The interpreted approach makes possible to define aspects and components at runtime.
The flexibility of Lua introduces a different style to aspect-oriented programming where it is not necessary to use a different language to define aspects. Besides, it is not necessary to implement a special compiler or interpreter to mixing the components with the aspects. This interpreted approach allows application-specific customization at runtime.
In this work we explore the use of a scripting language as a configuration language to glue components and also aspects.
Other works have extended a scripting language to offer support for aspect-oriented programming. We have chosen Lua because it is small, easier to use than some scripting language and it provides reflexive mechanisms, which allows to extend the language. The availability of bindings between Lua and some component models [6] was one reason of our option to Lua. Using CORBA, base components can be written in any language that has binding to CORBA. In this environment the application is composed by CORBA components (the functional code) and aspects.

With the separation of the aspect code from the component functional code, the reuse of CORBA components is increased and it can be used in many contexts in combination with different aspects.

References

1. Batista, T. and Rodriguez, N. Using a Scripting Language to Dynamically Interconnect Component-Based Applications. Anais do VI Simpósio Brasileiro em Linguagens de Programação (SBLP'2002), SBC, pag. 180-194, ISBN 85-88442-20-5, Rio de Janeiro, RJ, Junho 2002.
2. Bonér, J. AspectWerkz: dynamic AOP for Java. http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf. 2003.
3. Blank G., Vayngrib G. Aspects of Enterprise Java Beans, Aspect-Oriented Programming (AOP) Workshop at ECOOP'98. 1998.
4. Brichau, J., De Meuter, W. and De Volder, K. Jumping Aspects. Position paper at the workshop Aspects and Dimensions of Concerns. ECOOP 2000, Sophia Antipolis and Cannes, France, June 2000.
5. Bryant A. and Feldt R. AspectR - Simple aspect-oriented programming in Ruby. Available at http://aspectr.sourceforge.net/. 2002.
6. Cerqueira, R., Cassino, C., and Ierusalimschy, R. Dynamic component gluing across different componentware systems. In International Symposium on Distributed Objects and Applications (DOA'99), pages 362-371, Edinburgh, Scotland. OMG, IEEE Press. 1999.
7. Dechow, D. Advanced Separation of Concerns for Dynamic, Lightweight Languages. In: 5th Generative Programming and Component Engineering. 2003.
8. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., and Ossher, H. Discussing Aspects of AOP. Communications of the ACM, Vol. 44, No. 10, pp 33-38, October 2001.
9. Herrmann S. and Mezini M. Combining Composition Styles in the Evolvable Language LAC. ASoC Workshop in ICSE. 2001.
10. Ierusalimsky, R., Figueiredo, L. H., and Celes, W. Lua – an extensible extension language. Software: Practice and Experience, 26(6):635-652. 1996.
11. Kaustub, D. and Grimes, D. A comparison of object oriented scripting languages: Python and Ruby. Available at http://www.cs.washington.edu/homes/kd/courses/pythonruby.pdf . 2001.
12. Lieberherr, K., Lorenz, D. and Mezini, M. Programming with Aspectual Components. Technical Report, NU-CCS-99-01. Available at: http://www.ccs.neu.edu/research/demeter/bibli/aspectualcomps. html. March 1999.
13. Papadoupoulos, G. and Arbab, F. Coordination Languages and Models, In: Advances in Computers. Academic Press. 1998.
14. Rasche, A. and Polze, A. Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003), Hakodate, Hokkaido, Japan, ISBN 0-7695-1928-8, IEEE Computer Society Press. May 2003.
15. Rossum, G. v. Python Reference Manual. Available at http://www.python.org/doc/current/ref/ref.html. 2003.
16. Szyperski, C. Component Software: Beyond Object-Oriented Programming. Addison-Wesley. 1998.
17. Thomas D. and Hunt A. Programming Ruby: A Pragmatic Programmer's Guide. Available at http://www.rubycentral.com/book/. 2000.