A Lua-based AOP Infrastructure

Nélio Cacho, Thaís Batista and Fabrício Fernandes

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Campus Universitário - Lagoa Nova - 59.072-970 - Natal - RN
{cacho - fabricio}@consiste.dimap.ufrn.br
thais@ufrnet.br

Abstract

In this paper we describe an aspect-oriented infrastructure to handle dynamic AOP based on the Lua language. This infrastructure is composed of AspectLua, a Lua extension that allows the declaration of aspects, and a meta-object protocol, LuaMOP, that unifies the introspective and reflective mechanisms provided by Lua. Aspects are defined in isolation using AspectLua and then they are weaved through LuaMOP. An important feature of AspectLua is to allow the association of aspects with undeclared elements of the application code (anticipated join points). Furthermore, it combines a range of features to make AOP easier and powerful.

Keywords: MOP, Reflection, AOP, Dynamic AOP, Lua, Antecipated Join Points

1. Introduction

Aspect-oriented programming (AOP) has been gaining attention due to its focus on the modularization of crosscutting concerns. In general, aspect-oriented approaches are static – aspect code and components (base code) are mixed at compile time (static weaving). In this case, 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 dynamic evolution. More recently some dynamic approaches have been proposed to support weaving at runtime. In general they are built on top of a scripting language such as Python [13], Ruby [14], and Smalltalk [22]. These dynamic weaving approaches allow aspects to be woven at runtime. However, they present some limitations. An important limitation is to restrict the specification of aspects join points to refer to existing elements of the base code. In some situations it is necessary to define the aspect code to application elements that will be dynamically inserted. This dynamic insertion can be done by the application or even by the aspect itself.

We address this problem by proposing the concept of *anticipated join points*. Anticipated join points are interception points for elements that have not yet been declared and loaded in the application program. The use of anticipated join points makes it possible to intercept an invocation to an undeclared method and to apply to it a specific action, such as, lazy loading a code. Anticipated join points are introduced in AspectLua [24] to avoid the need of loading the application code that contains a given join point before loading the aspect code regarding this join point. We consider that the lack of support for anticipated join points is an important limitation of most AOP approaches.

Another limitation of most AOP proposals is that they do not combine a set of features to make AOP easier and powerful: (1) insertion and removal of aspects at runtime;

(2) the definition of precedence order among aspects; (3) the possibility of using wildcards; (4) the possibility of associating aspects with undeclared elements (*anticipated join points*); (5) a dynamic weaving process via a meta-object protocol.

In this work we present an aspect-oriented infrastructure that handles aspect-oriented programming and addresses the limitations described above. The infrastructure is composed of: (1) *AspectLua* – an extension to the Lua language [7] to allow aspect definition; (2) *LuaMOP* [25] – a meta-object protocol (MOP) that provides an abstraction over the reflective features of Lua and allows application methods and variables to be affected by the aspect definition. The advantage of using a MOP as an underlying mechanism to handle dynamic weaving is that it allows non-invasive changes of the original application code. Aspects are defined in isolation using the Aspect class provided by AspectLua and then they are weaved through LuaMOP. AspectLua offers an abstraction to hide the complexity of the weaving process. For instance, the programmer can define an anticipated join point without knowing that LuaMOP implements an underlying mechanism, named *Monitor*, to support anticipated join points.

We have chosen the Lua language to support dynamic AOP because it is dynamically typed and it provides facilities for extending its behavior without modification in the underlying interpreter. Such facilities are explored in the definition of AspectLua and LuaMOP. 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 into and removed from the application at runtime. In addition, the Lua philosophy is to be simple and small. We aim to keep this philosophy in our AOP infrastructure.

Although some researchers do not associate the use of AOP with scripting languages because, in general, such languages are not intended to write large and complex software systems, we argue that the benefits of AOP target not only large and complex software systems but it also has an important role in embedded systems where the problem of composition is even harder. This type of system needs to maintain the application code small. Thus, separation of concerns is essential and AOP is a good technique to manage crosscutting concerns in embedded systems [18].

In addition, as Lua is also used in a CORBA-based development application environment [25], the AspectLua infrastructure presented in this paper is useful for applying AOP to the dynamic adaptation of CORBA-based application. It makes adaptation of component code possible as well adaptation of aspect code and of the overall application.

This paper is organized as follows. Section 2 presents the underlying concepts of this work: aspect-oriented programming, computational reflection and the Lua language. Section 3 presents the aspect-oriented infrastructure presented in this work to handle dynamic AOP: LuaMOP, AspectLua and the relationship between them. Section 4 presents a case study that applies the AOP infra-structure in the dynamic customization of an AOP-based middleware. Section 5 discusses about related work. Finally, section 6 contains the final remarks.

2. Basic Concepts

2.1. Aspect-Oriented Programming

Aspect-Oriented Programming emphasizes the need to decouple concerns related to components from those related to aspects that crosscut components in an application. Although there is no consensus about the terminology and the elements of aspect-oriented programming, we refer in this work the terminology used in AspectJ [10] because it is the most used aspect-oriented language. The elements that compose AOP are: aspects, join points, pointcuts and advice. *Aspects* are the elements designed to encapsulate crosscutting concerns and remove them from the application base code (components). *Join Points* are the elements of the component language semantics that aspect programs coordinate with [9]. Join points can represent data flows of the component program, runtime method invocations in the component program, and others. *Pointcuts* are sets of join points. The definition of pointcuts makes it possible to get methods arguments values, attributes, exceptions, etc. Pointcut designators pre-defined in the language itself are used for this purpose. The main designators are *call*, *get*, and *set*, which are related, respectively, to method call, and variable reading and modification. Pointcuts can also be defined by programmers on the basis of pre-defined designators. An *advice* defines the action that must be taken when a join point is reached. It acts on a pointcut and can be configured to act before (*before* advice), after (*after* advice), around (*around* advice) the joint point, and others.

The weaving process places together the code defined in the join points and the advice. Weaving can be done either at compile time or at runtime. In AspectJ and AspectC++ [3] weaving is done at compile time. A current version of AspectJ supports weaving at load time. Since new language constructs to handle AOP were added to the language syntax, a special compiler plays the weaver role in order to mix source code and aspects code. The outcome is a new version of the system including both codes. The other approach, which involves aspects weaving at runtime, will be detailed in the next sections.

2.2. AOP with Reflection

The introduction of dynamic aspects in a programming language depends on its support for recognizing join points and for dealing with advice insertion. The recognition and

introduction of new behaviors (advice) can be implemented using computational reflection.

Reflection [8] is the ability of a system to inspect and to manipulate its internal implementation. The separation of application functionality and the execution mechanisms provides support for reflection. This separation allows the existence of two levels to support reflection: *base-level* and *meta-level*. The base-level contains the application concerns. The meta-level contains the building blocks responsible for supporting reflection. These levels are connected by a causal connection to allow modifications at the meta-level to be reflected into corresponding modifications at the base-level. Thus, modifications at the application should be reflected at the meta-level. The elements of the base-level and of the meta-level are respectively represented by base-level objects and meta-level objects.

The access to the meta-level objects is provided by a meta-object protocol (MOP), which defines an interface that enables accessing the structure of a program (classes, methods, fields, etc) and inspecting the execution environment. Events that can have the semantics modified by the meta-objects include: object creation, sending and receiving messages, searching methods, setting and getting values in variables. Meta-objects are instances of meta-classes that define fields and methods to modify and to inspect the execution environment.

The introspection facilities provided by MOPs allow the recognition of join points. It also easily supports the dynamic insertion of advice that represent the aspect code to be combined with the application code.

2.3. Reflection in Lua

Lua is an interpreted extension language developed at PUC-Rio. It is dynamically typed, which means that variables are not bound to types. However each value has an associated type. Lua syntax and control structures are similar to those of Pascal. It also offers some non-conventional features, such as the following: (1) *Functions* are *first-class* values and they may return several values, eliminating the need for passing parameters by reference; (2) Lua *tables* are the main data structuring facility in Lua. Tables implement associative arrays, are dynamically created objects, and can be indexed by any value in the language (except nil). Lua stores all elements in tables as *key-value* pairs. Tables may grow dynamically, as needed, and are garbage collected.

Lua offers reflective facilities such as: *metatables* and the _G environment variable. *Metatables* allow modification of the behavior of a table. This is done via the definition of functions to be invoked in specific points during the execution of a Lua program. Each function defined, named *metamethod*, is associated with a specific event. When an event occurs, the function is invoked to handle such an event. The code of Figure 1 illustrates the use of *metatables*.

```
1 commontable = {x=10, y=20}
2 local metatb = {__index = function (t,k)
                              print(k) end}
3 setmetatable(commontable, metatb)
```

Figure 1: Metatable definition

In the code of Figure 1, line 1 defines the *commontable* table with *x* and *y* fields. Line 2 defines the *metatb* metatable. It will act upon the "*index*" event by printing the index of the element. On line 3, *metatb* is applied, via the *setmetatable* method, upon the *commontable* table. Thus, when *commontable* is indexed, as in the *print(commontable.x)* invocation, the *metamethod* will be invoked to print the element used as the index, in this case "x".

Another reflective feature is the _G environment variable. It describes all global variables of an application, including tables and functions. This variable is a table that can be manipulated as any other table of the environment. It is possible to insert, to modify, and to remove variables and functions of the execution environment. The code illustrated in Figure 2 shows an example of a variable declaration by directly inserting it in _G.

In the code of Figure 2, the *declare* method receives the following parameters: the name and initial value of a variable. Then, it invokes the Lua *rawset* method. This method inserts in the _G table, a *name* field with value equal to *initval*. It also allows the use of a *metatable* to control reading and writing in global variables.

```
1 function declare (name, initval)
2   rawset(_G, name, initval or false)
3 end
```

Figure 2: Declare method

Despite these reflective facilities, Lua does not provide a MOP that unifies and organizes the introspection and reflection mechanisms required to make the introduction of AOP easier. Therefore, in the following sections, we will describe a MOP to the Lua language and its support for AOP.

3. Aspect-Oriented Infrastructure

Lua support for AOP is provided by an *aspect class* used to define aspects that are dynamically weaved by a meta-object protocol named LuaMOP.

Figure 3 illustrates the blocks that compose the AOP architecture that we call *AspectLua architecture*. The first layer is composed of the Lua language with its reflective facilities. The second layer is composed of the LuaMOP facilities that take advantage of the Lua reflective mechanisms. LuaMOP provides a set of meta-classes that support the dynamic introduction of aspects defined at the third layer. AspectLua provides the *aspect class* to the definition of AOP elements. A

programmer can take advantage of AspectLua without knowing either LuaMOP or the Lua reflective features. Moreover, AspectLua does not violate the internal mechanism of the Lua language, as it is built upon the Lua reflective features.

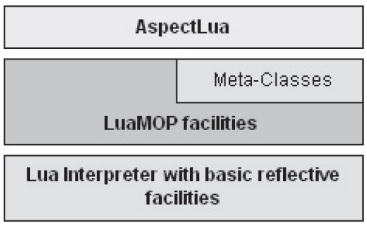

Figure 3: AspectLua architecture

3.1. LuaMOP

LuaMOP is a meta-object protocol that supports the creation of a meta-representation to each element that composes the Lua runtime environment: variables, functions, tables, userdata and so on. Each element is represented by a meta-class that provides a set of methods to query and to modify the behavior of each element of the base class. They are organized in a hierarchical way where *MetaObject* is the base meta-class (Figure 4). Derived from this meta-class are *MetaVariables, MetaFunctions, MetaCoroutine, MetaTable,* and *MetaUserData* meta-class. Furthermore, LuaMOP also provides a *Monitor* class to monitor the occurrence of events in the Lua runtime environment.

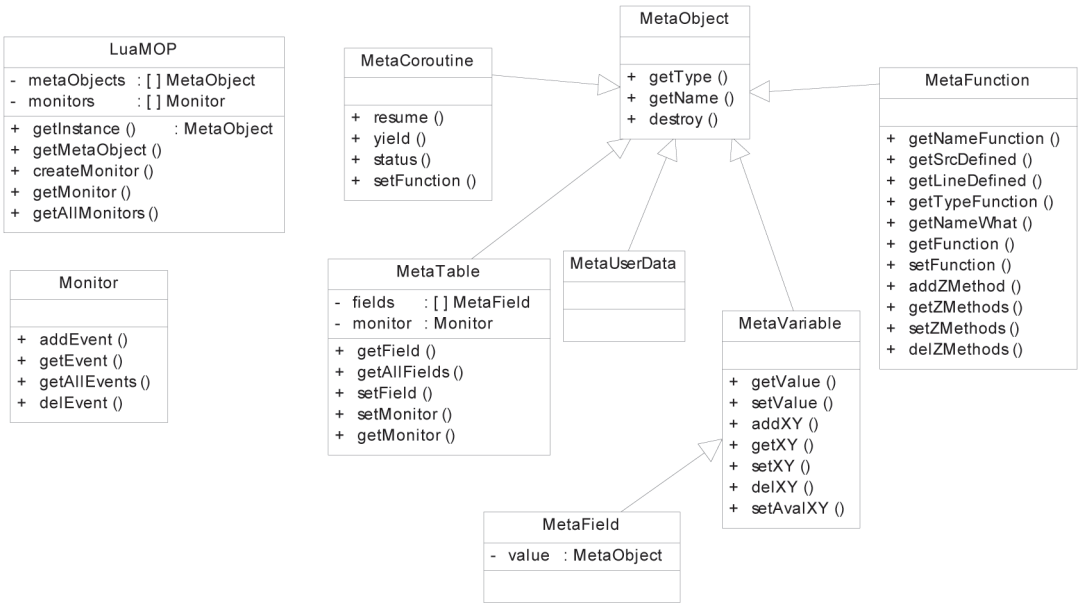

Figure 4: LuaMOP class diagram. X should be replaced by Pre and Pos, Y should be replaced by Get and Set and Z should be replaced by Pre, Pos or Wrap

The meta-representation provided by LuaMOP is created via the invocation of the *getInstance(instance)* method. This method returns the meta-object corresponding to the object with name or reference described by the instance parameter. This meta-object is an instance of a meta-class described above. For each meta-class there are methods that describe it and that supports changing the behavior of a meta-object. Thus, *getType()* and *getName()* methods can be invoked by all meta-classes, since these methods are part of the *MetaObject* meta-class. These methods return, respectively, the meta-object type and name. The *destroy()* method is used to disconnect the meta-object from the base object and to destroy the meta-object. The *getInstance* method can also be invoked, using as an input parameter a non-determined name. For instance: *getInstance("string.*")* returns a list (table) with meta-objects that represent the functions of the *string* package.

The *MetaVariable* meta-class provides the following methods: *getValue* and *setValue*. These two methods are used to get and to modify the value of a variable. The *get* and *set* events are two other functions that can be intercepted by LuaMOP. The *get* event occurs when a variable, table or function is referenced, indexed or invoked. The *set* event occurs when values are associated with variables, table elements and functions.

The *addPreGet*, *addPosGet*, *addPreSet*, and *addPosSet* methods insert a function to be executed before (or after) variable reading or writing. Figure 5 shows an example of the use of these functions. On the first line, the *balance* variable is set to 10. On the next line, a meta-object is created to represent the *balance* variable. The four following lines declare the *checkread* function and associate this function with the *metavar* meta-object, via the *addPreGet* method. The main goal of these functions is to control the access to the *balance* variable. Thus, if the function inserted by the *addPreGet* method returns a value different from *nil*, the reading process is interrupted. The existence of other functions demands that all functions return *nil* to allow reading the variable. This LuaMOP standard behavior can be modified by the *setAvalPreGet(funcaval)* function. The *funcaval* function receives as a parameter a table with all outcomes provided by the functions inserted using the *addPreGet* method. Based on this list, the *funcaval* function should return a non-*nil* value to interrupt the reading.

Line 10 shows the use of *addPosGet* function to associate the *convert_to_dollar* function with the *metavar* meta-object. The *convert_to_dollar* function is invoked after reading the variable and it receives the reading value. It can return a new value. On line 8, the *balance* variable value is divided by 2.65 and the outcome is returned to the application. The *addPreSet* method is used to modify the variable value. On line 15, this function is invoked to associate the *convert_to_real* function with the *metavar* meta-object.

The *convert_to_real* function is executed before writing the new value provided by the application. The *convert_to_real* function can return *nil* or a table. If it returns *nil*, the writing process is canceled and the original value of the writing process is maintained. The change of the original value is only performed via the return of a table with size greater than one (the case of line 12). The remainder of Figure 5 shows the use of *addPosSet* method that is invoked to associate the *writelog* function with the *metavar* meta-object. The *writelog* function is invoked after the *balance* variable is given a new value. This new value is represented by the *value* parameter. Similarly to the *setAvalPreGet* function, *setAvalPosGet*, *setAvalPreSet*, and *setAvalPosSet* functions can also be invoked to modify the behavior of each function. The *getXY*, *setXY*, and *delXY* functions are used to, respectively, get all functions associated with Pre/Pos and Get/Set, to determine a new function set, and to remove an element (function) of the functions set.

```
1 balance = 10
2 metavar = LuaMOP:getInstance("balance")
3 function checkread()
4    if (user ~= "admin")then return 1 end
5 end
6 metavar:addPreGet(checkread)
7 function convert_to_dolar(value)
8     return value / 2.65
9 end
10 metavar:addPosGet(convert_to_dollar)
11 function convert_to_real(value)
12    if (user == "admin") then return
                                {value *  2.65}
13    else return nil end
14 end
15 metavar:addPreSet(convert_to_real)
16 function writelog(value)
17   print("It was write the value:" .. value)
18 end
19 metavar:addPosSet(writelog)
```

Figure 5: LuaMOP example with add methods

A *MetaFunction* class represents all functions of a Lua application. This meta-class provides the following methods: *getNameFunction*, *getFunction*, and *setFunction*. *getNameFunction()* method gets the function name referenced by a meta-object. *getFunction()* method gets the function referenced by a meta-object, and *setFunction(newfunction)* allows modification, at runtime, of the function behavior. Some other functions that give details about a meta-object are provided: *getSrcDefined()* returns the file that contains the function definition; *getLineDefined()* returns the line that contains the function declaration; *getTypeFunction()* identifies if a function is written in Lua or in C; *getNameWhat()* identifies if a function is global or local.

```
1 function sum(a,b) return a + b end
2 function newsum(a,b) return a + b * 2 end
2 metafunction = LuaMOP:getInstance(sum)
3 print(metafunction:getNameFunction())
4 metafunction:setFunction(newsum)
5 print(metafunction:getNameFunction())
```

Figure 6: LuaMOP example with setFunction

The *MetaFunction* meta-class also offers the *addPreMethod*, *addPosMethod*, and *addWrapMethod* methods. These methods define the place where the behavior is added: Pre(before), Pos(after), and wrap the execution of a function. An example of the use of these functions is illustrated in Figure 7.

```
1 function reglog(self,value)
2  print("Deposited Value:",value)
3 end
4 metafun = LuaMOP:
            getInstance("Account.deposit")
5 metafun:addPosMethod(reglog)
6 Account:deposit(10)
```

Figure 7: LuaMOP example with addPosMethod

The meta-object is obtained on line 4. On line 5, the *addPostMethod* method is invoked to add the *reglog* function defined from lines 1 to 3. When the *deposit* method is executed (line 6), the LuaMOP mechanisms automatically invoke the *reglog* method.

To control the functions associated with a given behavior, *MetaFunction* provides the following methods: *getZMethods*, *setZMethods*, and *delZMethods*. The *getPreMethods* method, for instance, returns a list of all methods added to the *Pre* behavior. The list provided by the *getPreMethods* is ordered and sent as a parameter to the *setPreMethods* method. This latter method modifies the execution order of the methods defined to the *Pre* behavior. The removal of a method can be done using the *delPreMethods* method.

The *MetaTable* class represents the application tables and provides the following functions: *getField*, *getAllFields*, and *setField*. The *getField(name)* method receives the field name parameter and returns a *MetaField* that represents it. The *MetaField* class inherits from the *MetaVariable* class and, as a consequence, provides the same functionalities of a *MetaVariable*. For example: to add a function to be invoked before reading the variable value. To get all *MetaFields* of a *MetaTable*, the *getAllFields()* function can be used.

The *setField(namefield, value)* method is used to modify a table field and to insert a table field if it does not exist. In Lua, classes are represented by *Table* elements. Thus, the *setField* method can be used to add both new attributes and new methods.

Despite the fact that the meta-object provides the *setField* method, it does not exclude the use of another mechanism to insert new fields. The role of the meta-object is to maintain the causal connection and to update its properties according to the changes in the base objects. Figure 8 shows an example of this behavior.

```
1 Account = {}
2 Account.balance = 0
3 metaAccount = LuaMOP:getInstance("Account")
4
5 Account.NameAccount = "Mary"
6 function logchangename(value)
7  print("The new name is", value)
8 end
9 metavar = metaAccount:
                      getField("NameAccount")
10 metavar:addPosSet(logchangename)
```

Figure 8: LuaMOP and causal connection

On lines 1 and 2 the *Account* object is defined with the *balance* attribute. On the next line, a meta-object is created to represent the *Account* object. At this moment the *metaAccount* meta-object has only *balance* as a *MetaField*. On line 5, a new field, *NameAccount*, is defined to the *Account* object. This action changes the base object and triggers an automatic modification of the *metaAccount* meta-object that provides the *getField* method to recover the *MetaField* with name *NameAccount*. Such *MetaField* provides the same functionality of a *MetaVariable*. This allows the invocation of the *addPosSet* method to handle the modifications of the *NameAccount* field.

LuaMOP functionality goes beyond the provision of a meta-representation. It is also possible, via *Monitors*, to capture events from the runtime execution environment. A Monitor provides the same functionalities of a *Metatable*. The difference between them is the possibility of defining a *Monitor* to handle events related to elements that have not yet been declared in the application. The *Monitor* accepts the same events handled by a *Metatable* (add, sub, index, newindex, etc) and also a new one: *noindex*. This event is different from the *index* event because it is invoked only when the correspondent index is not found.

Figure 9 shows how a monitor can be used to load a library only when it is really used. This facility avoids unnecessary resource allocation. This example defines the dynamic loading of LuaSocket library. Thus, methods of the *socket* object, such as *bind* and *connect*, that are not yet available in the execution environment, will be loaded. The first line creates a *monitor* to observe the events sent to the *socket* object. Line 9 adds the *loadmethod* function to handle all events to *socket* object that do not have an index. In the case of a *noindex* event, the function receives as a parameter the name of the invoked function and the original parameters. Thus, the *socket.bind("*", 0)* method, that does not exist yet, is handled by the monitor. The monitor invokes *loadmethod* to load the LuaSocket library. Then, it gets and invokes the *socket.bind* function through the *metasocket* meta-object.

```
1 monitor = LuaMOP:createMonitor("socket.*")
2 function loadmethod(self, namefunc, arg )
3   dofile('luasocket/lua.lua')
4   socket = require("socket")
5   metasocket =
    LuaMOP:getMetaObject(namefunc)
6   local func = metasocket:getFunction()
7                     return func(unpack(arg))
8 end
9 monitor:addEvent("noindex", loadmethod)
```

Figure 9: Using LuaMOP's Monitor

The declaration of the socket object, on lines 3 and 4, is followed by the automatic creation of a meta-object to represent the *socket* object. This meta-object is also monitored by the monitor defined on line 1. The monitor does not interfere in a second invocation, such as a invocation of the *socket.connect()* method, since this method has already been declared. The monitor interferes in the first execution of the *socket* object. For instance, when the *bind* method, that has not been previously declared, is invoked. In this case, *loadmethod* function is invoked to load the LuaSocket library and to execute *socket.bind*. Lines 5 to 7 show how a field (socket.bind) of the automatically created meta-object is obtained and the *socket.bind* function is invoked.

3.2. AspectLua

AspectLua defines an *Aspect* class that handles all the aspect issues. Thus, AspectLua offers an abstraction layer that hides the complexities of meta-objects. Through AspectLua the user can define the AOP elements without knowing either LuaMOP or the Lua reflective facilities.

To use AspectLua, it is necessary to create an instance of the *AspectLua* class by invoking the *new* function. After creating a new instance, it is necessary to define a Lua table containing the aspect elements (name, pointcuts, and advice). Figure 10 illustrates an aspect definition:

- The first parameter of the *aspect* method is the aspect name;
- The second parameter is a Lua table that defines the pointcut elements: its name, its designator and the functions or variables that must be intercepted. The designator defines the pointcut type. AspectLua supports the following types: *call* for function calls; *callone* for those aspects that must be executed only once; *introduction* for introducing functions in tables (objects in Lua); and *get* and *set* applied upon variables. The *list* field defines functions or variables that will be intercepted. It is not necessary that the elements to be intercepted have been already declared. This list can use wildcards. For instance, *Bank.** means that the aspect should be applied for all methods of the *Bank* class;
- Finally, the third parameter is a Lua table that defines the advice elements: the type (after, before, around, and so on) and the action to be taken when reaching the pointcut. In Figure 10, *logfunction* acts as an aspect to the *deposit* function. For each *deposit* function invocation, *logfunction* is invoked *before* it in order to print the *deposit* value.

```
asp = Aspect:new()
id = asp:aspect( {name = 'logaspect'},
   {pointcutname = 'logdeposit',
   designator='call', list= {'Bank.deposit'}},
   {type = 'before', action = logfunction} )

Bank = {balance = 0}
function Bank:deposit(amount)
 self.balance = self.balance + amount
end
function logfunction(a)
 print('It was deposited: ' .. a)
end

oldasp = asp:getAspect(id)
oldasp.advice.type = 'after'
asp:updateAspect(id, oldasp)
```

Figure 10: Example of aspect definition

The invocation of the aspect method defines a pointcut for each aspect and returns an aspect identification (id). Thus, to associate various pointcuts with a same aspect it is necessary to invoke the *aspect* for each pointcut. To manage the defined aspects, AspectLua provides the following functions: *getAspect(id)*, *getAll()*, *removeAspect(id)*, and *updateAspect(id, newasp)*. The *getAspect* and *getAll* methods can be used to get one or all aspects already defined. After getting an aspect, it is possible to modify or to update its elements by using the *updateAspect* method. This method can modify *pointcuts* and *advice* of an aspect already defined. Aspect removal can be done by using the *removeAspect* method.

In Figure 10, the *Bank* object with the *deposit* method is declared after the join point definition. In a previous version of AspectLua, this definition was not possible because each target method of a join point needed to be previously declared. To address this limitation, the current version of AspectLua uses *monitors* to implement *anticipated join point* – a join point that does not have a meta-object but that has a monitor associated with it. *Anticipated join point* allows the programmer to define a join point for elements that have not yet been declared in the application program. In addition, it is possible to intercept this join point even if it does not exist in the application. This is the main difference between *anticipated join point* and some solutions implemented by other works [1,3,6,10]. These solutions support the definition of join point for elements that have not yet been declared but the interception of each join point happens just when the element is already loaded. Therefore, if the element is not loaded, the join point is not intercepted. In AspectLua it is possible to intercept join point for elements that is neither defined nor loaded.

An *anticipated join point* can be useful in many situations. For instance, for a dynamic resource allocation in embedded systems, anticipated join points can be used to support a lazy loading approach [12]. A simple lazy loading scenario is illustrated in Figure 9 by using a *Monitor*. AspectLua can be used to abstract away the use of Monitors. For instance, the following code implements the same functionality of Figure 9: *aspect({name = 'lazyload'}, {pointcutname = 'loadmethod', designator = 'call', list = {'socket.*'}}, {type = 'before', action = loadmethod})*. In this case, AspectLua automatically defines the monitor to handle the *anticipated join point* (*socket.**).

To control the execution order of aspects in a given pointcut, AspectLua offers *getOrder* and *setOrder* functions. *getOrder* is used to get the list of aspects associated with a variable or function. It receives as a parameter the name of the variable or the function. It returns a list with the current aspect invocation order. *setOrder* is

used to modify this order. This function receives the following parameters: variable or function name and the new execution order. In Figure 11 the *deposit* method has two aspects that will be executed before it. By default, the execution order is the order of aspect definition. Therefore, *logfunction* will be executed before *checkRights*. To modify this order, *setOrder* can be used with the following parameters: *deposit* and a table defining a different order. In order to get information about a variable or function, *getOrder* is invoked receiving its name as a parameter.

```
function checkRights() ... end
a:aspect( {name = 'secaspect'},
 {pointcutname = 'verifyRights',
 designator='call',list={'Bank.deposit'}},
 {type ='before',action=checkRights } )
local order= Aspect:getOrder('Bank.deposit')
Aspect:setOrder('Bank.deposit',{order[2],
order[1]})
```

Figure 11: Defining order to aspects invocations

AspectLua also allows the introduction of new methods in a class via the *introduction* designator. Figure 12 shows the introduction of the withdraw method in the *Bank* class. This designator does not demand the advice type. The last line of Figure 12 shows that after introducing a new method, it can be used in the same way as previously existing methods.

3.3. Integration between AspectLua and LuaMOP

AspectLua exploits the power of LuaMOP and uses it to the definition of aspects, pointcuts and advice. In LuaMOP, aspects are defined at the meta-level via meta-objects and application components are defined at the base-level. The weaving process that combines the two levels is achieved by LuaMOP. Due to the integration between AspectLua and LuaMOP, it is unnecessary to modify the Lua syntax to handle an aspect definition.

```
function withdraw() ... end
a:aspect({name = 'insertMethod'},
{name = 'newMethod',designator='introduction',
list = {'Bank.withdraw'}},{action = withdraw})
Bank:withdraw(5)
```

Figure 12: Use of introduction designator

Figure 13 illustrates the integration between AspectLua and LuaMOP. In this example, AspectLua is used in the definition of the *LogFunction* advice that should be executed at the join point *Bank.deposit*. In order to handle this issue, AspectLua asks LuaMOP to create the *MetaBank* meta-object as a meta representation of the *Bank* object and to insert the behavior (*LogFunction*) at the *MetaField deposit*. *LogFunction* should be executed before the *deposit* method. The existence of a meta-object means that all messages to the *Bank* object will be intercepted by LuaMOP and forwarded to the *MetaBank* meta-object. When the meta-object receives a message, it inspects its *MetaFields* to verify the need of executing an extra behavior. If not, the message is forwarded to the base-object. In Figure 13, *MetaBank* executes *LogFunction* and after that, the *deposit* function executes.

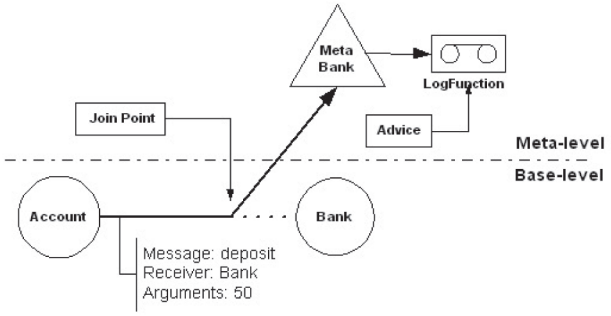

Figure 13: Integration between AspectLua and LuaMOP

AspectLua also uses *monitors* to support the definition of aspects associated with undeclared elements. Monitors act on the interception of *anticipated join points* and on the execution of its associated *advice*. Figure 14 shows the steps involved in the definition and execution of an *anticipated join point* that implements the lazy loading of LuaSocket. There are two grey boxes representing the Aspect configuration file and the application code. Both are defined by a programmer. The definition of the *anticipated join point* is done in the *Aspect Configuration script* by using the *aspect* method. In this case, an aspect to the *socket.** pointcut and an advice named *loadmethod*. Using the aspect definition, AspectLua verifies the inexistence of the socket object.

After that, it creates a monitor to receive the events to the socket object. The *createMonitor* method (provided by LuaMOP) creates a table whose name is the same of the target object, and inserts in this table a *metatable* to control the access. When the application invokes *socket.bind*, as the socket object does not yet exist, the monitor receives the invocations and forwards it to *loadmethod*. This method loads the LuaSocket library. Then, the *socket* object is created. Meanwhile, LuaMOP creates a meta-object to the *socket* object and associates it with the monitor. After that, the monitor is only invoked to handle invocations regarding to methods not implemented by the socket object.

3.4. Performance Evaluation

This section discusses performance issues regarding the use of a meta-representation in the Lua execution environment. The tests compare the execution time with and without the use of a meta-object. This way, it is possible to verify the impact of the meta-object presence at the invocation process by comparing the

execution time of functions X and Y. They were executed, respectively, in 59.72µs and 3.91 µs with no meta-object associated with them. The evaluation was done in a PC Duron 1.6MHz with 256MB of RAM, using Linux-Mandrake 9.2.

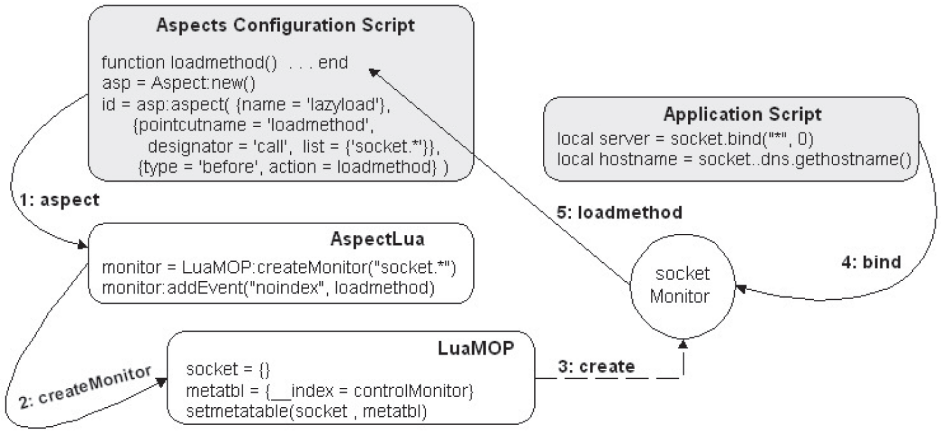

Figure 14: Anticipated Join Point implementation

Table 1 shows the results of the performance tests. The first line shows a comparison between the execution time of X and Y functions and the execution time of X function associated with a meta-object that contains the Y function as a *PreMethod*. The difference is low considering the time that the meta-object takes to manage the messages. The second line compares the access and execution time of a method that belongs to an object (table), for example *Bank.X()*, to the same object associated with a meta-object. The difference between these two invocations will be greater only when, in the following line, a function (Y) is associated with the *Bank* meta-object to be executed after the X function. In this case the difference increases from 2.15µs (in the previous line) to 6.7µs. This difference is related to the amount of time involved in loading the functions associated with the *Metafield*.

Table 1: Performance Evaluation. Time in µs.

Test	Without Meta-Object	With Meta-Object
Execution of X and Y Functions	63.75	66.95
Execution of X function, via an object (table)	61.03	63.18
Execution of X and Y functions, via an object (table)	64.34	71.04
Reading a variable	0.94	2.86
Writing in a variable	1.19	3.09

The two last lines of Table 1 compare the times to read and to write in a global variable. The difference (almost three times) between the execution times is more related to the execution time of reading and writing a variable, which is lower than any other inconsistency in the algorithms used by the meta-object. This means that reading and writing variables is a very quick task (0.94µs and 1.19µs) and as a consequence, the introduction of a new processing, such as a procedure that deals with messages to a variable, can increase the execution time.

4. Case Study

In order to illustrate the use of the AOP infrastructure in a large system we have implemented a case study that uses AOP to customize a middleware platform. The use of AOP to customize a middleware platform has been the subject of other research [28, 29] that recognize the benefits of AOP in the context of middlewawre architecture to increase configurability and adaptability.

The middleware system we used is LOpenOrb[15], a Lua implementation of the Open-ORB component model [17]. Figure 15 illustrates the elements that compose LOpenOrb architecture. They are organized in a layered style where the upper layers depend on the lower layers and each element provides an API.

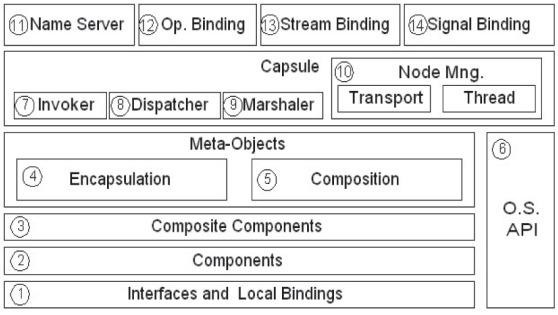

Figure 15: LOpenORB Architecture

The functionality of some methods provided by the LOpenOrb API is illustrated in Figure 16. This figure illustrates the steps to invoke the *deposit* method implemented in a remote server. The two first lines load and start LOpenOrb. On the next two lines *local_bank* and *remote_bank* interfaces are defined. Interfaces are access points of components. Each interface can export and/or import methods. Exported methods correspond to the provided services. Imported methods are required services. In Figure 16 the *local_bank* interface imports the *deposit* method while the *remote_bank* interface uses the local container (local *Capsule* in Open-ORB terminology) to get the remote reference of the *bank* interface. This reference is stored in the *bank_int.ref*. file. After obtaining the reference, a local *binding* is defined between the two interfaces. The role of the local *binding* is to associate compatible interfaces. Finally, the example illustrates the remote invocation of the *deposit* method. To support this invocation it is necessary to use the following layers of the LOpenOrb middleware implementation: 1,2,3,4,6,7,9 and 10. The layers required depends on the functionality used by the application. In the case of a method invocation, which is in the a same container (*Capsule*) of the *local_bank* interface, it is only necessary to use layer 1. As LOpenOrb is a monolithic middleware, it loads all layers at starting time (*init* method invocation). This simple example, including local and remote invocation, illustrates a problem of a generic middleware that includes a variety of services (object repositories, security services, QoS, and so on). The middleware is usually implemented as a black box with no support for adapting its infrastructure[1] to offer only the services

```
1 require "LOpenOrb"
2 LOpenOrb.init(arg)
3 local_bank = LOpenOrb.IRef({},
              {},{"deposit"})
4 remote_bank = LOpenOrb.localcapsule:
  getRemoteInterfaceByFile("./bank_int.ref")

5 LOpenOrb.localBind({local_bank},
          {remote_bank})
6 local_bank:deposit(50)
```

Figure 16: Using LOpenOrb

In order to address this lack of customizability support we use AOP to makes it possible to customize LOpenOrb. The goal is to avoid resource wasting and to improve dynamic adaptation. This approach targets two common problems of middleware platforms. The first one is related to the complexity of providing customized middleware implementations by separating basic code and croscutting concerns. The second one is related to the dynamic evolution of middleware platforms. The insertion of new functionalities must be controlled in order to avoid tangled code.

[1] The concept of Interceptors present in some middleware platforms are a simplified form of join points that are tightly coupled with the middleware internal structure. So, interceptors do not address separation of concerns. Furthermore, in this mechanism, advice are inserted by registering callback functions and follow a lot of constraints to avoid infinite recursions.

Table 2: Dependence relationships between invocations types and LOpenORB layers

Designator	Dependence layer
LOpenORB.*IRef	1
LOpenORB.localBind*	1
LOpenORB.localcapsule. getRemoteInterfaceByFile	1,2,3,4,6,7,9, 10
LOpenORB.localcapsule. serve	1,2,3,4,5,6,8,9, 10
LOpenORB.encapsulation.*	1,2,3,4

The aspect-oriented middleware we propose, named *Aspect Open-ORB*, allows the customization of the middleware according to the application requirements. This infra-structure is based on the idea that the middleware functionalities are defined by the application code. For instance, if the application code contains invocation to the *getRemoteInterfaceByFile* function, which means remote access to a server, the middleware implementation must load the layers responsible for remote invocations. To handle this issue, AspectLua monitors the invocation of each method, verifies its dependencies and loads the middleware layers needed to support the invocation. Table 2 shows the relationship between some methods provided by the LOpenOrb API and their corresponding layer dependencies. According to this table, an aspect is defined to each line. All join points are associated to a same advice that loads the layers needed to support the invocation of the join point.

Figure 17 illustrates the architecture of the Aspect Open-ORB infrastructure. *LOpenOrb Aspect Dependencies* define the dependencies between layers and types of method invocation. Each application is composed by its base code (core) and its aspects (Security, Fault Tolerance, etc). At runtime, AspectLua loads LOpenOrb elements according to the application needs. These dynamically loaded elements compose Aspect Open-ORB. Thus, the Aspect Open-ORB internal architecture is dynamically composed according to the application needs.

The goal of Aspect Open-ORB is to support middleware customization and the ability of dynamic insertion of new requirements beyond those provided by LOpenOrb. The middleware customization process requires the use of two sets of aspects. The first set is responsible for intercepting the invocation of the *LOpenORB.init* method. The second one supports the elements defined in Table 2.

To illustrate this issue, Figure 18 shows the code that implements the middleware customization. Lines 3 to 6 contain the definition of the *myInit* function. This function, via the aspect

defined on line 7, acts on *LOpenORB.init*. All invocations to the *init* function are redirected to the *myInit* function. This avoids the loading of all layers, which is done by the *init* method. As the layers are not loaded, the API of the elements is not available.

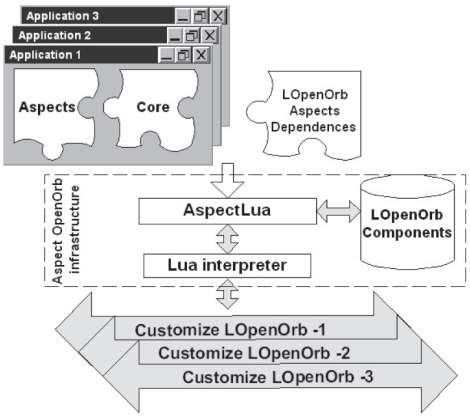

Figure 17: Aspect Open-ORB infrastructure

To maintain the availability of the LOpenOrb API it is necessary to define a second set of aspects that acts as anticipated join points to the invocations to the LOpenOrb API. Table 2 shows the elements that are defined as anticipated join points. This definition is illustrated from lines 9 to 12, which contain the code that creates the list of layers - *tblfiles* – indexed by the numbers presented in Figure 18. The next lines insert in *tblconf* the elements defined in Table 2. This insertion includes two fields: the Key field, which contains the aspect name, and the *dep* field, which stores the layers needed to invoke the method. Lines 43 to 45 show the definition of each element in the *tblconf* list as an aspect including an anticipated join point. Each aspect invokes only once (callone) the *loadfiles* method when the pointcut defined by *tblconf*.Key is reached. The goal of the *loadfiles* function is to load, according to the function name obtained in Line 25, a loop that returns in the *idxfile* variable each layer needed to support the invocation of the method. The *dofile* method loads the code of each layer. To maintain the loaded methods under control, the *deletefile* function is invoked after loading a method. This function removes the layer name from the *tblfiles* list. The *contload* variable is a counter that indicates the number of layers already loaded. This is important to avoid a double invocation of the desired method in the case of the following pointcuts: LOpenOrb.localcapsule.getRemoteInterface and LOpenOrb.localcapsule.*. LuaMOP is used to obtain the desired method that, at this moment, is already defined.

```
1 asp = Aspect:new()
2
3 function myInit(arg)
4  LOpenOrb.hostname = arg[2]
5  LOpenOrb.port = arg[3]
6 end
7 asp:aspect({name = 'skipInit'} , {name = 'skip', designator = 'call', list = {'LOpenOrb.init'}},
       {type ='around', action = myInit})
8
9 tblfiles = {}
10 tblfiles[1] = "lopenorb_interface_and_bidinglocal.lua"
11 tblfiles[2] = "lopenorb_component.lua"
12 tblfiles[3] = "lopenorb_composiste.lua"
13  . . .
14
15 tblconf = {}
16 table.insert(tblconf,{key = "LOpenOrb.*IRef", dep = {1} }
17 table.insert(tblconf,{key = "LOpenOrb.localBind*", dep = {1} }
18  . . .
19 table.insert(tblconf,{key = "LOpenOrb.localcapsule.serve", dep = {1,2,3,4,5,6,8,9,10} }
20 table.insert(tblconf,{key = "LOpenOrb.localcapsule.getRemoteInterfaceByFile", dep=
                                    {1,2,3,4,6,7,9,10} }
21 table.insert(tblconf,{key = "LOpenOrb.encapsulation.*", dep = {1,2,3,4} }
22 . . .
23
24 function loadfiles( ... )
25   local funcname = table.remove(arg,table.getn(arg))
26   loca contload = 0
27   for k,idxfile in ipairs(searchKey(funcname)) do
28                       local namefile = rawget(tblfiles,idxfile)
29                       if (namefile ~= nil)then
30                         dofile(namefile)
31                         deletefile(idxfile)
32                         contload = contload + 1
33                       end
34   end
35
36   if contload > 0 then
37                     metafunc = LuaMOP:getInstance(funcname)
38                     func = metafunc:getFunction()
39                     return func(unpack(arg))
40   end
41 end
42
43 for k,conf in ipairs(tblconf) do
44  asp:aspect({name = 'ConfORB'} , {name = 'loadfiles', designator = 'callone', list =
       {conf.key}}, {type ='around', action = loadfiles})
45 end
```

Figure 18: Middleware aspects configuration

Finally, the desired method is invoked on Line 39 and receives as parameter the arguments of the original invocation (arg).

In order to use Aspect Open-ORB it is necessary to load the file described in Figure 18 and after that to load the application code. Figure 16 shows a simple example of an application code. At runtime the invocation of the *LOpenOrb.init* method is replaced by the invocation of the *MyInit* method. This method defines the initialization parameters but does not load the LOpenORB implementation files. According to this situation, the next invocations in Figure 16 would return an error of missing method. However, as we are using anticipated join points to handle invocations to methods provided by the API, invocations to *LOpenOrb.IRef* or to any other method are forwarded to the proper *advice* that loads the implementation and invokes the target method.

5. Related Work

Related work include some AOP languages built on top of scripting languages. The most important are three AOP extensions based on well-known scripting languages: Python [13], Ruby [14] and Smalltalk [4]. AOPy [2] is built on top of Python. AOPy implements method-interception by wrapping methods inside the advice. Aspects definition uses the designator call and just one join point can be defined in a pointcut. In contrast, AspectLua supports the definition of several join points. AspectR [1] is built on top of Ruby. It implements AOP by wrapping code around existing methods in classes and supports wildcards. AspectS [6] is a Squeak/Smalltalk extension to support AOP. It uses modules and meta-level programming to handle AOP. It also supports wildcards.

The fact of being scripting languages brings some similarities among these AOP languages and AspectLua: they are built on top of a scripting language, no new language constructs are needed and aspect weaving occurs at runtime. The main difference between AspectLua and the other extensions is that none of them include all features supported by AspectLua. AOPy is very simple and supports only basic concepts. It does not support wildcards. Neither AOPy nor AspectR use a MOP to support AOP. AspectS has more similarities with AspectLua: both use a MOP, allow the definition of aspect precedence order, support the use of wildcards. However, none of these extensions allows the association of aspects with undeclared elements (*anticipated join points*). [12] provides a similar mechanism that does not use the actual idea of *anticipated join points* because this approach uses a *proxy* to represent the join points and needs an initialization method to identify the moment of loading an element. This approach is different from the AspectLua approach where it is not necessary any method to identify when an element must be loaded.

LAC – Lua Aspectual Component [5] – is a Lua extension whose main goal is to support the idea of Aspectual Components (AC) [11]. LAC is quite different from AspectLua because its elements are defined in order to support the idea of AC while AspectLua elements are defined following the traditional AOP concepts. LAC imposes a template where components and aspects are defined by different styles of classes. In contrast, AspectLua uses tables to represent aspects. The focus of LAC is in a model to implement AC. After defining this model, Lua was chosen to implement it. In contrast, the focus of AspectLua is in using Lua as an AOP language without introducing new commands or structure.

PROSE [19] is a dynamic AOP extension to the Java language. As AspectLua, PROSE does not introduce a new syntax for defining aspects. It uses the Java language itself. In the same way, there is no need of a special compiler. It uses the Java Virtual Machine Debug Interface (JVMDI) and just-in-time (JIT) features to make it possible the interception and execution of aspects. However, the main difference from PROSE to AspectLua is that AspectLua uses a pure interpreted approach.

JAC [27] and AspectWerkz [26] are AOP approaches that implement load time weaving strategies. They act at the class loader level. This means that the code is modified when it is loaded into the Virtual Machine. The main problem of these approaches is to violate the Java security mechanism. Furthermore, in AspectWerkz 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 original Java class loading in order to handle the aspect weaving. Second, it can only be used in Java or J2EE applications. AspectLua is integrated with a CORBA environment [25] 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 that of AspectWerkz.

The advantage of meta-object protocols is recognized by a number of works that propose a MOP for some traditional programming languages. For instance, OpenC++ [20] and OpenJava [21] are MOPs for C++ and Java, respectively. In this work we presented a MOP for Lua and also other tools combined with LuaMOP that compose a Lua-based AOP infrastructure.

As far as we are aware, the concept of anticipated join point, that is a central concept in our infrastructure, is not provided by other AOP approaches

6. Final Remarks

In this paper we have presented an AOP infrastructure based on Lua. The infrastructure is composed of LuaMOP and AspectLua. Aspects are defined using AspectLua. LuaMOP supports dynamic weaving by exploring the reflective features of Lua. We have described in detail how the weaving process takes place. As aspects are defined using Lua tables, it is not necessary to use different languages for the functional code and for the aspect code. For both programs the Lua language is used. This is a way of keeping the Lua philosophy – simple and small - in our AOP approach.

The infrastructure provides a range of features that introduces a great deal of flexibility to AOP: it is possible to define aspects at runtime; it supports the definition of aspect precedence order, wildcards, and the association of aspects with undeclared elements. It is worth pointing out that the concept of *anticipated join points* is very useful for dynamicity because it allows the dynamic insertion of aspects according to a new functionality of the component program. It goes beyond current AOP approaches where join points are linked to statically defined elements or, at most, join points are associated with elements that are loaded dynamically but before the interception process.

The idea of a dynamic AOP language is not new. However, the dynamic AOP approach presented in this work combines a set of features that are not offered together by other AOP language. We have chosen Lua because it is small, easy to use and it provides reflective mechanisms that allow extension of the language.

As a case study we applied our AOP infrastructure to customize a middleware platform. Middleware functionalities are expressed as aspects because it can be seen as an add-on functionality that supports method invocations. Anticipated join points play an important role in this context by allowing the association of aspects with undeclared elements.

As a future work we intend to implement support for remote aspect definition. We also intend to exploit scripting facilities inside the aspect definition by allowing the use of conditional statements to handle various pointcuts inside a given aspect.

References

[1] A. Bryant, R. Feldt. AspectR - Simple aspect-oriented programming in Ruby. http://aspectr.sourceforge.net/ , 2002.

[2] D. Dechow. Advanced Separation of Concerns for Dynamic, Lightweight Languages. In *5th Generative Programming and Component Engineering*.

[3] A. Gal, W. Schröder-Preikschat, O. Spinczyk. AspectC++: Language Proposal and Prototype Implementation. *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa - Fl, October, 2001.

[4] A. Goldberg, D. Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, 1983.

[5] S. Herrmann, M. Mezini. Combining Composition Styles in the Evolvable Language LAC. In: *ASoC Workshop in ICSE— International Conference on Software Engineering*, 2001.

[6] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS 2591, pp. 216-232, Springer-Verlag, London, 2002.

[7] R. Ierusalimsky, L. H. Figueiredo, W. Celes. Lua – an extensible extension language. *Software: Practice and xperience*, 26(6):635-652. 1996.

[8] P. Maes. Concepts and Experiments in Computational Reflection. *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, pp. 147-155, 1987.

[9] G. Kiczales, J. Lamping, A. Mendhekar et al. Aspect-oriented programming. In: *ECOOP'97 — European Conference on Object-Oriented Programming*. Springer-Verlag, Finland. 1997.

[10] G. Kiczales, E. Hilsdale, J. Hugunin et al. An Overview of AspectJ. In *ECOOP'2001 — European Conference on Object-Oriented Programming*. Budapest, Hungary. 2001.

[11] K. Lierberherr, D. Lorenz, M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS99 –01, Notheastern University. 1999.

[12] R. Miles. Lazy Loading with Aspects. ONJa-va.com, http://www.onjava.com/pub/a/onjava/2004/03/1 7/ lazyAspects.html. 2004.

[13] G. Rossum. Python Reference Manual, http:// www.python.org/doc/current/ref/ref.html. 2003.

[14] D. Thomas, A. Hunt. Programming Ruby: A Pragmatic Programmer's Guide. http://www.rubycentral.com/ book/, 2004.

[15] N. Cacho, T. Batista. Adaptação Dinâmica no Open-Orb: detalhes de implementação In *23th Brazilian Symposium on Computer Networks (SBRC'2005)*, SBC, Fortaleza, CE, May 2005, pp. 495-508.

[16] A. Andersen, G. S. Blair, F. Eliassen. A reflective component-based middleware with quality of service management. In *PROMS 2000, Protocols for Multimedia Systems*. Cracow, Poland, 2000.

[17] G. S. Blair et al. The design and implementation of Open ORB v2. *IEEE Distributed Systems Online*, 2(6), 2001. http://www.cs.uit.no/aa/abstracts/blair2001a.htm l.

[18] J. Sztipanovits, G. Karsai. Generative Programming for Embedded Systems. *The ACM SIGPLAN/ SIGSOFT Conference on Generative Programming and Component Engineering*. Lecture Notes In Computer Science (LNCS), Vol. 2487, pp. 32-49, 2002.

[19] Nicoara, G. Alonso. Dynamic AOP with PROSE. Department of Computer Science. Swiss Federal Institute of Technology Zürich. ttp:// www.iks.inf.ethz.ch/publications/publications/files/ PROSE-ASMEA05.pdf

[20] Chiba, S. A Metaobject Protocol for C++. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Austin, Texas, October 1995, pp. 285-299.

[21] Tatsubori, M. et al. OpenJava: A Class-based Macro System for Java. In *Reflection and Software Engineering*, LNCS 1826, Springer Verlag, 200, pp. 117-133.

[22] D. H. H. Ingalls, T. Kaehler, J. Maloney et al. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of OOPSLA '97*, Atlanta, Georgia, October 5-9, 1997. SIGPLAN Notices 32(10).

[23] A. D. Almeida, N. Cacho, T. Batista. LuaSpace Plus: Um Ambiente Visual para Desenvolvimen-to de Aplicações CORBA. In *Proceeding of the 18th Brazilian Symposium on Software Engineering(SBES'2004)*, SBC, pp. 163-177, Brasília, DF, October 2004.

[24] N. Cacho, F. Fernandes, T. Batista. Handling Dynamic Aspects in Lua. *Journal of Universal Computer Science (J.UCS)*, 11(7):1177-1197, 2005.

[25] F. Fernandes, T. Batista, N. Cacho. Exploring reflection to dynamically aspectizing corba--based applications. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pp. 220–225, New York, USA. ACM Press. 2004.

[26] J. Bonér. AspectWerkz - Dynamic AOP for Java. http:// codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf. 2003.

[27] R. Pawlak, L. Duchien, G. Florin et al. JAC: An Aspect-Based Distributed Dynamic Framework http:// jac.objectweb.org/.2005.

[28] C. Zhang, D. Gao, H. Jacobsen. Towards Just- in-time Middleware Architectures. In *Fourth International Conference on Aspect-riented Software Development*, Chicago, USA, March 2005.

[29] F. Hunleth, R. Cytron, C. Gill. Building Customizable Middleware using Aspect Oriented Programming. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. 2001. Tampa, Florida.