Handling Dynamic Aspects in Lua

Nélio Cacho, Fabrício Fernandes, Thaís Batista

Departamento de Informática e Matemática Aplicada (DIMAp)
Universidade Federal do Rio Grandre do Norte (UFRN)
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 using 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 an Aspect class provided by AspectLua and then they are weaved through LuaMOP. The difference of AspectLua to other aspect-oriented languages is that it combines a range of powerful features.*

1. Introduction

Aspect-oriented programming (AOP) has been gaining attention due to its focus on separation of concerns and modularization of crosscutting concerns. In general, aspect-oriented approaches are static – aspect code and functional modules 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 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, Ruby, and Smalltalk. These dynamic weaving approaches allow that aspects can be woven at runtime. However, they present some limitations. None of them combines a set of features to make easier and powerful AOP: (1) insertion and removal of aspects at runtime; (2) the definition of aspects precedence order; (3) the possibility of using wildcards; (4) the possibility of associating aspects with undeclared elements (*virtual join points*); (5) a dynamic weaving process via a meta-object protocol.

Virtual join points are interception points for elements that have not yet been declared in the application program. Using virtual join points it is possible to intercept an invocation to an undeclared method and to apply a specific action, such as, lazy loading a code. Virtual join points are introduced in AspectLua to avoid the need of loading the application code that contains a given join point before loading the aspect code regarding to this join point. We consider that the lack of support for virtual join points is a limitation of the AOP approaches because 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.

In this work we present an aspect-oriented infrastructure that handles aspect-oriented programming and addresses the issues described above. The infrastructure is composed of: (1) *AspectLua* – an extension to the Lua language [Ierusalimsky et. al 1996] to allow aspects definition; (2) *LuaMOP* – 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 application original 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 a virtual join point without knowing that LuaMOP implements an underlying mechanism, named *Monitor*, to support virtual join points.

We choose 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.

This paper is organized as follows. Section 2 presents the underlying concepts of this work: aspect-oriented programming, reflection and the Lua language. Section 3 presents the aspect-oriented infrastructure proposed in this work to handle dynamic AOP: LuaMOP, AspectLua and the relationship between them. Section 4 discusses 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 computational 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 [Kiczales et al. 2001] because it is the most traditional aspect-oriented language. *Aspects* are the elements designed to encapsulate crosscutting concerns and take them out of the application basic code (components). *Join Points* are the elements of the component language semantics that aspect programs coordinate with [Kiczales et al. 1997]. Join points can represent data flows of the component program, runtime method invocations in the component program, etc. *Pointcuts* are the union of join points. Through pointcuts, it is possible to get methods arguments values, attributes, exceptions, etc. Pre-defined pointcuts designators 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.

The *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, etc.

The weaving process places together the code defined in the join points and the advices. Weaving can be done at compile time or at runtime. In AspectJ and AspectC++ [Gal et al. 2001] weaving is done at compile time. New language constructs to handle AOP were added to the language syntax. A special compiler represents the weaver that mixes 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 advices insertion. The recognition and introduction of new behaviors (advices) are directly related to the use of reflection.

Reflection [Kiczales et al. 1991] 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 for a causal connection to allow modifications at the meta-level can 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. This way, meta-objects are instances of 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 and the dynamic insertion of advices that will 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. This 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* allows 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 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. On line 2, the *metatb* metatable is defined. 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.

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

Figure 2: Declare method

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.

Despite of all these reflective facilities, Lua does not provide a MOP that unifies and organizes the introspection and reflection mechanisms required to make easier the introduction of AOP. 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.

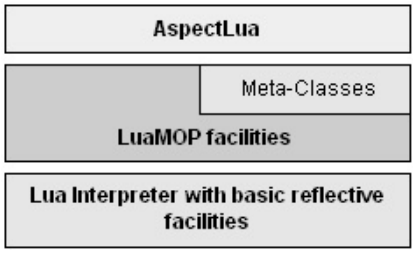

Figure 3: AspectLua architecture

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.

3.1. LuaMOP

LuaMOP is a meta-object protocol that allows 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.

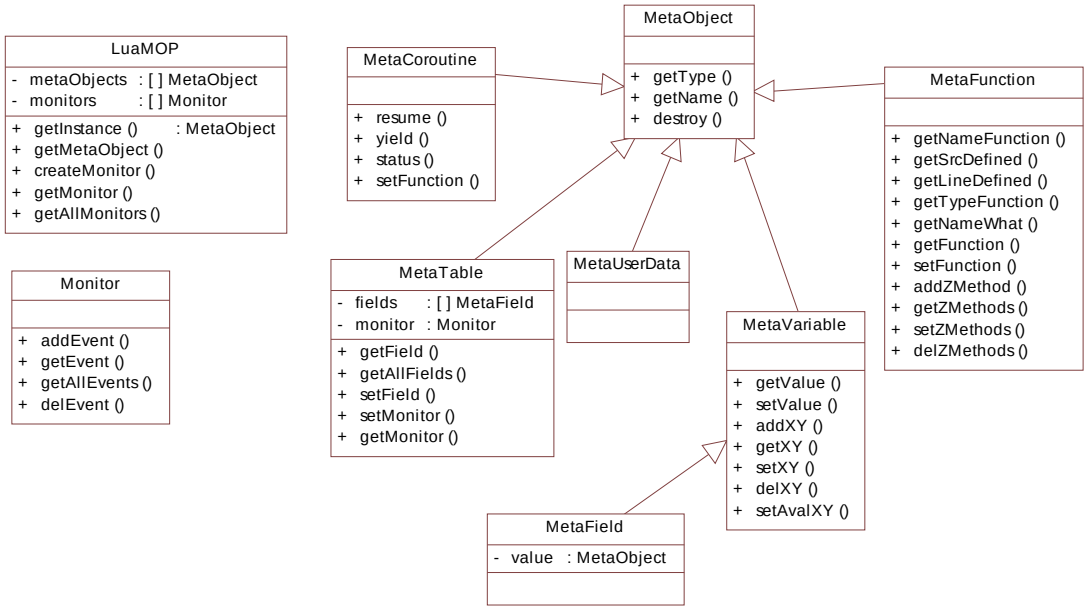

Figure 4: LuaMOP class diagram. X should be replaced by Pre and Pos, Y should be replaced by Get and Set and Z should be replaced by Pre, Pos or Wrap

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, *balance* variable is set to 10. On the next line, a meta-object is created to represent such 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 such 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.

```
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

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.

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

An example of the use of the functions provided by the *Function* meta-object is illustrated in Figure 6. Initially the *sum* and *newsum* functions are defined. Next, the *meta-function* meta-object is get and on the next line the *getNameFunction* method is invoked. It returns the "*sum*" value. On line 4, the *setFunction* method is invoked to modify the implementation of the *sum* function via the *newsum* function. This way, when the *getNameFunction* method is invoked, on line 5, it returns *newsum* instead of *sum*.

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 line 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 is added 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 line 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 *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 shows 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 aspects 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 advices).

Figure 10 illustrates 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, the *logfunction* function acts as an aspect to the *deposit* function. For each *deposit* function invocation, *logfunction* function is invoked *before* it in order to print the *deposit* value.

```
Bank = {balance = 0}
function Bank:deposit(amount) self.balance = self.balance + amount end
function logfunction(a) print('It was deposited: ' .. a) end
asp = Aspect:new()
id = asp:aspect( {name = 'logaspect'},
          {pointcutname = 'logdeposit', designator = 'call',  list = {'Bank.deposit'}},
          {type = 'before', action = logfunction} )
oldasp = asp:getAspect(id)
oldasp.advice.type = 'after'
asp:updateAspect(id, oldasp)
```

Figure 10: Example of aspect definition

The aspect identification (id) returned by the *aspect* method can be used to control the defined aspects. To handle this issue, 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 *advices* of an aspect already defined. Aspect removal can be done by using the *removeAspect* method.

In Figure 10, the *Bank* object with *deposit* method is declared before the invocation of *aspect* method. If *Bank* object has not been declared, AspectLua would consider it as a *virtual join point* – a join point that does not have a meta-object but that has a monitor. A *virtual join point* can be useful in many situations. For instance, for a dynamic resource allocation in embedded systems, virtual join points can be used to support a lazy loading approach [Miles 2004]. 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 *virtual 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* function 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

3.3. Integration between AspectLua and MOP

AspectLua exploits the power of LuaMOP and uses it to the definition of aspects, pointcuts and advices. In LuaMOP perspective, 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 aspects definition.

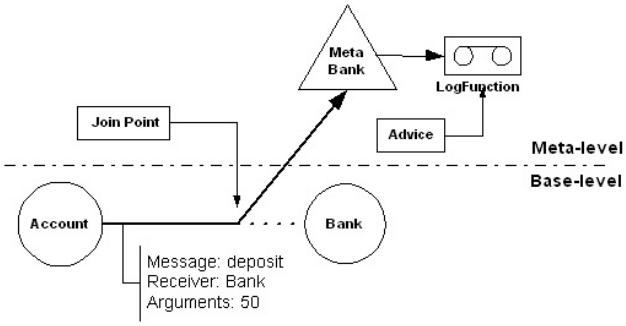

Figure 12: Integration between AspectLua and LuaMOP

Figure 12 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 12, *MetaBank* executes *LogFunction* function and after that, the *deposit* function executes.

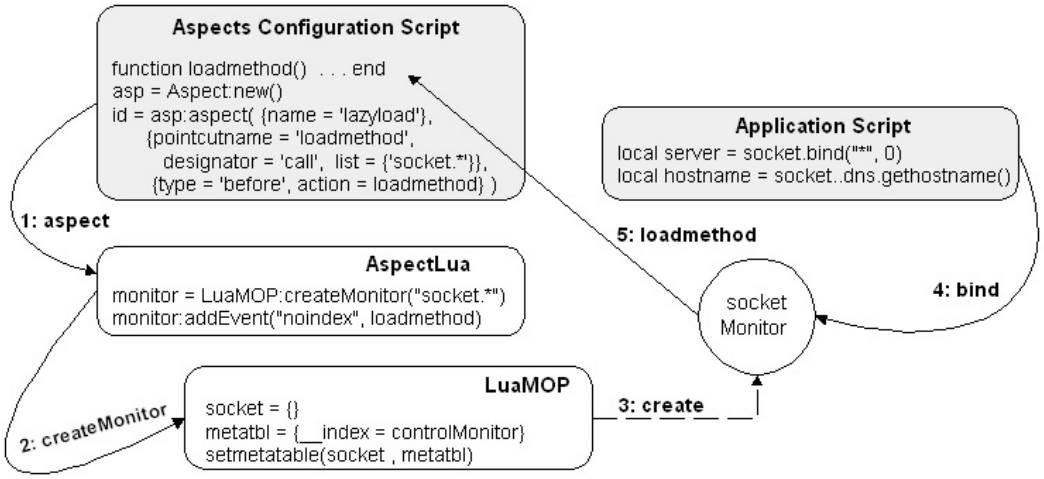

Figure 13: Virtual Join Point implementation

AspectLua also uses *monitors* to support the definition of aspects associated with undeclared elements. Monitors act on the interception of *virtual join points* and on the execution of its associated *advices*. Figure 13 shows the steps involved in the

definition and execution of a *virtual 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 *virtual 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. X and Y functions were used in the comparison and 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.

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.

4. Related Works

Related works include some AOP languages built on top of scripting languages. The most important are three AOP extensions based on well-known scripting languages: Python [Rossum 2003], Ruby [Thomas and Hunt 2000] and Smalltalk [Goldberg and Robson 1983]. AOPy [Dechow 2003] 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 [Bryant and Feldt 2002] is built on top of Ruby. It implements AOP by wrapping code around existing methods in classes and supports wildcards. AspectS [Hirschfeld 2002] 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 (*virtual join points*). [Miles 2004] provides a similar mechanism that does not use the actual idea of *virtual join points* because this approach uses a *proxy* to represent the join points.

LAC – Lua Aspectual Component [Herrmann and Mezini 2001] – is a Lua extension whose main goal is to support the idea of Aspectual Components (AC) [Lierberherr et al. 1999]. LAC is quite different of AspectLua because it 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.

Interceptors of 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, advices are inserted by registering callback functions and follow a lot of constraints to avoid infinite recursions.

5. 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 and 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.

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 *virtual 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 elements statically defined.

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.

5. References

Bryant A. and Feldt R. (2002) "AspectR - Simple aspect-oriented programming in Ruby", Available at http://aspectr.sourceforge.net/.

Dechow, D. (2003) "Advanced Separation of Concerns for Dynamic, Lightweight Languages", In: 5th Generative Programming and Component Engineering.

Gal, A., Schröder-Preikschat, W. and Spinczyk, O. (2001) AspectC++: Language Proposal and Prototype Implementation. University of Magdeburg.

Goldberg, A. and Robson, D. (1983) Smalltalk-80: The Language and Its Implementation. Addison-Wesley.

Herrmann S. and Mezini M. (2001) "Combining Composition Styles in the Evolvable Language LAC", In: ASoC Workshop in ICSE — International Conference on Software Engineering.

Hirschfeld, R. (2002) "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, pp. 216-232, LNCS Vol. 2591, ISBN:3-540-00737-7, Springer-Verlag, London, UK.

Ierusalimsky, R., Figueiredo, L. H., and Celes, W. (1996) "Lua – an extensible extension language. Software: Practice and Experience, 26(6):635-652.

Kiczales, G., des Rivieres, J. and Bobrow, D. (1991) "The Art of the Metaobject Protocol", MIT Press.

Kiczales, G., Lamping, J., Mendhekar, A. et al. (1997) "Aspect-oriented programming", In: ECOOP'97 — European Conference on Object-Oriented Programming", Proceedings of ECOOP´97. Springer-Verlag, Finland.

Kiczales, G., Hilsdale, E., Hugunin, J. et al. (2001) "An Overview of AspectJ", In: ECOOP'2001 — European Conference on Object-Oriented Programming, Budapest, Hungary.

Lierberherr, K., Lorenz, D. and Mezini M. (1999) "Programming with Aspectual Components", In: Technical Report NU-CCS99 –01, Notheastern University.

Miles, R. (2004) "Lazy Loading with Aspects", ONJava.com, available at: http://www.onjava.com/pub/a/onjava/2004/03/17/lazyAspects.html

Rossum, G. v. (2003) "Python Reference Manual", Available at http://www.python.org/doc/current/ref/ref.html.

Thomas D. and Hunt A. (2000) "Programming Ruby: A Pragmatic Programmer's Guide", Available at http://www.rubycentral.com/book/.