AspectLua - A Dynamic AOP Approach

Nélio Cacho
(Federal University of Rio Grande do Norte, Brazil
cacho@consiste.dimap.ufrn.br)

Thaís Batista
(Federal University of Rio Grande do Norte, Brazil
thais@ufrnet.br)

Fabrício Fernandes
(Federal University of Rio Grande do Norte, Brazil
fabricio@consiste.dimap.ufrn.br)

Abstract: In this paper we describe AspectLua – a dynamic aspect-oriented language based on Lua. It relies on a meta-object protocol, LuaMOP, which unifies the introspective and reflective mechanisms provided by Lua and handles the weaving process. In order to improve support for dynamicity, AspectLua allows the association of aspects with undeclared elements of the application code (*virtual join points*). In addition, it provides an automatic support for managing aspects execution order.

Keywords: AOP, MOP, Lua, Dynamic Aspects, Reflection.
Categories: D.2.3, D.3.3

1 Introduction

Aspect-oriented programming (AOP) has gained popularity as a powerful mechanism to promote modularity by separating the code that crosscuts components from the component code (base code). An *aspect* is a unit that encapsulates *crosscutting concerns*. A *weaving process* supports the composition of components and aspects. Weaving can take place at compile time (static weaving) or at runtime (dynamic weaving). Recently, some approaches are using weaving at loading time. While static weaving avoids type mismatches and runtime overhead, it does not fit well in the support of a common requirement of nowadays applications: dynamic adaptability. In order to meet this requirement, a number of AOP approaches adopt weaving at runtime. Building dynamic weaving on top of a scripting language is commonplace. There are AOP implementations of Python [Rossum 03], Ruby [Thomas 00], Perl [Liang 04], and Smalltalk [Ingalls et al. 97].

In this paper we present *AspectLua* - an AOP extension to the Lua [Ierusalimsky et al. 96] scripting language. Lua offers a fertile substract for the integration of aspects because it is dynamically typed and offers reflective features that allows the programmer to extend its behavior without modifying the underlying interpreter. In addition, the Lua philosophy is to be simple and small. AspectLua follows this idea. AspectLua shares some features with other AOP scripting languages. For instance, weaving is done at runtime and both components and aspects can be dynamically

inserted into and removed from the application. On the other hand, it combines some features found in isolation in other languages and also introduces the possibility of specifying aspects for inexistent application elements (*virtual join points*). We consider that this concept is essential to handle adaptability.

Since in a highly dynamic scenario aspects are inserted into and removed from the application at runtime, the problem of aspects execution order is highlighted. In order to overcome this problem, AspectLua API offers functions that allow the programmer to manage aspects execution order. It also provides an automatic support for this task via the concept of *alert*. The programmer can define an alert to a join point and associate a function that is invoked when a new aspect is defined to this join point.

We also present *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 this concept.

This paper is organized as follows. Section 2 presents aspect-oriented programming concepts and a technique commonly used to support it: computational reflection. Section 3 presents AspectLua architecture and functionality. Section 4 presents a case study that validates AspectLua concepts. Section 5 discusses performance evaluation. Section 6 discusses about related work. Finally, section 7 contains the final remarks.

2 Aspect-Oriented Programming

2.1 Basic Concepts

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 [Kiczales et al. 01] because it is one of the most used 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. 97]. Join points can represent data flows of the component program, runtime method invocations in the component program, etc. *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.

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 either at compile time or at runtime. In AspectJ and AspectC++ [Gal et al. 01] weaving is done at compile time. A recent version of AspectJ uses 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 and Computational Reflection

The capability of a programming language to support dynamic aspects depends on its mechanisms to recognize join points and to deal with advices insertion at runtime. The recognition of join points and the introduction of new behaviors (advices) can be implemented using computational reflection. Some works [Kojarski et al. 03], [Sullivan 01] discusses the relationship between reflection and AOP.

Reflection [Kiczales et al. 91] 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. Thus, changes at the meta-level are reflected into corresponding modifications at the base-level and vice-versa. 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 whose semantics can be 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 base-level objects.

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

3 AspectLua Infrastructure

3.1 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* supports the modification of a table behavior. 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* is 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: _G example

In this code, 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 makes it possible the use of a *metatable* to control the reading and writing in global variables.

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

3.2 AOP in Lua

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. According to [Sullivan 01] a robust MOP makes it easier to implement an aspect language.

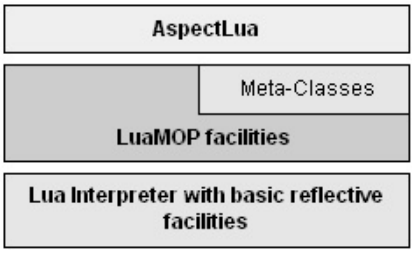

Figure 3: AspectLua architecture

Figure 3 illustrates the blocks that compose the *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. Thus, AspectLua offers an abstraction layer that hides the complexities of meta-objects. A programmer can take advantage of AspectLua without knowing either LuaMOP or the Lua reflective features.

3.2.1 Creating Aspects

To create an aspect it is necessary to use the *new* function that creates an instance of the AspectLua class. After that, to define a Lua table containing the aspect elements (name, pointcuts, and advices). Figure 4 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. This list can use wildcards. For instance,

*Bank.** means that the aspect should be applied for all methods of the *Bank* class. It is not necessary that the elements to be intercepted have been already declared.

- 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 4, 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} )
```

Figure 4: Aspect definition

In Figure 4, the *Bank* object with *deposit* method is declared before the invocation of the *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 dynamic resource allocation in embedded systems, virtual join points can be used to support a lazy loading approach [Miles 04]. See in Figure 13 a simple lazy loading scenario.

3.2.2 Managing Aspects

AspectLua provides a set of functions to manage aspects: *getAspect(id), getAll(), removeAspect(id),* and *updateAspect(id, newasp).* Such functions use the *aspect identification - id -* returned by the *aspect* method. *getAspect* and *getAll* are used to obtain one or all aspects already defined. After obtaining an aspect, it is possible to modify it or to update its elements by using the *updateAspect* function. Using this function it is possible to modify *pointcuts* and *advices* of an aspect already defined. *removeAspect* supports aspect removal. Figure 5 illustrates the use of these functions.

To control the execution order of aspects in a given pointcut, AspectLua offers *getOrder* and *setOrder* functions. *getOrder* returns 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* allows the programmer to modify this order. This function receives the following parameters: variable or function name and the new execution order. Figure 5 defines two aspects

to be executed before the *deposit* method. By default, the execution order follows 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()
      if not ok_right then
        error("No Permission to execute")
      end
end
id = asp:aspect( {name = 'secaspect'},
              {pointcutname = 'verifyRights',
              designator = 'call', list =  {'Bank.deposit'}},
              {type = 'before', action = checkRights } )
local order = Aspect:getOrder('Bank.deposit')
oldasp = asp:getAspect(id)
oldasp.advice.type = 'after'
asp:updateAspect(id, oldasp)
Aspect:setOrder('Bank.deposit', { order[2], order[1]})
```

Figure 5: Defining order to aspects invocations

As different aspects can be defined to a given join point, the problem of determining the execution order is commonplace. In Figure 5, the programmer uses the *setOrder* function to fix this problem. However, in complex applications with a great number of aspects, the identification of the different aspects that act on the same pointcut can be a hard task. In order to give an automatic support for this task, AspectLua provides the following methods: *addAlert(type, joinpoint, func*), *removeAlert(ida)* and *getAlert(ida). addAlert* receives a *type* of alert (Add, Remove, Update, Equal), the join point and a function to be invoked to handle the alert. For instance, *addAlert("Add", "Bank.deposit", warning)* defines an alert that invokes the *warning* method whenever an aspect is added to the *Bank.deposit* join point. In order to support alerts management**,** the *addAlert* method returns an alert identifier (*ida*) to be used by the *getAlert* and *removeAlert* methods. These methods are used to get and to remove an alert, respectively.

Similary to the *Add* alert type, which determines the monitoring of the *aspect* method invocations, *Remove* and *Update* determine the monitoring of *removeAspect* and *updateAspect.* In contrast, the *Equal* alert type determines the monitoring of *aspect* or *updateAspect.* In this case, the monitoring consists in verifying if during the execution of these methods, a new aspect is defined to a same join point that is already associated with another aspect. In this way, the verification process returns *true* from the second aspect of a same join point onwards.

3.2.3 Implementations details

Figure 3 illustrates the architectural view of the AspectLua. Figure **6** shows how the AspectLua architecture is implemented. AspectLua internal structure manages the

meta-objects and the alerts. AspectLua is used by invoking methods provided by its API (aspect, removeAspect, updateAspect, etc). Each method invocation is initially forwarded to the Alert Manager that verifies the join point and the invoked method in order to discover Alerts. After that, the invocation is handled by the AspectLua *core* that combines each join point with the proper meta-object. This relationship is done by the *Aspect list* and indexed using the *id* of the aspect. This process controls the instantiated meta-objects for the join points. Each join point can be represented by one or more meta-objects.

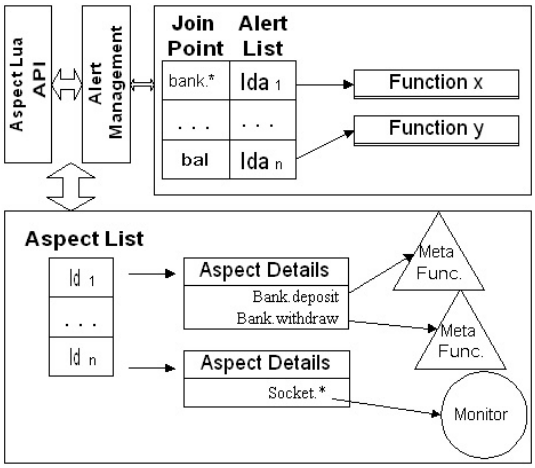

Figure 6: AspectLua internal architecture

To instantiate and manipulate meta-objects, AspectLua relies on LuaMOP. LuaMOP is a meta-object protocol that enables 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 7). 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.

The flexibility provided by AspectLua in defining join points to different elements (variables, functions, tables) is supported by the set of meta-objects provided by LuaMOP. This way, 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 8 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 is 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 8, *MetaBank* executes *LogFunction* function and after that, the *deposit* function is executed.

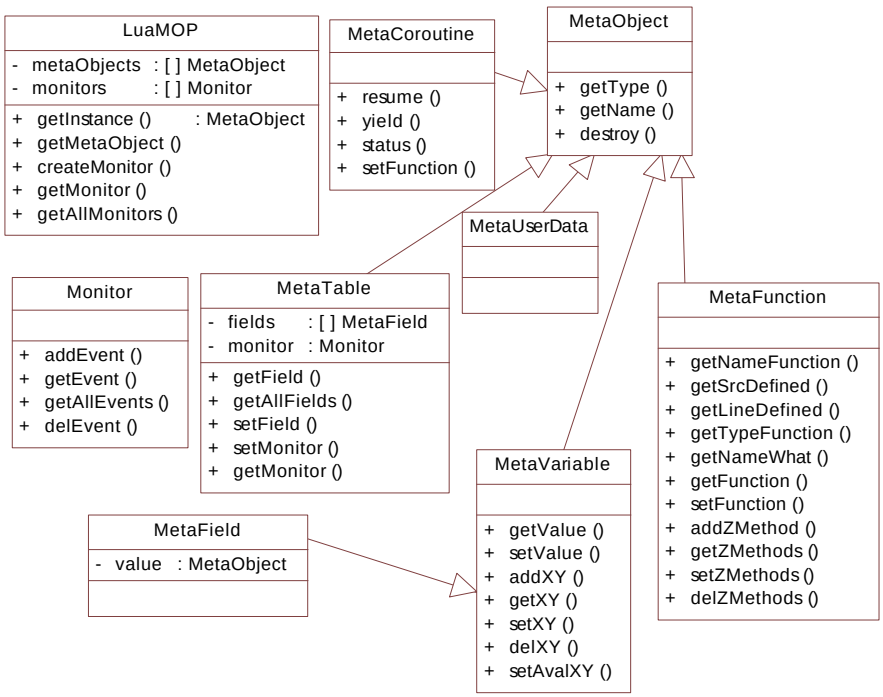

Figure 7: LuaMOP class diagram. X should be replaced by Pre and Pos, Y should be replaced by Get and Set and Z should be replaced by Pre, Pos or Wrap

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 support changes in 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.

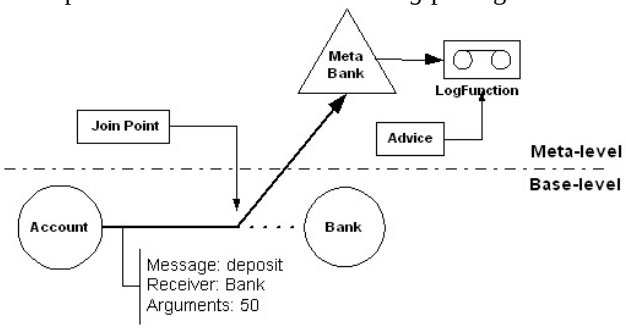

Figure 8: Integration between AspectLua and LuaMOP

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 9 shows an example 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.

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 9 shows the use of *addPosSet* method that is invoked to associate the *writelog* function with the *metavar* meta-object. When the *balance* variable receives a new value, the *writelog* function is invoked. 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 respectively used to get all functions associated with *Pre/Pos* and *Get/Set*, to determine a new set of functions, 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 9: 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)* supports the 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 10: LuaMOP example with setFunction

An example of the use of the functions provided by the *Function* meta-object is illustrated in *Figure 10*. 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. In this way, when the *getNameFunction* method is invoked (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 11.

```
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 11: 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 changes in the base objects. Figure 12 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 12: 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 13 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 the LuaSocket library. Thus, methods of the *socket* object, such as *bind* and *connect*, that are not yet available in the execution environment, is 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.

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 *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. The same functionality can be used by AspectLua to abstract away the use of Monitors. For instance, the following code implements the same functionality of Figure 13: *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.**).

```
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 13: Using LuaMOP's Monitor

4 Case Study

To illustrate the use of AspectLua we have implemented a case study of a banking application with fault tolerance support. The application is composed of two different components: a client and a server. Fault tolerance is based on the use of replicated servers.

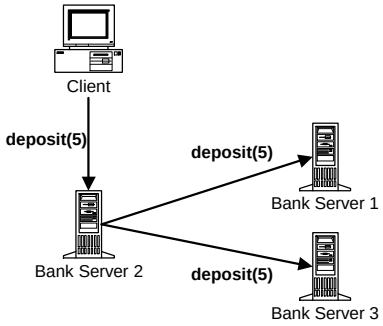

Figure 14: Replication Process

Figure 14 shows how the replication process works. A client invokes the deposit method in *BankServer2*. *BankServer2* processes the invocation and, through a crosscutting code, forwards the request to *BankServer1* and *BankServer3*. This replication implies that all servers must contain the same value. This approach is useful for fault tolerance because if *BankServer2* is unavailable, the client-side can be

invoked to forward the request to *BankServer1* or *BankServer3*. Figure 15 illustrates the *deposit* method invocation from client to the *bank* server object. In this case, if the *bank* object becomes unavailable, the application fails.

```
bank = luaorb.createproxy(readIOR("./account.ref"),"IDL:Account:1.0")
bank:deposit(5)
```

Figure 15: Client code

To overcome this problem, we insert an aspect in the *bank* object for searching other implementations instead of returning an error to the client.

```
1 function trynewreferences(self, ...)
2       newbank = Generic()
3       table.remove(arg, 1)
4       newbank:deposit(unpack(arg))
5  end
6 client = Aspect:new()
7 client:aspect({name = clienteIntercept'},
            {name = 'replicationMethods',
      designator = 'call',
      list = { 'bank.deposit'} },
            {type = 'around', action = trynewreferences })
```

Figure 16: Client code

Figure 16 shows the code of the *trynewreferences* function. On line 2 the client creates a generic connector [Batista and Carvalho 02] – a mechanism that dynamically selects components to provide services required by an application. On line 4 the *deposit* method is invoked. The function of the generic connector is to find an element that implements the *deposit* method. An aspect that acts on the *deposit* method of the *bank* object is defined on line 7. It executes the *trynewreferences* function instead of executing the *deposit* method.

Figure 17 shows *bank_impl* table that represents the implementation of the IDL:Account:1.0 CORBA interface. The *deposit* method is described by the code on line 4. Finally, on line 6, the *bank_impl* implementation is registered at the searching mechanism.

In order to dynamically adapt the server to include replication we define two aspects. The first aspect defines that all invocations to the *deposit* method of B*ank* should be followed by the execution of the *replication_deposit* method. Another aspect is necessary to introduce the implementation of the *deposit_rep* function. This function is invoked to replicate the information. This method is necessary to avoid recursion among several servers. Since *deposit_rep* method does not exist in the original *Account* interface, previously defined, the IDL with this function must be loaded. This is illustrated on the first line of the code of Figure 18.

```
1 bal = 0
2 Bank = {
3          deposit = function(self,amount)
4                    bal = bal + amount
5          end}
6 source_server,id = ls_createservant(Bank,
                              "IDL:Account:1.0")
```

Figure 17: Server code without replication – coreServer.lua

The *replication_deposit* function represents the replication process. Lines 4 and 5 get the reference of the *discovery service* [Cacho et al. 04]. On line 7 the *search* function is used to find out servers that have the *deposit_rep* operation and whose *id* is different from the current server *id* (expressed by the *offered* variable). Next, the *deposit_rep* method is invoked for all servers described at the *search_result* table.

```
1 os.execute("idl  --feed-ir -ORBIfaceRepoAddr inet:localhost:15000
                  account_rep.idl")
2 dofile("AspectLua.lua")
3 function replication_deposit(self,amount)
4   local proxy_Discovery=luaorb.createproxy(readIOR("./search.ref"),
                              "IDL:CosDiscovering/SearchComponents:1.0")
5          proxy_Lookup = proxy_Discovery.getLookup
6          search_result = {}
7          search_result = proxy_Lookup:search(
                                   "(operationame =='deposit_rep')and
                                    (offerId != "..id..")")
8          replication = Generic()
9          for i,refid in ipairs(search_result) do
10                   replication:deposit_rep(amount)("offerId=="..refid)
11         end
12 end
13 function deposit_rep(self,amount)
14          bal = bal + amount
15 end
16 a = Aspect:new()
17 a:aspect({name = 'AccountDeposit'},
      {name = 'replicationMethods',  designator = 'call',   list = {'Bank.deposit'}},
            {type ='before', action = replication_deposit})
18 a:aspect({name = 'AccountDeposit_rep'},
       {name = 'replicationMethods',   designator='introduction',
            list = {'Bank.deposit_rep'} },
           {type ='after', action = deposit_rep})
```

Figure 18: Replication aspects – aspectServer.lua

After defining the aspect code, the next step is to specify the application configuration file. This file defines the elements that compose the application: the

server code (coreServer.lua) and, optionally, the replication code. The user can choose to include replication or not (Figure 19).

```
dofile("coreServer.lua")
print("Would you like to use replication ?(y,n)")
answer = io.read()
if (answer == "y") then
        dofile("aspectServer.lua")
end
```

Figure 19: Application configuration file

The possibility of choosing between inserting or not an aspect can cause problems when different aspects acts on a same join point and when the execution order can interfere in the final result. For instance, when executing the code illustrated in Figure 5 the result can be unpredictable because it is not known if the replication aspect will be applied or if the replication aspect use the setOrder function to modify the execution order. Thus, it can be dangerous to use dynamic insertion of aspects instead of using crosscutting concerns, especially when the execution order can interfere in the final result. To overcomme this problem, in Figure 20 an alert is defined to handle all invocations to the bank_impl.deposit join point Thus, when the aspectServer.lua file is invoked, the fail_security function prints the following message: “Fail security (probably): aspectServer.lua, addAspect”. The aldetail parameter is provided by AspectLua to describe the method that is the source of an alert. Alerts allow the programmer to be aware about the definition of aspects to a given join point. As a consequence, the programmer can identify possible interferences between aspects and define the proper execution order.

```
1 function fail_security (self, aldetail)
2         print(“Fail security (probably):”, aldetail.file, aldetail.opname)
3 end
4 Aspect: addAlert(“*”, “Bank.deposit”, fail_security)
```

Figure 20: Defining a security Alert

5 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 needed by the meta-object 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, on 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 (on the previous line) to 6.7μs. This difference is related to the amount of time involved in loading the functions associated with the *Metafield*.

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

Table 1: Performance Evaluation. Time in μs.

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.

6 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 [Rossum 03], Ruby [Thomas and Hunt 00] and Smalltalk [Goldberg and Robson 83]. AOPy [Dechow 03] 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 02] is built on top of Ruby. It implements AOP by wrapping code around existing methods in classes and supports wildcards. AspectS [Hirschfeld 02] 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 04] provides a similar mechanism that does not use the idea of *virtual join points* because this approach uses a *proxy* to represent the join points. In addition, AspectLua includes alerts as a mechanism to help the programmer to manage aspects execution order.

LAC – Lua Aspectual Component [Herrmann and Mezini 01] – is a Lua extension whose main goal is to support the idea of Aspectual Components (AC) [Lierberherr et al. 99]. 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.

PROSE (*PROgrammable extenSions of sErvices*) [Popovici et al. 02] is a platform based on Java, which supports dynamic AOP. As AspectLua, PROSE does not introduce a new syntax for defining aspects. It uses the Java language itself. Aspects are created by writing Java classes based on the PROSE library. In PROSE, as in AspectLua, aspects can be woven and unwoven at rutntime. In the same way, there is no need of a special compiler. It also offers an Aspect Monitor tool that shows a tree-like structure of aspects describing the crosscut objects and the join-points in the running application. 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. Among a lot of similarities, there are some important differences from PROSE to AspectLua: (i) AspectLua uses a pure interpreted approach; (ii) PROSE does not offer a rich API including methods for monitoring aspects definition and for determining aspect execution order.

In terms of dynamic insertion of new funcionalities, 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.

7 Final Remarks

In this paper we presented AspectLua - an AOP extension to the Lua language that shares some common features of other AOP scripting languages and introduces some new mechanisms to give a powerfull support to dynamic adaptability Aspects are defined using AspectLua that relies on LuaMOP that supports the dynamic weaving by exploiting 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 component 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. In addition, this work proposed the idea of applying alerts to give an automatic support for managing aspects precedence order.

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.

Acknowledgements

This work was supported by the Brazilian Research Council, CNPq, under process 55.2007/02-1

References

[Batista and Carvalho 02] Batista, T., Carvalho, M.: "Component-Based Applications: A Dynamic Reconfiguration Approach with Fault Tolerance Support". In Software Composition Workshop (SC) - affiliated to European Joint Conferences on Theory and Practice of Software (ETAPS), Grenoble - FR, April 2002. Published in Electronic Notes in Theorical Computer Science (ENTCS), Vol. 65, Number 4, (2002).

[Bryant and Feldt 02] Bryant, A., Feldt, R.: "AspectR - Simple aspect-oriented programming in Ruby", Available at http://aspectr.sourceforge.net/, (2002)

[Cacho et al. 04] Cacho, N., Batista T., Elias, G.: "Um Serviço CORBA para Descoberta de Componentes". 18th Brazilian Symposium on Software Engineering (SBES'2004). Brasília, DF, (2004), 273-288.

[Dechow 03] Dechow, D.: "Advanced Separation of Concerns for Dynamic, Lightweight Languages", In: 5th Generative Programming and Component Engineering (GPCE) Young Researchers Workshop Available at http://se.inf.ethz.ch/events/gpce_yrw03/program/program.html, (2003).

[Gal et al. 01] Gal, A., Schröder-Preikschat, W., Spinczyk, O.: "AspectC++: Language Proposal and Prototype Implementation". University of Magdeburg. (2001)

[Goldberg and Robson (83)] Goldberg, A., Robson, D.: "Smalltalk-80: The Language and Its Implementation". Addison-Wesley, (1983).

[Herrmann and Mezini 01] Herrmann S., Mezini M.: "Combining Composition Styles in the Evolvable Language LAC", In: Workshop on Advanced Separation of Concerns in Software Engineering (ASoC) at the 23nd International Conference on Software Engineering (ICSE), (2001).

[Hirschfeld 02] Hirschfeld, R.: "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, , Lecture Notes in Computer Science (LNCS), Vol. 2591, Springer-Verlag, London, UK, (2002), 216-232.

[Ierusalimsky et al. 96] Ierusalimsky, R., Figueiredo, L. H., Celes, W.: "Lua – an extensible extension language". Software: Practice and Experience, 26(6), (1996), 635-652,

[Kiczales et al. (91)] Kiczales, G., des Rivieres, J., Bobrow, D.: "The Art of the Metaobject Protocol", MIT Press, (1991).

[Kiczales et al. 97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: "Aspect-Oriented Programming", In: ECOOP'97 — European Conference on Object-Oriented Programming", Proceedings of ECOOP´97. Springer-Verlag, Finland, (1997)

[Kiczales et al. 01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: "An Overview of AspectJ", In Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01). Lecture Notes in Computer Science, vol. 2072, J. Knudsen, Ed. Budapest, Hungary, (2001), 327-353.

[Kojarski et al. 03] Kojarski, S., Lieberherr, K., Lorenz, D.H., Hirschfeld, R.: (2003) "Aspectual Reflection". In Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), Boston, MA, USA, March 18, (2003), 17-21.

[Liang 04] Liang, C.: "Programming language concepts and Perl". *J. Comput. Small Coll.* 19, 5 May (2004), 193-204.

[Lierberherr et al. 99] Lierberherr, K., Lorenz, D., Mezini M.: (1999) "Programming with Aspectual Components", In: Technical Report NU-CCS99 –01, Notheastern University.

[Miles 04] Miles, R.: "Lazy Loading with Aspects", ONJava.com, available at: http://www.onjava.com/pub/a/onjava/2004/03/17/lazyAspects.html, (2004).

[Popovici et al. 02] Popovici, A., Gross, T., Alonso, G. (2002) "Dynamic Weaving for Aspect-Oriented Programming" In Proceedings of Int'l Conference on Aspect-Oriented Software Development (AOSD'02), ACM Press, (2002), 141-147.

[Rossum 03] Rossum, G. V.: "Python Reference Manual", Available at http://www.python.org/doc/current/ref/ref.html. (2003)

[Sullivan 01] Sullivan, G.: "Aspect-Oriented Programming using Reflection and Metaobject Protocols". Communications of the ACM. Vol., 44, Issue 10, October (2001), 95-97.

[Thomas and Hunt 00] Thomas D., Hunt A.: (2000) "Programming Ruby: A Pragmatic Programmer's Guide", Available at http://www.rubycentral.com/book/. (2000)