LuaCOM User Manual

Version 1.0 (beta2)

Vinicius Almendra Renato Cerqueira

3rd December 2002

Contents

1 Introduction

LuaCOM is an add-on library to the Lua language that allows Lua programs to use and implement objects that follow Microsoft's *Component Object Model* (COM) specification **and** use the *ActiveX technology* for property access and method calls.

2 Features

Currently, the LuaCOM library supports the following features:

- dynamic creation of COM objects registered in the system registry, via the `luacom_CreateObject` function;
- dynamic access to running COM objects via the `luacom_GetObject`;
- COM method calls as normal Lua function calls;
- property access as normal table field access;
- type conversion between OLE Automation types and Lua types for almost all types;
- object disposal using Lua garbage collection mechanism;
- implementation of COM interfaces and objects using Lua tables;
- use of COM connection point mechanism to handle bidirecional communication and event handling;
- fully compatible with Lua 4.

3 How to use

Using LuaCOM is straightforward: you just have to link your program with LuaCOM's library, include the LuaCOM's header — `luacom.h` — and call the propter initialization and termination functions before using any of LuaCOM's functionalities. Here is an example of a simple C program using LuaCOM.

```
/*
 * Sample C program using luacom
 */
#include <stdio.h>
#include <ole2.h> // needed for OleInitialize and OleUninitialize
#include <lua.h>

#include "luacom.h"

int main (int argc, char *argv[]) {

  /* COM initialization */
  CoInitialize(NULL);

  /* library initialization */

  lua_State *L = lua_open(0);

  luacom_open(L);

  if(lua_dofile("activex_sample.lua") != 0) {
    puts("Error running sample.lua!");
    exit(1);
  }
  luacom_close(L);
  lua_close(L);

  CoUninitialize(NULL);
  return 0;
}
```

Notice that it's necessary to initialize COM before `luacom_open` and to "uninitialize" only after the last `lua_close`, otherwise exceptions will occurr.

4 LuaCOM Elements

LuaCOM is composed by the following elements:

- LuaCOM API, used primarily to initialize the library, create objects, implement ActiveX interfaces in Lua and to manipulate connection points;
- LuaCOM objects, which make available in Lua ActiveX objects and interfaces;
- ActiveX binding, which translates accesses on LuaCOM objects to ActiveX interface calls *and* ActiveX accesses on an inteface implemented in Lua to Lua function calls or table accesses;

- LuaCOM type conversion rules, which govern the type conversion between Lua and ActiveX values;
- LuaCOM parameters passing rules, which describe how LuaCOM translate a Lua parameter list to a COM one and vice versa.

4.1 LuaCOM API

Currently, the LuaCOM API is divided in two parts: the Lua API and the C/C++ API. The C/C++ API is used primarily for initialization of the library and for low-level construction of LuaCOM objects. The Lua API permits Lua programs to access all the functionality of LuaCOM. Below there is summary of the LuaCOM API. Detailed information on these functions is available at section 6.

Lua API

Function	Description
`luacom_CreateObject`	Creates a LuaCOM object
`luacom_NewObject`	Creates a LuaCOM object implemented in Lua
`luacom_GetObject`	Creates a LuaCOM object associated with an instance of an already running ActiveX object
`luacom_ExposeObject`	Exposes a LuaCOM object, so that other applications can get a reference to it
`luacom_RevokeObject`	Undoes the operation of `luacom_ExposeObject`
`luacom_RegisterObject`	Fills in the registry entries necessary for exposing a COM object.
`luacom_Connect`	Creates a connection point between an object and a Lua table
`luacom_ImplInterface`	Implements an IDispatch interface using a Lua table
`luacom_ImplInterfaceFromTypelib`	Implements an IDispatch interface described in a Type Library using a Lua table
`luacom_addConnection`	Connects two LuaCOM objects
`luacom_releaseConnection`	Disconnects a LuaCOM object from its connection point
`luacom_isMember`	Checks whether a name correspond to a method or a property of an LuaCOM object
`luacom_ProgIDfromCLSID`	Gets the ProgID associated with a CLSID
`luacom_CLSIDfromProgID`	Gets the CLSID associated with a ProgID
`luacom_GetIUnknown`	Returns an `IUnknown` interface to a LuaCOM object as a userdata.

C/C++ API

Function	Description
`luacom_open`	Initializes the LuaCOM library in a Lua state. It must be called before any use of LuaCOM features.
`luacom_close`	LuaCOM's termination function
`luacom_IDispatch2LuaCOM`	Takes an IDispatch interface and creates a LuaCOM object to expose it, pushing the object on the Lua stack.

4.2 LuaCOM objects

LuaCOM deals with *LuaCOM objects*, which are no more than a Lua table with the LuaCOM tag and a reference to the LuaCOM C++ object; this one is, in turn, a proxy for the ActiveX object: it holds an `IDispatch` pointer to the object and translates Lua accesses to ActiveX calls and property accesses. Here is a sample where a LuaCOM object is used:

```
-- Instantiate a Microsoft(R) Calendar Object
calendar = luacom_CreateObject("MSCAL.Calendar")

-- Error check
if calendar == nil then
  print("Error creating object")
  exit(1)
end

-- Method call
calendar:AboutBox()

-- Property Get
current_day = calendar.Day

-- Property Put
calendar.Month = calendar.Month + 1

print(current_day)
print(calendar.Month)
```

LuaCOM objects can be created using the LuaCOM Lua API; there are a number of function that return LuaCOM objects. The most relevant ones are `luacom_CreateObject` and `luacom_GetObject`. Then may also be created on implicitly, when a return or output value of a COM method is a dispinterface.

LuaCOM objects are released through Lua's garbage collection mechanism, so there isn't any explicit API function to destroy them.

A LuaCOM object may be passed as an argument to method calls on other LuaCOM objects, if these methods expect an argument of type dispinterface. Here is a sample to ilustrate this situation:

```
-- Gets a running instance of Excel
excel = luacom_GetObject("Excel.Application")

-- Gets the set of worksheets
sheets = excel.Worksheets

-- gets the first two sheets
sheet1 = sheets.Item(1)
sheet2 = sheets.Item(2)

-- Exchange them (here we pass the second sheet as a parameter
-- to a method)
sheet1:Move(nil, sheet2)
```

4.3 ActiveX binding

The ActiveX binding is responsible for translating the table accesses to the LuaCOM object to ActiveX interface calls. Besides that, it also provides a mechanism for implementing ActiveX dispinterfaces using ordinary Lua tables.

4.3.1 Implementing dispinterfaces in Lua

The ActiveX binding has a C++ class that implements a generic `IDispatch` interface. The implementation of this class translates the method calls and property accesses done on the objects of this class to Lua calls and table accesses. So, one may implement an ActiveX interface entirely in Lua provided it has a type library describing it. This type library may be a stand-alone one (referenced by its location on the filesystem) or may be associated with some registered component. In this case, it may be refenced by the `ProgID` of the component.

The C++ objects of this class can be used in any place where a `IDispatch` or `IUnknown` interface is expected. Follows a sample of implementing ActiveX dispinterfaces in Lua.

```
-- Creates and fills the Lua table that will implement the
-- ActiveX interface

events_table = {}

function events_table:AfterUpdate()
  print("AfterUpdate called!")
end

-- Here we implement the interface DCalendarEvents, which is part
-- of the Microsoft(R) Calendar object, whose ProgID is MSCAL.Calendar

events_obj = luacom_ImplInterface(
  events_table,
  "MSCAL.Calendar",
  "DCalendarEvents")

-- Checks for errors
--
if events_obj == nil then
  print("Implementation failed")
  exit(1)
end

-- Tests the interface: this must generate a call to the events:AfterUpdate
-- defined above
--
events_obj:AfterUpdate()
```

If the interface to be implemented is described in a stand-alone type library, the function `luacom_ImplInterfac` must be used instead:

```
-- Creates and fills the Lua table that will implement the
-- ActiveX interface

hello_table = {}

function hello:Hello()
  print("Hello World!")
end

-- Here we implement the interface IHello
--
hello_obj = luacom_ImplInterfaceFromTypelib("hello.tlb","IHello")

-- Checks for errors
--
if hello_obj == nil then
  print("Implementation failed")
  exit(1)
end

-- Tests the interface
--
hello_obj:Hello()
```

Both functions return a LuaCOM object, whose corresponding ActiveX object is implemented by the supplied table. So, any Lua calls to this LuaCOM object will be translated to ActiveX calls which, in turn, will be translated back to Lua calls on the implementation table. This LuaCOM object can be passed as an argument to ActiveX methods who expect a dispinterface or to LuaCOM API functions (like `luacom_addConnection`).

One can also use the `luacom_NewObject` function, which is best suited to the situation where one needs to create a complete ActiveX object in Lua and wants to export it, so that it can be accessed through COM by any running application.

4.3.2 Using Methods and Properties

The ActiveX interfaces have two "types" of members: properties and methods. LuaCOM deals with both, although there may be some limitations not shown here (see chapter 5).

Method accesses are done in the same way as calling Lua functions stored in a table and having a "self" parameter:

```
obj = luacom_CreateObject("TEST.Test")

if obj == nil then
  exit(1)
end

-- method call
a = obj:Teste(1,2)

-- another one
obj:Teste2(a+1)
```

It's important to notice the need of using the colon – ":" – for method calls. Although LuaCOM does not use the `self` parameter that Lua passes in this case, its presence is assumed, that is, LuaCOM always skips the first parameter in the case of method calls. Forgetting it may cause nasty bugs.

Accessing properties is much like the same of accessing fields in Lua tables:

```
obj = luacom_CreateObject("TEST.Test")

if obj == nil then
  exit(1)
end

-- property access
a = obj.TestData

-- property setting
obj.TestData = a + 1
```

Here we have just the opposite: it's necessary to use a dot – "." – to access properties. Using a colon may also cause bugs, as LuaCOM blindly ignores (or uses) the first parameter, depending on the expected type of access (method call or property access)[1].

[1]LuaCOM gets this information, whether it's a property or a method, from the type information of the member being

Indexed Properties The ActiveX standard allows properties to have parameters. In fact, the properties are much like methods with a restricted form.

To read an indexed property, one must access the property as a method, passing the parameters, but using the dot, instead of the colon:

```
-- reading an indexed method
x = obj.Test(1)
```

To write to an indexed property is analogous, just passing the value to be set as another parameter, following the index:

```
-- write the property: first the index(es), them the value
obj.Test(1,"a")
```

It's important to notice that LuaCOM distinguishes the read and write accesses by the number of parameters.

Property Access in Lua When implementing a COM interface in Lua, LuaCOM also supports the concept of property and of indexed properties. LuaCOM translate property reads and writes to table field accesses:

```
interface = {}

interface.Test = 1
interface.TestIndex = {2,3}

obj = luacom_ImplInterface(interface, "TEST.Test", "ITest")

-- must print "1"
print(obj.Test)

-- must print nil (if there is no member named Test2)
print(obj.Test2)

-- this writes the filed Test
obj.Test = 1

-- Indexed property read. Must return 3 (remember that
-- indexed tables start at 1 in Lua)
i = obj.TestIndex(2)

-- Sets the indexed field
obj.TestIndex(2,4)

-- Now must return 4
i = obj.TestIndex(2)
```

used.

4.3.3 Connection Points

The *connection points* are a standard ActiveX mechanism whose primary objective is to allow the ActiveX object to notify its owner of any kind of events. The connection point works as an "event sink", where events and notifications go through.

To establish a connection using LuaCOM, the owner of the ActiveX object must create a table to implement the connection interface, whose description is provided by the ActiveX object (this interface is called a *source* interface) and then call the API function `luacom_Connect`, passing as arguments the LuaCOM object for the ActiveX object and the implementation table. Doing this, LuaCOM will automatically find the default source interface, create a LuaCOM object implemented by the supplied table and then connect this object to the ActiveX object. Here follows a sample:

```
-- Creates the ActiveX object
--
calendar = luacom_CreateObject("MSCAL.Calendar")

if calendar == nil then
  exit(1)
end

-- Creates implementation table
--
calendar_events = {}

function calendar_events:AfterUpdate()
  print("Calendar updated!")
end

-- Connects object and table
--
res = luacom_Connect(calendar, calendar_events)

if res == nil then
  exit(1)
end

-- This should trigger the AfterUpdate event
--
calendar:NextMonth()
```

It's also possible to separately create a LuaCOM object implementing the connection point source interface and then connect it to the object using `luacom_AddConnection`.

```
-- Creates the ActiveX object
--
calendar = luacom_CreateObject("MSCAL.Calendar")
```

```
if calendar == nil then
  print("Error instantiating calendar")
  exit(1)
end

-- Creates implementation table
--
calendar_events = {}

function calendar_events:AfterUpdate()
  print("Calendar updated!")
end

-- Creates LuaCOM object implemented by calendar_events
--
event_handler = luacom_ImplInterface(calendar_events,
  "MSCAL.Calendar",
   "DCalendarEvents")

if event_handler == nil then
  print("Error implementing DCalendarEvents")
  exit(1)
end

-- Connects both objects
--
luacom_addConnection(calendar, event_handler)

-- This should trigger the AfterUpdate event
--
calendar:NextMonth()

-- This disconnects the connection point established
--
luacom_releaseConnection(calendar)

-- This should NOT trigger the AfterUpdate event
--
calendar:NextMonth()
```

4.3.4 Parameter Passing

LuaCOM has some policies concerning parameter passing. They specify how LuaCOM will translate COM parameter lists to Lua and vice-versa. There are two differente situations to which these policies apply: calling a method of a COM object from Lua and calling a Lua function from COM. The main question here is how to deal with the different types of parameters supported by COM (“in” parameters, “out” parameters, “in-out” parameters, “optional” parameters and “defaultvalue” parameters).

Calling COM from Lua This situation happens when the accessing a property or calling a method of a COM object through the LuaCOM object. Here follows a sample:

```
word = luacom_GetObject("Word.Application")

-- Here we are calling the "Move" method of the Application object of
-- a running instance of Microsoft(R) Word(R)
word:Move(100,100)
```

In this situation, there are two steps in the parameter passing process:

1. convert Lua parameters to COM (this will be called the "lua2com" situation);
2. convert COM's return value *and* output values back to Lua (this will be called the "com2lua" situation).

lua2com situation The translation is done based on the type information of the method (or property); it's done following the order the parameters appear in the type information of the method. The Lua parameters are used in the same order. For each parameter there are three possibilities:

The parameter is an "in" parameter LuaCOM gets the first Lua parameter not yet converted and converts it to COM using LuaCOM type conversion engine.

The parameter is an "out" parameter LuaCOM ignores this parameter, as it will only be filled by the called method. That is, the "out" parameters SHOULD NOT appear in the Lua parameter list.

The parameter is an "in-out" parameter LuaCOM does the same as for "in" parameters.

When the caller of the method wants to omit a parameter, it must pass the `nil` value; LuaCOM then proceeds accordingly, informing the called method about the omission of the parameter. If the parameter has a default value, it is used instead. Notice that LuaCOM does not complain when one omits non-optional parameters. In fact, LuaCOM ignores the fact that a parameter is or isn't optional. It leaves the responsibility for checking this to the implementation of the called method.

com2lua situation When the called method finishes, LuaCOM translates the return value and the output values (that is, the values of the "out" parameters) to Lua return values. That is, the method return value is return to the Lua code as the first return value; the output values are returned in the order they appear in the parameter list (notice that here we use the Lua feature of multiple return values). If the method does not have return values, that is, is a "`void`" method, the return values will the output values. If there are no output values either, then there will be no return values.

The called method can omit the return value or the output values; LuaCOM them will return `nil` for each omitted value.

To illustrate these concepts, here follows a sample of these situations. First, we show an excerpt of an `ODL` file describing some methods of a COM object:

```
  HRESULT TestShort(
    [in] short p1, // an "in" parameter
    [out] short* p2, // an "out" parameter
    [in,out] short* p3, // an "in-out" parameter
    [out,retval] short* retval); // the return value
```

Now follows a sample of what happens when calling the method:

```
-- assume that "com" is a \luacom\ object

-- Here we set p1 = 1, p3 = 2 and leave p2 uninitialized
-- When the method returns, r1 = retval and r2 = p2 and r3 = p3
r1, r2, r3 = com:TestShort(1,2)

-- WRONG! The are only two in/in-out parameters! Out parameters
-- are ignored in the lua2com parameter translation
r1, r2, r3 = com:TestShort(1,2,3) -- WRONG!

-- Here p1 = 1, p2 is uninitialized and p3 is omitted.
r1, r2, r3 = com:TestShort(1)

-- Here we ignore the output value p3
r1,r2 = com:TestShort(1)

-- Here we ignore all output values (including the return value)
com:TestShort(1,2)
```

Calling Lua from COM This situation happens when one implements a COM dispinterface in Lua. The ActiveX binding has to translate the COM method calls to Lua function calls. The policy here concerning parameter list translation is the same as the one above, just exchanging "Lua" for "COM" and vice-versa. That is, all "in" an "in-out" COM parameters are translated to parameters to the Lua function call (the output parameters are ignored). When the call finishes, the first return value is translated as the return value of the COM method and the other return values are translated as the "in-out" and "out" values, following the order they appear in the method's type information. Continuing the previous example, here we show the implementation of a method callable from COM:

```
implementation = {}

-- This method receives TWO in/in-out parameters
function implementation:TestShort(p1, p2)
  -- the first one is the retval, the second the first out param
  -- the third the second out param (in fact, an in-out param)
  return p1+p2, p1-p2, p1*p2
end

-- Implements an interface
obj = luacom_ImplInterface(implementation, "TEST.Test", ITest)

-- calls the function implementation:TestShort via COM
```

```
r1, r2, r3 = obj:TestShort(1,2)
```

4.3.5 Exception Handling

COM exceptions are converted to `lua_error`'s containing the data of the exception.

4.4 Type Conversion

LuaCOM is responsible for converting values from COM to Lua and vice versa. This type conversion is done following some rules. This rules must be known to avoid misinterpretation of the conversion results and to avoid errors.

4.4.1 Numeric types

All COM numeric types are converted to Lua *number* type. Lua numbers are converted to the `double` type.

4.4.2 Strings

Lua strings are converted to BSTR (Basic Strings) and vice versa. A Lua string containing a number may be converted to a COM numeric type if the interface of the component receiving that value requires numeric data.

4.4.3 Boolean values

Lua uses the `nil` value as false and non-`nil` values as true. As LuaCOM gives a special meaning for `nil` values in the parameters, it can't use Lua convention for true and false values; instead, LuaCOM uses the C convention: the true value is a number different from zero and the false value is a zero value. Here follows a sample:

```
-- This function alters the state of the of the window.
-- state is a Lua boolean value
-- window is a LuaCOM object

function showWindow(window, state)

  if state then
    window.Visible = 1

    -- this has the same result
    windows.Visible = -10
  else
    window.Visible = 0
  end

end

-- Shows window
```

```
showWindow(window, 1)

-- Hides window
showWindow(window, nil)
```

4.4.4 Pointers to `IDispatch` and LuaCOM objects

A pointer to `IDispatch` is converted to a LuaCOM's object whose implementation is provided by the received pointer. A LuaCOM's object is converted to COM simply passing its interface implementation to COM.

4.4.5 Pointers to `IUnknown`

LuaCOM just allows passing and receiving `IUnknown` pointers; it does not operate on them. They are converted from/to userdatas with a specific tag.

4.4.6 Arrays and Tables

LuaCOM converts Lua tables to SAFEARRAY's and vice-versa. To be converted, Lua tables must be "array-line", that is, all of its elements must be or "scalars" or tables of the same lenght. These tables must also be "array-like". Here are some samples of how is this conversion done:

Lua table	Safe Array
`table = {"name", "phone"}`	$\begin{bmatrix} ''name'' & ''phone'' \end{bmatrix}$
`table = {{1,2},{4,9}}`	$\begin{bmatrix} 1 & 2 \\ 4 & 9 \end{bmatrix}$

4.4.7 `CURRENCY` type

The `CURRENCY` values are converted to Lua as numbers. When converting a value to COM where a `CURRENCY` is expected, LuaCOM accepts both numbers and strings formatted using the current locale for currency values. Notice that this is highly dependent on the configuration and LuaCOM just uses the VARIANT conversion fuctions.

4.4.8 The `DATE` type

When converting from COM to Lua, the `DATE` values are transformed in strings formatted according to the current locale. The converse is true: LuaCOM converts strings formatted according to the current locale to convert them to `DATE` values.

4.4.9 Error Handling

When LuaCOM cannot convert a value from or to COM it issues an exception, that may be translated to a `lua_error` or to a COM exception, depending on who is the one being called.

5 Release Information

Here is provided miscellaneous information specific to the current version of LuaCOM. Here are recorded the current limitations of LuaCOM, its known bugs, the history of modifications since the former version, technical details etc.

5.1 Limitations

Here are listed the current limitations of LuaCOM, as of the current version, and information about future relaxation of this restrictions.

- LuaCOM does not support named parameters; it might be implemented at request;
- LuaCOM doesn't support COM methods with variable number of parameters. This will be implemented in a future release;
- there isn't support for converting tables that are not "array-like". This may be relaxed in a future version, depending on the feasibility;
- LuaCOM only allows one connection point for each ActiveX object. This limitation may be relaxed in future versions;
- some functions of LuaCOM's Lua API are NOT protected against bad parameters. There may be Lua errors or application errors if they are called this way. This is being worked on and may be solved soon;
- it's not possible to create an instance of an ActiveX object whose initialization is done through a persistence interface (`IPersistStream`, `IPersistStorage` etc). Anyway, most of the ActiveX objects already tested initialize themselves through `CoCreateInstance`. Initialization via persistence interfaces is planned for a future release;
- LuaCOM doesn't provide access to COM interfaces that doesn't inherit the `IDispatch` interface. That is, only Automation Objects are supported. This restriction is due to the late-binding feature provided by LuaCOM. It's possible to provide access to these COM interfaces via a "proxy" Automation Object, which translate calls made through automation to vtable (early-binding) calls. It's also possible to implement this "proxy" directly using LuaCOM C/C++ API, but this hasn't been tested nor tried;
- LuaCOM doesn't handle exceptions very well yet. Currently, almost all exceptions call Lua function `lua_error`, possibly aborting the Lua code. A more careful exception handling mechanism is due to the next release.

5.2 Known bugs

Here are recorded the known bugs present in LuaCOM. If any other bugs are found, please report them through LuaCOM's home page.

Currently, there are no real "bugs" recorded, only limitations.

5.3 Future Enhancements

Besides the enhancements listed in the sections 5.1 and 5.2, there are other planned enhancements:

- better feedback when errors happen during the execution of LuaCOM API functions, besides returning `nil`;
- dynamic creation of type libraries;
- better support for creating full-fledged COM objects using Lua.

5.4 History

5.4.1 Version 0.9.2

- removal of `LUACOM_TRUE` and `LUACOM_FALSE` constants; now booleans follow the same convention of the C language;
- memory and interface leaks fixed;
- some functions of the API have slightly different names;
- changes in memory allocation policy, to follow more strictly practices recommended in COM documentation;
- parameter passing policies changed;
- added limited support for `IUnknown` pointers;
- changes in type conversion;
- added limited support for implementing and registering COM objects in Lua

5.4.2 Version 0.9.1

- conversion to Lua 4;
- better handling of different kinds of type information (e.g. now can access Microsoft Internet Explorer(R) object);
- now handles more gracefully exceptions and errors;
- added support for optional parameters with default values;
- LuaCOM does not initializes COM libraries anymore; this is left to the user;
- more stringent behaviour about the syntax of method calls and property access (methods with “:” and properties with “.”).

6 Reference

6.1 The C/C++ API

6.1.1 luacom_open

Prototype

```
void luacom_open(lua_State* L);
```

Description

This function initializes the LuaCOM library, registering Lua functions and Lua tags in the given Lua state L. Notice that it's necessary to initialize COM before, using `OleInitialize` or `CoInitialize` or something like that.

Sample

```
int main()
{
  lua_State *L = lua_open(0);

  OleInitialize(NULL);

  luacom_open(L);

  .
  .
  .
}
```

6.1.2 luacom_close

Prototype

```
  void luacom_close(lua_State* L);
```

Description

This function is intented to clean up the data structures associated with LuaCOM in a specific Lua state (L). Currently, it does nothing, but in future releases it will do. So, do not remove from your code! It must be also called before the COM termination functions (`OleUninitialize` and `CoInitialize`) and before `lua_close`.

Sample

```
int main()
{
  lua_State *L = lua_open(0);

  OleInitialize(NULL);

  luacom_open(L);

  .
  .
  .
  luacom_close(L);

  lua_close(L);

  OleUninitialize();
}
```

6.1.3 luacom_IDispatch2LuaCOM

Prototype

```
  int luacom_IDispatch2LuaCOM(lua_State *L, void *pdisp_arg);
```

Description

This functions takes a pointer to a `IDispatch`, creates a LuaCOM object for it and pushes it in the Lua stack. This function is useful when one gets an interface for a COM object from `C/C++` code and wants to use it in Lua.

Sample

```
void CreateAndExport(lua_State* L)
{
  // Creates the object
  IUnknown *obj = CreateObj();

  // Gets the IDispatch
  IDispatch* pdisp = NULL;
  QueryInterface(IID_IDISPATCH, &pdisp);

  // pushes onto lua stack
  luacom_IDispatch2LuaCOM(L, (void *) pdisp);
}
```

6.2 The Lua API

6.2.1 luacom_CreateObject

Use

```
luacom_obj = luacom_CreateObject(ProgID)
```

Description

This function finds the Class ID referenced by the ProgID parameter and creates an instance of the object with this Class ID. If there is any problem (ProgID not found, error instantiating object), the function returns nil.

Parameters

Parameter	Type
ProgID	String

Return Values

Return Item	Possible Values
luacom_obj	LuaCOM object nil

Sample

```
inet_obj = luacom_CreateObject("InetCtls.Inet")

if inet_obj == nil then
  print("Error! Object could not be created!")
end
```

6.2.2 luacom_Connect

Use

```
  implemented_obj = luacom_Connect(luacom_obj, implementation_table)
```

Description

This functions finds the default source interface of the object `luacom_obj`, creates an instance of this interface whose implementation is given by `implementation_table` and creates a connection point between the `luacom_obj` and the implemented source interface. Any calls made by the `luacom_obj` to the source interface implementation will be translated to lua calls to member function present in the `implementation_table`. If the function suceeds, the LuaCOM object implemented by `implementation_table` is returned; otherwise, `nil` is returned.

Parameters

Parameter	Type
`luacom_obj`	LuaCOM object
`implementation_table`	Table or userdata

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil

Sample

```
events_handler = {}
```

```
function events_handler:NewValue(new_value)
  print(new_value)
end

events_obj = luacom_connect(luacom_obj, events_handler)
```

6.2.3 luacom_ImplInterface

Use

```
  implemented_obj = luacom_ImplInterface(impl_table, ProgID, interface_name)
```

Description

This function finds the type library associated with the ProgID and tries to find the type information of an interface called “interface_name”. If it does, then creates an object whose implementation is “impl_table”, that is, any method call or property access on this object is translated to calls or access on the members of the table. Then it makes a LuaCOM object for the implemented interface and returns it. If there are any problems in the process (ProgID not found, interface not found, interface isn’t a dispinterface), the function returns nil.

Parameters

Parameter	Type
`impl_table`	table or userdata
`ProgID`	string
`interface_name`	string

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil

Sample

```
myobject = {}

function myobject:MyMethod()
  print("My method!")
end

myobject.Property = "teste"

luacom_obj = luacom_ImplInterface(myobject, "TEST.Test", "ITest")
```

```
-- these are done via Lua
myobject:MyMethod()
print(myobject.Property)

-- this call is done through COM
luacom_obj:MyMethod()
print(luacom_obj.Property)
```

6.2.4 luacom_ImplInterfaceFromTypelib

Use

```
impl_obj = luacom_ImplInterfaceFromTypelib(
  impl_table,
  typelib_path,
  interface_name,
  coclass_name)
```

Description

This function loads the type library whose filepath is "typelib_path" and tries to find the type information of an interface called "interface_name". If it does, then creates an object whose implementation is "impl_table", that is, any method call or property access on this object is translated to calls or access on the members of the table. Then it makes a LuaCOM object for the implemented interface and returns it. If there are any problems in the process (ProgID not found, interface not found, interface isn't a dispinterface), the function returns nil. The "coclass_name" parameter is optional; it is only needed if the resulting LuaCOM object is to passed to the function `luacom_Connect`, `luacom_AddConnection` or `luacom_ExposeObject`. This parameter specifies the Component Object class name to which the interface belongs, as one interface may be used in more than one "coclass".

Parameters

Parameter	Type
`impl_table`	table or userdata
`typelib_path`	string
`interface_name`	string
`coclass_name`	(optional) string

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil

Sample

```
myobject = {}
```

```
function myobject:MyMethod()
  print("My method!")
end

myobject.Property = "teste"

luacom_obj = luacom_ImplInterfaceFromTypelib(myobject, "test.tlb",
"ITest", "Test")

-- these are done via Lua
myobject:MyMethod()
print(myobject.Property)

-- this call is done through COM
luacom_obj:MyMethod()
print(luacom_obj.Property)
```

6.2.5 luacom_GetObject

Use

```
luacom_obj = luacom_GetObject(ProgID)
```

Description

This function finds the Class ID referenced by the ProgID parameter and tries to find a running instance of the object associated having this Class ID. If there is any problem (ProgID not found, object is not running), the function returns nil.

Parameters

Parameter	Type
ProgID	String

Return Values

Return Item	Possible Values
luacom_obj	LuaCOM object nil

Sample

```
excel = luacom_GetObject("Excel.Application")

if excel  == nil then
  print("Error! Could not get object!")
end
```

6.2.6 luacom_NewObject

Use

```
  implemented_obj = luacom_ImplInterface(impl_table, ProgID)
```

Description

This function is analogous to `luacom_ImplInterface`, doing just a step further: it locates the default interface for the ProgID and uses its type information. That is, this function creates a Lua implementation of a Component Object default interface. This is useful when implementing a complete COM object in Lua. If there are any problems in the process (ProgID not found, default interface is not a dispinterface), the function returns nil.

Parameters

Parameter	Type
`impl_table`	table or userdata
`ProgID`	string

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil

Sample

```
myobject = {}

function myobject:MyMethod()
  print("My method!")
end

myobject.Property = "teste"

obj = luacom_NewObject(myobject, "TEST.Test")

-- these are done via Lua
myobject:MyMethod()
print(myobject.Property)

-- this call is done through COM
luacom_obj:MyMethod()
print(luacom_obj.Property)
```

6.2.7 luacom_ExposeObject

Use

```
cookie = luacom_ExposeObject(luacom_obj)
```

Description

This function creates and registers a *class factory* for the `luacom_object`, so that other running applications can use it. It returns a cookie that must be used to unregister the object. If the function fails, it returns `nil`.

ATTENTION: the object MUST be unregistered (using `luacom_RevokeObject`) before calling `luacom_close` or `lua_close`, otherwise unhandled exceptions might occurr.

Parameters

Parameter	Type
luacom_obj	LuaCOM object

Return Values

Return Item	Possible Values
cookie	number nil

Sample

```
myobject = luacom_NewObject(impl_table, "Word.Application")

cookie = luacom_ExposeObject(myobject)

function end_of_application()
  luacom_RevokeObject(cookie)
end
```

6.2.8 luacom_addConnection

Use

```
result = luacom_addConnection(client, server)
```

Description

This function connects two LuaCOM objects, setting the `server` as a event sink for the `client`, that is, the client will call methods of the server to notify events (following the COM model). This will only work if the `client` supports connection points of the `server`'s type. If the function succeeds, it returns 1; otherwise, it returns `nil`.

Parameters

Parameter	Type
client	LuaCOM object
server	LuaCOM object

Return Values

Return Item	Possible Values
result	number nil

Sample

```
obj = luacom_CreateObject("TEST.Test")

event_sink = {}

function event_sink:KeyPress(keynumber)
  print(keynumber)
end

event_obj = luacom_ImplInterface(
              event_sink, "TEST.Test", "ITestEvents")

result = luacom_addConnection(obj, event_obj)

if result == nil then
  print("Error!")
  exit(1)
end
```

6.2.9 luacom_releaseConnection

Use

```
luacom_releaseConnection(client)
```

Description

This function disconnects a LuaCOM object from its event sink.

Parameters

Parameter	Type
client	LuaCOM object

Return Values

There are none.

Sample

```
obj = luacom_CreateObject("TEST.Test")

event_sink = {}

function event_sink:KeyPress(keynumber)
  print(keynumber)
end

event_obj = luacom_ImplInterface(
              event_sink, "TEST.Test", "ITestEvents")

result = luacom_addConnection(obj, event_obj)

if result == nil then
  print("Error!")
  exit(1)
end

.
.
.

luacom_releaseConnection(obj)
```

6.2.10 luacom_ProgIDfromCLSID

Use

```
progID = luacom_ProgIDfromCLSID(clsid)
```

Description

This function is a proxy for the Win32 function `ProgIDFromCLSID`.

Parameters

Parameter	Type
clsid	string

Return Values

Return Item	Possible Values
progID	string nil

Sample

```
progid = luacom_ProgIDfromCLSID("{8E27C92B-1264-101C-8A2F-040224009C02}")
obj = luacom_CreateObject(progid)
```

6.2.11 luacom_CLSIDfromProgID

Use

```
clsid = luacom_CLSIDfromProgID(progID)
```

Description

It's the inverse of `luacom_ProgIDfromCLSID`.

6.2.12 luacom_ShowHelp

Use

```
luacom_ShowHelp(luacom_obj)
```

Description

This function tries to locate the `luacom_obj`'s help file in its type information and shows it.

Parameters

Parameter	Type
luacom_obj	LuaCOM object

Return Values

None.

Sample

```
obj = luacom_CreateObject("TEST.Test")

luacom_ShowHelp(obj)
```

6.2.13 luacom_GetIUnknown

Use

```
iunknown = luacom_GetIUnknown(luacom_obj)
```

Description

This function returns a userdata holding the `IUnknown` interface pointer to the COM object behind `luacom_obj`. It's important to notice that Lua does not duplicates userdata: many calls to `luacom_GetIUnknown` for the same LuaCOM object will return the same userdata. This means that the reference count for the `IUnknown` interface will be incremented only once (that is, the first time the userdata is pushed) and will be decremented only when all the references to that userdata go out of scope (that is, when the userdata suffers garbage collection).

One possible use for this function is to check whether to LuaCOM objects reference the same COM object.

Parameters

Parameter	Type
luacom_obj	LuaCOM object

Return Values

Return Item	Possible Values
iunknown	userdata with IUnknown tag nil

Sample

```
-- Creates two LuaCOM objects for the same COM object
-- (a running instance of Microsoft(R) Word(R)

word1 = luacom_GetObject("Word.Application")
word2 = luacom_GetObject("Word.Application")

-- These two userdata should be the same
unk1 = luacom_GetIUnknown(word1)
unk2 = luacom_GetIUnknown(word2)

assert(unk1 == unk2)
```

6.2.14 luacom_isMember

Use

```
answer = luacom_isMember(luacom_obj, member_name)
```

Description

This function returns true (that is, different from `nil`) if there exists a method or a property of the `luacom_obj` named `member_name`.

Parameters

Parameter	Type
luacom_obj	LuaCOM object
member_name	string

Return Values

Return Item	Possible Values
answer	nil or non-nil

Sample

```
obj = luacom_CreateObject("MyObject.Test")

if luacom_isMember(obj, "Test") then
  result = obj:Test()
end
```

7 Credits

LuaCOM has been developed by Renato Cerqueira (rcerq@tecgraf.puc-rio.br) and Vinicius Almendra (almendra@tecgraf.puc-rio.br). The project has been sponsored by TeCGraf (Technology Group on Computer Graphics).