LuaCom User Manual

Version 1.0 (beta)

Vinicius Almendra Renato Cerqueira

25th June 2001

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;
- 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 the most important types;
- object disposal using Lua garbage collection mechanism;
- implementation of COM interfaces using Lua tables;
- use of COM connection point mechanism to handle bidirecional communication and event handling.

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 "luacom.h"

int main (int argc, char *argv[]) {
  /* library initialization */
  iolib_open();
  strlib_open();
  mathlib_open();

  if(!luacom_open()) {
    puts("Error initializing LuaCOM!");
    exit(1);
  }

  if(lua_dofile("activex_sample.lua") != 0) {
```

```
    puts("Error running sample.lua!");
    exit(1);
  }
  luacom_close();
  return 0;
}
```

LuaCom's initialization may fail, so it's necessary to check the return value of `luacom_open`.

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 access 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_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_GetObject	Creates a LuaCom object associated with an instance of an already running ActiveX object

C/C++ API

Function	Description
`luacom_open`	Initializes the LuaCom library. It must be called before any use of LuaCom features.
`luacom_close`	Frees any system resources used by LuaCom.
`luacom_idispatch2luacom`	Takes an IDispatch interface and creates a LuaCom object to expose it, pushing the object on the C2Lua 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 C++ object holds the interface pointer to the ActiveX 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.7")

-- 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 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 [1].

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. Currently, these dispinterfaces must be defined in a *type library*, which can be associated to an registered ActiveX object or just a stand-alone type library. 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 Calendar object, whose ProgID is MSCAL.Calendar and the
-- major version is 7

events_obj = luacom_ImplInterface(
  events_table,
  "MSCAL.Calendar.7",
  "DCalendarEvents",
  7)

-- 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_ImplInterfa` must be used instead:

[1] *dispinterfaces* are ActiveX interfaces that have (or are designed to have) an `IDispatch` implementation, which permits late-binding and its use by interpreted languages.

```
-- 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 the 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`).

4.3.1 Connection Points

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 point 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.7")

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.7")

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.7",
   "DCalendarEvents", 7)

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()
```

4.4 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. It's also necessary to verify which type conversions are supported (see section 5.1.1).

4.4.1 Numeric types

All COM numeric types are converted to Lua *number* type and vice versa.

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 a numeric element.

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, it defines two global variables, LCTRUE and LCFALSE, that must be used when passing boolean values to LuaCom or receiving them, either via parameters or via properties. 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 = LCTRUE
  else
    window.Visible = LCFALSE
  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 Arrays and Tables

To be converted to COM, a table must follow these restrictions

- must be an array;
- all of its non-table elements must be of the same type.

If the table follows these restrictions, it will be converted to a SAFEARRAY whose elements will be the non-table elements of the table.

The conversion from a SAFEARRAY to a table is analogous. 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}}`	$\begin{bmatrix} 1 & 2 \\ 4 & 9 \end{bmatrix}$

4.4.6 CURRENCY and DATE

These types are treated as common numbers. LuaCom doesn't interpret them: just blindly converts from or to `double`'s using the Windows API. See section 5.1.1 for the current status of LuaCom's support for these two types.

4.5 Parameter Passing

Method call for LuaCom objects are done the same way as calling Lua functions inside tables:

```
table = {}

function table:method(parameter)
  return retval
end

-- method call
a = table:method(2)
```

Nevertheless, there are some differences concerning optional parameters. In COM, you can omit parameters in method calls. To do so in LuaCom, just put a `nil` value as the value of the parameters to be omitted. Here follows a sample:

```
-- the Find method expects 4 parameters, of which the last 2 are optional

-- Call with all parameters
obj:Find("name", "John", LCTRUE, "index")

-- Call omitting the optional parameters
obj:Find("name", "John")

-- Call omitting the third parameter
obj:Find("name", "John", nil, "index")
```

It's important to notice that the `nil` value IS NOT converted to a COM boolean false (see section 4.4.3).

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 Technical Information

5.1.1 Supported COM Types

The following types are fully supported:

- numeric types;
- strings;
- booleans;
- `IDispatch` pointers;
- safe arrays of numeric and string types.

In section 4.4 there is a description of how LuaCom converts from COM types to Lua ones and vice versa. The `CURRENCY` and `DATE` types are supported but are blindly converted from and to `double`; no interpretation of their values is done. They should be used carefully as they haven't been tested yet.

5.2 Limitations

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

- SAFEARRAYs of `VARIANT`s aren't supported yet. This should be implemented in the next version;
- indexed properties are not supported yet. Their implementation is due to the next version;
- 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;
- 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 due to the next release;
- the automatic disposal of COM objects through Lua garbage collection mechanism may not release all COM objects. This should be improved on the next 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 allow the implementation of a full-fledged Automation Object, as it lacks a some functionalities needed for this (class factories, type library construction etc). This should be implemented on a future release;
- 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.

5.3 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.

- there are some memory leaks and interface leaks.

5.4 History

There is no history yet, as this is the first public release of LuaCom...

6 Reference

6.1 The Lua API

This section is still incomplete. Here are documented just the most important functions needed for dealing with COM objects.

6.1.1 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 and prints an error message in the terminal.

Parameters

Parameter	Type
ProgID	String

Return Values

Return Item	Possible Values
luacom_obj	LuaCom tag nil

Sample

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

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

6.1.2 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 tag
`implementation_table`	Table

Return Values

Return Item	Possible Values
implemented_obj	LuaCom tag nil

Sample

```
events_handler = {}

function events_handler(new_value)
  print(new_value)
```

```
end

events_obj = luacom_connect(luacom_obj, events_handler)
```

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).