LuaInterface: User's Guide

Fabio Mascarenhas[1]

[1]Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente, 225 – 22453-900
Rio de Janeiro, RJ, Brasil

mascarenhas@acm.org

1. Introduction

LuaInterface is library for integration between the Lua language and Microsoft .NET platform's Common Language Runtime (CLR) [2]. Several languagens already have compilers targeting the CLR, and CLR implementations already exist for Microsoft Windows, BSD and Linux operating systems [3, 4].

Lua is a programming language designed for extending applications, with its interpreter implemented as an easily embeddable library. Describing Lua's syntax and resources is outside the scope of this manual, but it can be found inside Lua's reference manual [1].

The following section explains how to compile and install LuaInterface. Section 3 covers its use by CLR application, and Section 4 its use by Lua scripts.

2. Installing LuaInterface

LuaInterface needs the Lua interpreter to work. An interpreter for Lua 5.0 is already included in the LuaInterface distribution, along with the LuaInterface binaries for Microsoft Windows and the .NET CLR (`LuaInterface.dll` and `luanet.dll`). You should copy `lua50.exe` and `lua50.dll` to somewhere in your PATH, then copy `LuaInterface.dll` to your Global Assembly Cache. LuaInterface uses Compat-5.1, so copy `luanet.dll` to somewhere in your `package.cpath`.

Compiling LuaInterface from sources is not difficult. The distribution includes a project file for compiling `luanet.dll` under Visual Studio .Net 2003, and a compilation script for compiling it under Visual Studio 6. You can compile `LuaInterface.dll` by simply compiling all C# files at `src/LuaInterface`.

3. Lua from the CLR

CLR applications access the Lua interpreted trough the `LuaInterface.Lua` class. Instantiating an object of this class creates a new Lua interpreter, and distinct instances are fully independent.

`Lua` class' indexer creates, reads and modifies global variables, indexed by variable name, as in the following example (the indexer always returns an `object` that must be cast to the correct type):

```
// Start a Lua interpreter
Lua lua = new Lua();
// Create global variables "num" and "str"
lua["num"] = 2;
lua["str"] = "a string";
// Create an empty table
lua.NewTable("tab");
// Read global variables "num" and "str"
double num = (double)lua["num"];
string str = (string)lua["str"];
```

The `DoString` and `DoFile` methods execute Lua scripts. The methods return an array with the script's return valus. Example:

```
// Execute a Lua script file
lua.DoFile("script.lua");
// Execute chunks of Lua code
lua.DoString("num=2");
lua.DoString("str='a string'");
// Lua code returning values
object[] retVals = lua.DoString("return num,str");
```

LuaInterface automatically converts Lua's `nil` to CLR's `null`, strings to `System.String`, numbers to `System.Double`, booleans to `System.Boolean`, tables to `LuaInterface.LuaTable`, functions to `LuaInterface.LuaTable`, and vice-versa. Userdata are a special case: CLR objects with no matching Lua type are converted to userdata, and these userdata converted back to the original object when passed to the CLR. LuaInterface converts other userdata to `LuaInterface.LuaUserData`.

`LuaTable` and `LuaUserData` objects also have an index to read and modify fields, indexed either by strings or by numbers. `LuaFunction` and `LuaUserData` objects have a `Call` method to execute the function, with any number of arguments, and return the function's return values inside an array.

Finally, class `Lua` also has the `RegisterFunction` method to register CLR methods as global Lua functions. Its parameters are the name of the variable where the function will be stored, the target of the method and the `MethodInfo` object representing the method, for example, `lua.RegisterFunction("foo",obj,obj.GetType().GetMethod("Foo"))` registers method `Foo` of object `obj` as function `foo`.

4. CLR from Lua

This section covers instantiation and use of CLR objects from Lua scripts, either run by the Lua interpreter or executed inside a CLR application. All the examples below are in the Lua language.

4.1. Loading CLR types and instating objects

To instantiate an object a script needs a type reference. They also need type references to access static fields and to call static methods. To get a type reference the script first should load an assembly containing the type with the function `load_assembly`. Then it uses the `import_type` function to get the reference. The following example shows how the two functions should be used:

```
require"luanet"
  -- Loads the System.Windows.Forms and System.Drawing assemblies
  luanet.load_assembly("System.Windows.Forms")
  luanet.load_assembly("System.Drawing")
  Form = luanet.import_type("System.Windows.Forms.Form")
  Point = luanet.import_type("System.Drawing.Point") -- structure
  -- Loading an enumeration
  StartPosition = luanet.import_type("System.Windows.Forms.FormStartPosition")
```

Calling a type reference instantiates an object of the reference's type. LuaInterface uses the first contructor that matches the number and type of arguments passed, in case there are overloaded constructors. The matching process also converts numerical strings to numbers and numbers to strings, if necessary. Lua numbers uses as arguments to a numeric parameter are also converted to respective CLR numeric type.

The `get_constructor_bysig` returns a constructor given a type and the constructor's signature (type references to the constructor parameter's types). Calling the returned constructor instantiates an object. The following example shows the different ways to instantiate a CLR object:

```
  -- SomeType is a reference to a type with the following constructors
  -- 1. SomeType(string)
  -- 2. SomeType(int)
  -- 3. SomeType(int,int)
  obj1 = SomeType(2,3) -- instantiates SomeType with constructor 3
  obj2 = SomeType("x") -- instantiates SomeType with constructor 1
  obj3 = SomeType(3) -- instantiates SomeType with constructor 1
  Int32 = import_type("System.Int32")
  -- Gets the SomeType constructor with signature (Int32)
  SomeType_cons2 = get_constructor_bysig(SomeType,Int32)
  obj3 = SomeType_cons2(3) -- instantiates SomeType with constructor 2
```

4.2. Accessing fields and methods

Scripts can access the fields of a CLR object with the same syntax to index tables. The values written to a field are converted to the field's type, for example, a number written to an `Int32` field is converted to `Int32`. Non-indexed properties are accessed like fields.

LuaInterface has a shortcut for indexing single-dimension arrays, indexing the array's reference with a number, for example, `arr[3]`. For multidimensional arrays the methods of class `Array` should be used instead.

Scripts can call an object's methods with the same syntax used to call table's methods, passing the target as the first argument ou using the ":" operator. They can also access indexed properties through the properties' `get` and `set` methods (usually `get_PropertyName` and `set_PropertyName`).

```
-- button1, button2 and form1 are CLR objects
button1.Text = "OK";
button2.Text = "Cancel";
form1.Controls:Add(button1);
form1.Controls:Add(button2);
form1:ShowDialog();
```

Lua only has call-by-value parameters, so when a script calls a method with *out* or *ref* parameters LuaInterface returns the output values of these parameters together with the method's return value. *out* parameters should be omitted from the method call, as in the following example:

```
-- calling int obj::OutMethod1(int,out int,out int)
retVal,out1,out2 = obj:OutMethod1(inVal)
-- calling void obj::OutMethod2(int,out int)
retVal,out1 = obj:OutMethod2(inVal) -- retVal será nil
-- calling int obj::RefMethod(int,ref int)
retVal,ref1 = obj:RefMethod(inVal,ref1)
```

If a method is overloaded the version that gets called is the first that matches the number and type of the call's arguments, ignoring *out* parameters. The following example shows how a script should call different versions of an overloaded `SomeMethod` method of `SomeType`:

```
-- Versions of SomeType.SomeMethod:
-- 1. void SomeMethod(int)
-- 2. void SomeMethod(string)
-- 3. void SomeMethod(OtherType)
-- 4. void SomeMethod(string,OtherType)
-- 5. void SomeMethod(int,OtherType)
-- 6. void SomeMethod(int,OtherTypeSubtype)
-- obj1 is instance of SomeType
-- obj2 is instance of OtherType
-- obj3 is instance of OtherTypeSubtype
obj1:SomeMethod(2) -- version 1
obj1:SomeMethod(2.5) -- version 1, round down
obj1:SomeMethod("2") -- version 1, converts to int
obj1:SomeMethod("x") -- version 2
obj1:SomeMethod(obj2) -- version 3
obj1:SomeMethod("x",obj2) -- version 4
obj1:SomeMethod(2,obj2) -- version 4
obj1:SomeMethod(2.5,obj2) -- version 4, round down
obj1:SomeMethod(2,obj3) -- version 4, cast
-- versions 5 and 6 never get called
```

There is the function `get_method_bysig` in case a method has versions that can never be called. Given an object or a type reference and a method signature (name and types of the methods) the function returns the method with that signature, as in the following example:

```
-- Versions of SomeType.SomeMethod:
-- 5. void SomeMethod(int,OtherType)
-- obj1 is instance of  SomeType
-- obj2 is instance of OtherType
Int32 = luanet.import_type('System.Int32')
```

```
SomeMethod_sig5 = luanet.get_method_bysig(obj1,'SomeMethod',
  Int32,obj2:GetType())
SomeMethod_sig5(obj1,2,obj2) -- calls version 5
```

If a method or field's name is one of Lua's reserved words the script can still access them using the `obj["name"]` notation. If an object has two methods with the same signature but belonging to distinct interfaces, like `IFoo.method()` and `IBar.method()`, the notation `obj["IFoo.method"](obj)` calls the first version.

LuaInterface returns exceptions raised during the execution of a method as errors, with the exception object as the error message. If a script wishes to capture the error it should call the method using `pcall`.

4.3. Handling events

Events in LuaInterface have an `Add` and a `Remove` method, respectively used to register and unregister event handlers. `Add` takes a Lua function as argument, converting it to a CLR delegate appropppriate to the event and returning the delegate. `Remove` takes an event handler delegate as an argument and removes the handler, as in the following example:

```
function handle_mouseup(sender,args)
  print(sender:ToString() .. ' MouseUp!')
  button.MouseUp:Remove(handler)
end
handler = button.MouseUp:Add(handle_mouseup)
```

Scripts can also register event handlers with the `add` and `remove` methods of the event's object (usually `add_EventName` and `remove_EventName`), but `add` does not return the delegate created, so if a function is registered this way it can not be unregistered later.

4.4. Delegates and subtyping

LuaInterface provides three ways to extend the CLR with new dynamically created types. The first was already commented on in the context of event handlers: passing a Lua function where a delegate is expected, LuaInterface creates a new delegate type and passes it to the CLR.

The second way is by passing a Lua table where an interface implementation is expected. LuaInterface creates a new type implementing the interface and na object of that type that delegates its methods to the table, as in the following example:

```
-- interface ISample { int DoTask(int x, int y); }
-- SomeType.SomeMethod signature: int SomeMethod(ISample i)
-- obj is instance of SomeType
sum = {}
function sum:DoTask(x,y)
  return x+y
end
-- sum is converted to instance of ISample
res = obj:SomeMethod(sum)
```

If there are overloaded methods in the interface all versions of the method will be delegated to the same Lua function, and it is up to the function to decide which version was called from the number and type of the arguments. LuaInterface does not pass *out* parameters to the function, but the function should return their values and the output values of *ref* parameters together with the method's return value, like the following example:

```
-- interface ISample2 { void DoTask1(ref int x, out int y);
--                                int DoTask2(int x, out int y); }
-- SomeType.SomeMethod signature: int SomeMethod(ISample i)
-- obj is instance of SomeType
inc = {}
function inc:DoTask1(x)
  return x+1,x
end
function inc:DoTask2(x)
  return x+1,x
end
res = obj:SomeMethod(sum)
```

The last way to create new CLR types is through subclassing of an existing class, overrideing some or all of its virtual methods with functions of a Lua table (if the Lua table does not override a method LuaInterface uses the original version). The table functions can call the superclass' methods through a field called `base`.

To turn a table into an instance of a subclass the script should call the `make_object` function, with the table and a type reference to the class that will be subclassed as arguments. The following example shows how to create subclasses:

```
-- class SomeObject {
--   public virtual int SomeMethod(int x, int y) { return x+y; } }
-- SomeType.SomeMethod signature: int SomeMethod(SomeObject o)
-- obj is instance of SomeType
some_obj = { const = 4 }
function some_obj:SomeMethod(x,y)
  local z = self.base:SomeMethod(x,y)
  return z*self.const
end
SomeObject = luanet.import_type('SomeObject')
luanet.make_object(some_obj,SomeObject)
res = some_obj:SomeMethod(2,3) -- returns 20
res = some_obj:ToString() -- calls base method
res = obj:SomeMethod(some_obj) -- passing as argument
```

The same issues about overloading and out/ref parameters present when implementing an interface apply to subclassing. Finally, the `free_object` function takes a table that was previously argument to a `make_object` call and severs the table's connection with the CLR subclass instance. Scripts must do this before discarding the last reference to the table, or memory leaks will occur.

References

[1] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua 5.0 Reference Manual. Technical Report 14/03, PUC-Rio, 2003. Available at `http://www.lua.org`.

[2] E. Meijer and J. Gough. Technical Overview of the Common Language Runtime. Technical report, Microsoft Research, 2002. Available at `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`.

[3] D. Stutz. The Microsoft Shared Source CLI Implementation, 2002. Available at `http://msdn.microsoft.com/library/en-us/Dndotnet/html/mssharsourcecli.asp`.

[4] Ximian. The Mono Project, 2003. Available at `http://www.go-mono.com/`.