Journal of Universal Computer Science, vol. 10, no. 7 (2004), 892-909
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04

LuaInterface: Scripting the .NET CLR with Lua

Fabio Mascarenhas
(Pontifical Catholic University of Rio de Janeiro, Brazil
mascarenhas@acm.org)

Roberto Ierusalimschy
(Pontifical Catholic University of Rio de Janeiro, Brazil
roberto@inf.puc-rio.br)

Abstract: We present LuaInterface in this paper, a library for scripting the .NET CLR with Lua. The .NET Common Language Runtime (CLR) aims to provide interoperability among objects written in several different languages.

LuaInterface gives Lua the capabilities of a full CLS consumer. The Common Language Specification (CLS) is a subset of the CLR specification, with rules for language interoperability, and CLS consumers are languages that can use CLS-compliant libraries. LuaInterface lets Lua scripts instantiate and use CLR objects, and even create new CLR types. CLR applications may also use LuaInterface to embed a Lua interpreter, using Lua scripts to extend the application.

LuaInterface is implemented as a bridge between the Lua interpreter and the CLR. The implementation required no changes to the interpreter, and the level of integration is the same that a Lua compiler would have.

Key Words: scripting, language interoperability, reflection
Category: D.3.3

1 Introduction

The Microsoft .NET Framework aims to provide interoperability among several different languages, through its Common Language Runtime (CLR) [Meijer and Gough, 2002]. The CLR specification is an ISO and ECMA standard [Microsoft, 2002], and implementations for non-Windows platforms already exist [Stutz, 2002, Ximian, 2003]. Several languages have compilers for the CLR, and compilers for many other languages are under development [Bock, 2003].

Lua [Ierusalimschy, 2003, Ierusalimschy et al., 1996] is a scripting language widely used in the game industry. Lua is easy to embed, small, fast, and flexible. It has a simple syntax, is interpreted, dynamically typed, and has several reflexive facilities. Lua also has first-class functions, lexical scoping, and coroutines.

Scripting languages are often used for connecting components written in other languages (“glue” code). They are also used for building prototypes, and as languages for configuration files. The dynamic nature of these languages allows the use of components without previous type declarations and without the need for a compilation phase. Nevertheless, they perform extensive type checking at runtime and provide detailed information in case of errors. The combination of these features can increase developer productivity by a factor of two or more [Ousterhout, 1998].

This paper presents LuaInterface, a library for the CLR that allows Lua scripts to access the object model of the CLR, the Common Type System (CTS). With LuaInterface, Lua becomes a scripting language for components written in any language that runs in the CLR. LuaInterface provides all the capabilities of a full CLS consumer. The Common Language Specification (CLS) is a subset of the CLR specification that establishes a set of rules for language interoperability [Microsoft, 2002, CLI Partition I Section 7.2.2]. Compilers that generate code capable of using CLS-compliant libraries are called *CLS consumers*. Compilers that can produce new libraries or extend existing ones are called *CLS extenders*.

A full CLS consumer must be able to call any CLS-compliant method or delegate, even methods with names that are keywords of the language; to call distinct methods of a type with the same signature but from different interfaces; to instantiate any CLS-compliant type, including nested types; to read and write any CLS-compliant property; and access any CLS-compliant event.

With LuaInterface, Lua scripts can instantiate CTS types, access their fields, and call their methods, all using the standard Lua syntax. CLR applications can also use LuaInterface, to run Lua code, access Lua data, call Lua functions, and register CLR methods as functions. Applications can also use Lua as a language for their configuration scripts, or as an embedded scripting language, and Lua scripts can glue together different components. Besides these consumer facilities, LuaInterface also provides support for creating new CTS types, with restrictions.

Lua is dynamically typed, so it needs no type declarations to instantiate or use CLR objects. It checks at runtime the correctness of each operation. LuaInterface makes extensive use of the reflexive features of the CLR, and does not need any preprocessing or creation of stubs for each object that a script uses. Its implementation required no changes to the Lua interpreter: the interpreter is compiled to a dynamically linked library that the CLR interfaces with, using P/Invoke (the native code interface of the CLR).

Porting scripting languages to the CLR has been hard. ActiveState has tried to build Perl and Python compilers, but abandoned both projects [ActiveState, 2000, Hammond, 2000]. Smallscript Inc. has been working on a Smalltalk compiler for the CLR since 1999, and “preparation for release is in progress” since early 2003 [Inc., 2000]. LuaInterface, on the other hand, is just a bridge between the Lua interpreter and the CLR, so was much simpler to implement than a compiler. Nevertheless, LuaInterface has full CLS consumer capabilities; the integration is seamless, and we achieved the same level of integration that a Lua compiler would have.

The rest of this paper is structured as follows: Section 2 shows how applications can use LuaInterface and the methods it exposes, with examples. Section 3 describes particular issues of the implementation. Section 4 presents some related work and compares them with LuaInterface, and Section 5 presents some conclusions and future developments.

2 Interfacing Lua and the CLR

Lua is an embeddable language, so it has an API that allows an application to instantiate and control a Lua interpreter. LuaInterface wraps this API into a class named `Lua`, which provides methods to execute Lua code, to read and write global variables, and to register CLR methods as Lua functions. Auxiliary classes provide methods to access fields of Lua tables and to call Lua functions. Instantiating an object of class `Lua` starts a new Lua interpreter. Multiple instances may be created, and are completely independent. Methods `DoFile` and `DoString` of class `Lua` execute a Lua source file and a string of Lua code, respectively. Scripts can read or write global variables by indexing an instance of class `Lua` with the variable name. The following C# code shows how to use instances of class `Lua`:

```
// Start a new Lua interpreter
Lua lua = new Lua();
// Run Lua chunks
lua.DoString("num = 2"); // create global variable 'num'
lua.DoString("str = 'a string'");
// Read global variables 'num' and 'str'
double num = (double)lua["num"];
string str = (string)lua["str"];
// Write to global variable 'str'
lua["str"] = "another string";
```

Functions are first-class values in Lua; Lua objects are just tables, with functions stored in fields as their methods. By convention, these functions receive a first argument called `self` that holds a reference to the table. There is syntactic sugar for accessing fields and methods. The `obj.field="foo"` statement is the same as `obj["field"]="foo"`, for example. An expression as `obj:foo(arg1,arg2)` is a method call, and syntactic sugar for `obj["foo"](obj,arg1,arg2)`, that is, the receiver goes as the first argument to the call.

2.1 Type conversions

Lua has seven types: `nil`, *number*, *string*, *boolean*, *table*, *function* and *userdata*. The `nil` type has only one value, `nil`, and represents an uninitialized (or invalid) reference; values of the *number* type are double-precision floating point numbers; values of the *string* type are character strings; the *boolean* type has two values, `true` and `false`; values of the *table* type are associative arrays; functions belong to the *function* type; and the *userdata* type is for arbitrary data from the embedding application.

The CLR is statically typed, so whenever a Lua value is passed to the CLR, LuaInterface converts the Lua value to the type the CLR expects, if possible; otherwise LuaInterface throws an exception. If the CLR is expecting a value of type `object`, LuaInterface uses the default mapping: `nil` to `null`, numbers to `System.Double`, strings to `System.String`, and booleans to `System.Boolean`. LuaInterface converts Lua tables to instances of `LuaTable`. Indexing an instance of this class accesses a field of the table with the corresponding key. Functions are converted to instances of `LuaFunction`. This class defines a `call` method that calls the corresponding Lua function and returns an array with the return values of the function.

If the CLR expects a numeric type, LuaInterface converts numbers to the expected type, rounding the number, if necessary. LuaInterface also converts numerical strings to CLR numbers. Likewise, LuaInterface converts numbers to strings, if the CLR is expecting a string. If the CLR expects a boolean value, LuaInterface converts any Lua value, except `false` and `nil`, to `true`. If the CLR expects a *delegate*, and the Lua value is a function, LuaInterface also converts the function to a delegate of the expected type. LuaInterface creates a delegate that executes the Lua function, with the arguments to the delegate becoming arguments to the function.

Whenever a value is passed from the CLR to Lua, LuaInterface converts the CLR value to the closest Lua type: `null` to `nil`, numeric values to numbers, `System.String` instances to strings, `System.Boolean` instances to booleans. LuaInterface converts instances `LuaTable` and `LuaFunction` to the corresponding tables and functions, respectively. LuaInterface converts all other CLR values to proxies to the actual value. Section 2.4 covers this.

2.2 Methods, constructors and overloading

The CLR supports method overloading. Whenever a Lua script calls an overloaded method, LuaInterface must select which method to actually call. LuaInterface goes through each of the methods, first checking whether the number of parameters match the number of arguments to the call. If the numbers match, LuaInterface checks whether each argument can be converted to the type of its corresponding parameter, according to the rules outlined in Section 2.1. LuaInterface throws an error if it does not find a suitable method to call. Constructors can also be overloaded (and usually are). LuaInterface uses this same procedure to select which constructor to use, when instantiating an object.

LuaInterface uses the first method that matches, and does not try to check if other methods could be a better match. This is simpler, faster, and unambiguous. Using the first match has consequences, though: there can be overloaded methods that cannot be directly called. If the first method takes a `Double` parameter, and the second takes a `Int32` parameter, the second method is never selected; any value that can be converted to `Int32` can also be converted to `Double`. Section 2.6 presents a workaround for this issue.

2.3 Loading CTS types and instantiating objects

Lua scripts need a *type reference* to instantiate new CLR objects. They need two functions to get a type reference: first they must use `load_assembly`, which loads an assembly, making its types available for importing as type references; then they must use `import_type`, which searches all loaded assemblies for a type and returns a reference to it. The following excerpt illustrates how these functions work:

```
load_assembly("System.Windows.Forms")
load_assembly("System.Drawing")
Form = import_type("System.Windows.Forms.Form")
Button = import_type("System.Windows.Forms.Button")
Point = import_type("System.Drawing.Point")
```

Scripts can use `import_type` to get type references for structures (e.g. `Point`) and enumerations, as well as classes. To instantiate a new CLR object, a script calls the respective type reference as a function. The following example extends the previous one to show how objects are instantiated:

```
form1 = Form()
button1 = Button()
button2 = Button()
```

Expressions like `Form()` and `Button()`, in the example above, are regular Lua syntax; Lua code can call any value. Lua has extensibility features that allow libraries to control how these operations are carried; Section 3 explains how LuaInterface uses these features.

A script can also use a type reference to call static methods from the type. The syntax is the same syntax used to call methods of a Lua object; for example, the expression `Form:GetAutoScaleSize(arg)` calls method `GetAutoScaleSize` from type `Form`. LuaInterface dynamically searches the type for the desired static method. Scripts can also use a type reference to read and write the static fields from the type. For example, `var=Form.ActiveForm` assigns to variable `var` the value of the `ActiveForm` property from type `Form`.

2.4 Accessing other CTS types

Some CTS types have a direct mapping to Lua. These types are `null`, numeric types, `Boolean`, `String`, and the types `LuaTable` and `LuaFunction` presented on Section 2.1. LuaInterface maps values of all the other CTS types to proxies to those values. In the object instantiation example in Section 2.3, for instance, the values assigned to the `form1`, `form2`, `button1` and `button2` variables are all proxies to the actual CLR objects. Scripts can use these proxies as they use any other Lua object: they can read fields, assign to fields, read properties, assign to properties, and call methods. The following example continues the two previous ones, showing how to use properties and methods:

```
button1.Text = "Ok"
button2.Text = "Cancel"
button1.Location = Point(10,10)
button2.Location = Point(button1.Left,button1.Height+
  button1.Top+10)
form1.Controls:Add(button1)
form1.Controls:Add(button2)
form1:ShowDialog()
```

In the previous example, the `button1.Text="Ok"` statement assigns the string `"Ok"` to the `Text` property on object `button1`. The `form1.Controls:Add(button1)` statement reads property `Controls` on object `form1`, then calls method `Add` on the value of this property, passing object `button1` as the argument to the method call. The three previous examples combined, when run, show a form with two buttons.

LuaInterface passes to Lua, as an error, any exception that occurs during execution of a CLR method. The exception object is the error message (Lua “error messages” are not restricted to strings). Then the script can use Lua mechanisms to capture and treat those errors.

2.5 Event handling

Events are a CLR facility to implement callbacks. To support an event, a type declares two methods: one method to add a handler to the event and another to remove a handler. The metadata for the type declares the event name, the type of the event handlers, and the methods to add and remove handlers. An application can use the reflection API of the CLR to discover what events a type declares, and to add or remove event handlers; the reflection API gets this information from the metadata of the type.

LuaInterface represents events as objects that define two methods: `Add` and `Remove`. These methods respectively add and remove a handler for the event.

Method `Add` receives a Lua function, converts the function to a delegate of the type the event expects, and adds the delegate as a handler to that event. Method `Remove`, in turn, receives a delegate registered as an event handler and removes it.

For example, if an object `obj` defines an event `Ev`, the expression `obj.Ev` returns an object that represents the event `Ev`. If `func` is a function, the `obj.Ev:Add(func)` statement registers `func` as a handler to event `Ev`. Each time event `Ev` fires, the CLR calls the delegate, which in turn calls function `func`. The following Lua code extends the previous examples to add event handlers to both buttons:

```
function handle_mouseup(sender,args)
  print(sender:ToString() .. " MouseUp!")
end
button1.MouseUp:Add(handle_mouseup)
button2.Click:Add(os.exit)
```

In the previous example, the `button1.MouseUp:Add(handle_mouseup)` statement registers the `handle_mouseup` function as a handler to event `MouseUp` on object `button1`. This function prints a message on the console. The CLR provides the `sender` and `args` parameters; they are, respectively, the object that fired the event and data specific to the event. The `button2.Click:Add(os.exit)` statement registers the `os.exit` function, from the Lua standard library, as a handler to event `Click` on object `button2`. This function ends the script. It has no parameters, but this is not a problem: the Lua interpreter will discard the two arguments passed to the function.

2.6 Additional full CLS consumer capabilities

The features already presented cover most uses of LuaInterface, and most of the capabilities of a full CLS consumer. The following paragraphs present the features that cover the rest of the capabilities that a full CLS consumer must provide.

The CLR offers both call-by-value and call-by-reference parameters. Call-by-reference parameters come in two types: *out* parameters can only be assigned to, and *ref* parameters can be both read from and assigned to. Lua offers only call-by-value parameters, so LuaInterface supports *out* and *ref* parameters using multiple return values (functions in Lua can return any number of values). LuaInterface returns the values of *out* and *ref* parameters after the return value of the method, in the order they appear in its signature.

The standard method selection of LuaInterface uses the first method that matches the number and type of the arguments to the call, so some methods of an object may never be selected (as discussed in Section 2.2). To call those methods, LuaInterface provides the `get_method_bysig` function. It takes an object, the method name, and a list of type references. Calling `get_method_bysig` returns a function that, when called, executes the method that matches the provided signature. The first argument to the call must be the receiver of the method. Scripts can also use `get_method_bysig` to call instance methods of the CLR numeric and string types, and to call static methods. Constructors can be overloaded, too, so there is also the `get_constructor_bysig` function.

To call a method named after a Lua keyword, a script can use the fact that the `obj:method(...)` and `obj["method"](obj,...)` forms are equivalent in Lua. For example, the `obj:function(...)` statement invalid in Lua, as `function` is a Lua keyword. The script should use the equivalent statement `obj["function"](obj,...)`.

To call distinct methods with the same signature, but belonging to different interfaces, scripts can prefix the method name with the interface name (this *InterfaceName.MethodName* notation is used by the reflection API of the CLR). To call method `foo` of interface `IFoo`, for example, a script should use the `obj["IFoo.foo"](obj,...)` expression.

Finally, to get a reference to a nested type, a script calls `import_type` with the name of the nested type following the name of the containing type and a plus sign. Again, this notation is used by the reflection API of the CLR. An example of importing a nested type using this notation is the `import_type("ContainingType+NestedType")` statement.

2.7 Creating new CTS types

LuaInterface provides the `make_object` function to create new types. The function takes a Lua object and a CTS interface. LuaInterface automatically creates a new class that implements the interface. The constructor of this class receives a Lua object, storing it. The methods of this class delegate their execution to methods of the stored Lua object. After creating the class, LuaInterface instantiates it, passing the Lua object to the constructor. The `make_object` function returns the newly instantiated object. For example, let `IExample` be an interface defined by the following C# code:

```
public interface IExample {
  float Task(float arg1, float arg2);
}
```

The `IExample` interface defines a method called `Task` that takes two `float` arguments and returns a `float`. Now, let `tab` be a Lua table defined by the following Lua code:

```
tab = { mult = 2 }
function tab:Task(arg1, arg2)
  return self.mult * arg1 * arg2
end
```

The `tab` table also defines a method called `Task`, which takes two arguments, multiplies them, then multiplies the result by a field of `tab` called `mult` and returns the final result. Let's now define a class that uses `IExample` instances, with the C# code:

```
public class TestExample {
  public static void DoTask(IExample ex, float arg1,
      float arg2) {
    Console.WriteLine(ex.Task(arg1, arg2));
  }
}
```

The `TestExample` class defines a static `DoTask` method that takes an `IExample` instance and two `float` values as arguments, then calls method `Task` on the `IExample` instance, passing both `float` values, and prints the result on the console. We finish this example with the following Lua code:

```
ex = make_object(tab, IExample)
TestExample:DoTask(ex, 2, 3)
```

This code assumes that references to the `IExample` interface and the `TestExample` class have already been imported. The call to `make_object` creates an instance of the `IExample` interface that delegates its `Task` method to `tab`. The last line calls method `DoTask` of class `TestExample`, passing the instance that `make_object` created, plus numbers 2 and 3. Inside `DoTask`, the `Task` method of this instance is called with numbers 2 and 3. This calls method `Task` on `tab`, and the result (12) is returned to `DoTask` and printed on the console.

Whenever a script passes a Lua object where the CLR is expecting an interface instance, LuaInterface automatically calls `make_object` with the Lua object and the interface type, and passes the object `make_object` returns instead. In the previous example, the last excerpt of Lua code could be rewritten as the following code, with the same result:

```
TestExample:DoTask(tab, 2, 3)
```

The `make_object` function can actually take any class, not just interfaces. It creates a new subclass of this class. This feature is not fully implemented, and has some issues. The LuaInterface manual [Mascarenhas, 2000] provides more details.

3 Implementation of LuaInterface

We wrote LuaInterface mostly in C#, with a tiny (less than 30 lines) part in C. This part needs minimal changes for each platform. The library depends on Lua version 5.0, and assumes the existence of a DLL or shared library containing the implementation of the Lua API, and a library containing the implementation of the Lua library API.

3.1 Wrapping the Lua API

LuaInterface accesses the Lua API functions through Platform/Invoke (P/Invoke for short), the native-code interface of the CLR. Access is straightforward, with each function exported by the Lua libraries corresponding to a static method in the C# code of LuaInterface. For example, the C prototype

```
void lua_pushstring(lua_State *L, const char* s);
```

translated to C# becomes

```
static extern void lua_pushstring(IntPtr L, string s);
```

P/Invoke automatically marshals basic types from the CLR to C. It marshals delegates as function pointers, so passing methods to Lua is almost straightforward; almost, because the application must remember to keep references to all delegates passed to C, otherwise they may be collected and their function pointers invalidated. Pointers are marshalled as instances of `IntPtr`, an opaque, immutable type that can be passed back to C as the original pointer.

Another small difficulty is a conflict of function calling conventions. C compilers use the CDECL calling convention by default (caller cleans the stack), while the Microsoft .NET compilers use the STDCALL convention (callee cleans the stack). This means P/Invoke marshals delegates as STDCALL function pointers, while Lua expects CDECL function pointers, leading to program crashes. The solution was to write an extension to the Lua API, a function that wraps a STDCALL function pointer inside a CDECL function, and passes a pointer to the CDECL function to Lua (this is the reason for the C part in our implementation).

The API has functions to convert Lua numbers, strings and booleans to C, and vice-versa. P/Invoke converts from C to the CLR, so the implementation of the `Lua` class just calls the Lua API functions when converting to (or from) numbers, strings and booleans. Instances of the `LuaTable` class contain a reference to the actual Lua table, and use the API functions to access its fields. Functions are converted in a similar fashion.

3.2 Passing CLR objects to Lua

Lua has a data type, called *userdata*, that lets an application pass arbitrary data to the interpreter. When an application creates a new userdata the interpreter allocates space for it, then returns a pointer to the allocated space. The behavior of an userdata is extensible; the application can attach functions to the userdata so Lua can call them when it is garbage-collected, indexed as a table, called as a function, or compared to other values.

When LuaInterface needs to pass a CLR object to Lua, it stores the object inside a CLR vector, creates a new userdata, stores the index of the object (in the vector) inside this userdata, and passes it instead. LuaInterface also stores a reference to the userdata inside a Lua table. This table lets LuaInterface reuse an userdata it already created. It stores weak references to the userdata, so the interpreter can eventually collect them. When the interpreter collects an userdata, its finalizer function removes the original object from the vector.

3.3 Using CLR objects from Lua

A Lua method call, such as the `obj:foo(arg1,arg2)` expression, is syntactic sugar for `obj["foo"](obj,arg1,arg2)`. Suppose `obj` is an userdata that represents a CLR object; because `obj` is not a table, the `obj["foo"]` operation triggers a function to handle it. This function searches in the type of object, using the reflection API of the CLR, for methods called `foo`. If one or more methods are found, LuaInterface returns a delegate that represents those methods. It returns `nil` if no method is found. The Lua interpreter then calls the delegate returned by LuaInterface. The arguments to the call are `obj`, `arg1` and `arg2`. The delegate converts `arg1` and `arg2` to the types the method expects, and calls the method `foo` on object `obj`. If `foo` is overloaded, the delegate first checks which method accepts two arguments with compatible types, and then calls that method.

Using reflection to look for a method, and then creating a delegate for it, are costly operations. So, LuaInterface caches the delegates, to pay the cost only once, when the method is first called. Objects belonging to the same type share the same cache. If a method is overloaded, there is also the cost of checking for the method that matches the arguments. In any case, there is also the cost of deciding how the arguments are converted to the types the method expects. So, LuaInterface has a second cache, inside the delegate, that stores the last version called, as well as the functions it uses to convert the arguments. The delegate first tries the method in its cache; the delegate goes back to check for a matching method if there is a problem.

Returning to the `obj["foo"]` example, suppose `foo` is a field, instead of a method. In this case, the `obj["foo"]` operation will find a field called `foo`, instead of a method, inside the type of the object. The operation then returns the value of the field. LuaInterface also stores the field in a cache, so the cost of looking for the field is paid only once. This cache is also shared by all instances of a type. If `foo` is a property or an event the operation is similar. If a script tries to assign to a field, as in `obj.foo=val`, the assignment triggers another function to handle it. This function receives `obj`, `"foo"` and `val` as arguments. LuaInterface searches for a `foo` field in the type of the object, converts the value to be assigned to the type of the field, and completes the assignment. Assignment to properties proceeds similarly. LuaInterface also uses, for this operation, the same cache it uses when reading fields and properties.

Type references returned by the `import_type` function are instances of class `Type`; their reflexive searches (for methods, fields, etc.) are limited to static members, but otherwise they are just like other object instances. When a script calls a type reference (to instantiate a new object), LuaInterface calls a function that searches in the constructors for the type, calling the one that matches the arguments.

3.4 Delegates, events, and interfaces

Every time a Lua function is passed where the CLR expects a delegate, LuaInterface dynamically creates a subclass of `LuaDelegate`. This subclass defines a method with the signature of the delegate, and a constructor that receives a Lua function as its argument. LuaInterface then creates an instance of this subclass, and creates a delegate from this instance. The newly created delegate is passed to the CLR. The subclass of `LuaDelegate` is created using the `Reflection.Emit` API of the CLR. The `Reflection.Emit` API has classes to generate assemblies, types, and emit Common Intermediate Language bytecodes. The types created by the API can be kept in memory or committed to disk. The class that LuaInterface generates are kept only in memory. LuaInterface stores the class in a cache, and it is reused if LuaInterface needs another delegate with the same signature.

LuaInterface returns events as objects that implement an `Add` and a `Remove` method. The `Add` method receives a Lua function and creates a delegate with the same signature as the handlers of the event, then registers this delegate as a handler, returning the delegate. Method `Remove` receives a delegate that was previously registered by `Add` and removes it. The `make_object` function also uses the `Reflection.Emit` API of the CLR to generate its classes.

4 Related Work

The LuaPlus distribution is a C++ interface for the Lua interpreter that includes a CLR wrapper to the Lua API [Jensen, 2003]. The CLR wrapper is similar to the LuaInterface API wrapper: it has methods to run Lua code, to read and write Lua globals, and to register delegates as Lua functions. Arbitrary CLR objects are passed to the interpreter as userdata, but Lua scripts cannot use their properties and methods. The delegates that LuaPlus registers as Lua functions also cannot have an arbitrary signature, as they can in LuaInterface. LuaPlus does not offer the same level of integration that LuaInterface offers.

LuaOrb is a library, implemented in C++, for scripting CORBA objects and implementing CORBA interfaces [Cerqueira et al., 1999] in Lua. As LuaInterface, LuaOrb uses reflection to access properties and to call methods of CORBA objects, using Lua syntax. LuaOrb can register Lua tables as implementations of CORBA interfaces, through the CORBA Dynamic Skeleton Interface. This interface has no similar in CLR, although a similar feature was implemented for LuaInterface by runtime code generation through `Reflection.Emit`.

LuaJava is a scripting tool for Java. It allows Lua scripts to use Java objects and create classes from Lua tables [Cassino et al., 1999]. On the consumer side, LuaJava uses Java reflection to find properties and methods and the Java native code API to access the Lua C API, an approach very similar to the one in LuaInterface. On the extender side, it uses dynamic generation of bytecodes to create Java classes from tables. LuaJava generates a class that delegates method calls to the Lua table, and this class is loaded by a custom class loader. The `Reflection.Emit` API of the CLR makes this task much easier, with its utility classes and methods for generating and loading memory-only classes.

The Script for the .NET Framework [Clinick, 2001], by Microsoft, is a set of script engines that a CLR application can host. It provides two engines by default, a Visual Basic engine and a JScript engine. Scripts have full access to CTS classes and the application can make its objects available to them. The scripts are compiled to the CLR Intermediate Language (IL) before they are executed.

PerlNET [Dubois, 2002] is a commercial library, by ActiveState, that integrates the Perl interpreter to the CLR. Like LuaInterface, PerlNET uses P/Invoke to bridge the Perl 5.6 interpreter and the CLR. On the consumer side, Perl scripts can instantiate CLR objects and call their methods using Perl syntax. On the extender side, PerlNET packages Perl classes and modules as CTS classes, with their functions and methods visible to other objects. The classes that PerlNET generates are permanent, and can define new methods that are visible from the CLR. The classes generated by LuaInterface are temporary, and can only override methods of their base classes.

Dot-Scheme [Pinto, 2003] is a bridge between PLT Scheme and the CLR. Dot-Scheme is similar to LuaInterface on the consumer side; Scheme programs can instantiate and use CLR objects using the Scheme syntax. It is not a full CLS consumer, though, offering no way for Scheme code to define CLR delegates or handle CLR events.

LuaInterface is also related to other bridges between scripting languages and the Java Virtual Machine, such as Tcl Blend [DeJong and Redman, 2003] and the Java-Python Extension [Giacometti, 2003]. These bridges use the Java Native Interface and Java reflection to instantiate and use JVM objects. The bridge between the Hugs98 Haskell interpreter and the CLR [Finne, 2003] is yet another example of related work.

The following section presents an evaluation of the performance of LuaInterface, comparing its performance to the performance of PerlNET. We only compare LuaInterface with PerlNET; LuaPlus does not offer the feature we are comparing, LuaORB and LuaJava do not use the CLR, the Script for the .NET Framework uses a completely different approach (the scripts are compiled to IL), and Dot-Scheme has, according to its author, not yet been optimized for performance.

4.1 Performance evaluation

We focused the performance tests on CLR method calls from Lua scripts, as this is the main feature of LuaInterface. We evaluated times for calls to six distinct methods. They vary by the number and types of their parameters. Three of the methods have all parameter and return values of type `Int32`, and vary by the number of parameters (zero, one or two). The times for these method calls, in microseconds, are shown on Figure 1. The other three methods in this evaluation have object types as parameter and return values, and also vary by their number of parameters. Figure 2 shows the times for these method calls. All the times were collected in the same machine, under the same conditions, and are an average of ten different runs of a million method calls each[1].

The *MethodBase.Invoke* column shows the minimum possible times for reflexive method calls. The *Cache* column shows the time for a method call from Lua, when the method is already in cache. The *Partial cache* column shows the time with the internal cache of the delegate disabled (which is also the time for a call when there is a type mismatch), and the *No cache* column shows the time with both caches disabled (which is the time for the first call of a method). The *PerlNET* column shows the time for a method call from the Perl language, using the PerlNET bridge.

Both figures show how it is costly to search for a method and to find how each argument must be converted, in the difference among method calls from Lua. The times using the caches are about a fifth of the times without caching.

[1] Athlon 1.2GHz, with 256Mb memory, running Windows XP Professional with version 1.1 of the .NET Common Language Runtime. The Lua interpreter was compiled with the Microsoft VC.NET compiler, with all optimizations turned on. No other applications were running while the times were being collected.

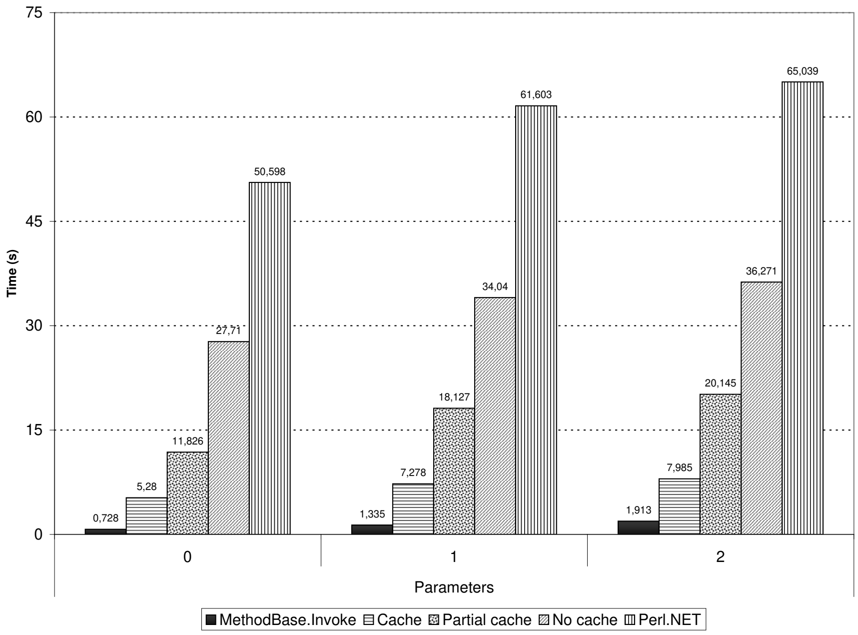

Figure 1: Times for method calls with `System.Int32` parameters

The slight increase in the times for method calls from Lua, when comparing Figures 1 and 2, comes from the checks that LuaInterface must make on values of object types. When LuaInterface encounters an argument of type userdata, it must check whether the userdata really represents a CLR object. Finally, the P/Invoke API itself brings some overhead. Each P/Invoke call adds ten to thirty CPU instructions, possibly more, depending on the types used [Microsoft, 2003]. The overhead in each call to the Lua API is small, but each method call involves several API calls, so they add up to about a fifth of the total time.

5 Conclusions and Future Work

This paper presented LuaInterface, a library that gives Lua the capabilities of a full CLS consumer. Lua scripts can use LuaInterface to instantiate and use CLR objects, and CLR applications can use LuaInterface to run Lua code. We implemented the library in C#, with a tiny part in C. The C part, and the Lua interpreter, compile in all the platforms where the CLR is available with minimal changes, so portability was not affected. The Lua interpreter was designed to be easily embeddable; access to the interpreter was straightforward, through the

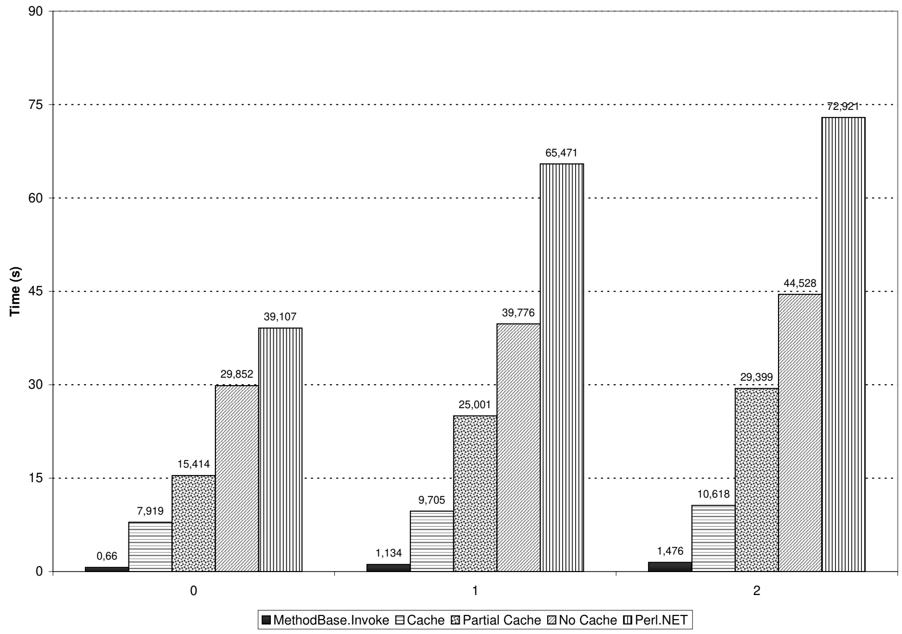

Figure 2: Times for method calls with object type parameters

P/Invoke library of the CLR. We created an object-oriented wrapper to the Lua C API functions, providing a more natural interface for CLR applications.

The times for CLR method calls from Lua are about five to eight times the minimum time for reflexive method calls, except for the first time a method is called, when times can climb to thirty to forty times the minimum. Yet LuaInterface is five to ten times faster than a similar commercial library, for the Perl language.

What we learned during the course of this project:

- The extensibility of Lua made it easy to implement the full CLS consumer capabilities without any changes to the interpreter or language, and without the need for a preprocessor.
- The dynamic typing of the Lua language and the reflection API of the CLR were crucial for the lightweight approach to integration that was used in this project.
- Reflexive calls are not the performance bottleneck for the library, as we initially thought it would be, although searching for a method using reflection is;

– P/Invoke is very easy to use and very clean, but its overhead adds up to a fifth of the total time for a method call, more than we expected. We use P/Invoke as little as possible.

The approach we used to build LuaInterface can also be used to build bridges for other scripting languages. Any language that has a native code interface, dynamic typing, and a way to dynamically extend the behavior of its objects, can use the same approach. Most scripting languages have these prerequisites, by their nature of "glue" languages.

LuaInterface has a public release [Mascarenhas and Ierusalimschy, 2003]. We are now working on another approach of integration between Lua and the CLR, through compilation of Lua code directly to the Intermediate Language of the CLR. There is already a prototype that compiles Lua virtual-machine bytecodes to IL. Performance tests show that the generated code is faster than similar code generated by the Microsoft JScript compiler for the CLR.

References

[ActiveState, 2000] ActiveState (2000). Release Information for the ActiveState Perl for .NET compiler. Available at `http://www.activestate.com/Corporate/Initiatives/NET/Perl_release.html`.

[Bock, 2003] Bock, J. (2003). .NET Languages. Available at `http://www.jasonbock.net/dotnetlanguages.html`.

[Cassino et al., 1999] Cassino, C., Ierusalimschy, R., and Rodriguez, N. (1999). LuaJava — A Scripting Tool for Java. Technical report, Computer Science Department, PUC-Rio. Available at `http://www.tecgraf.puc-rio.br/~cassino/luajava/index.html`.

[Cerqueira et al., 1999] Cerqueira, R., Cassino, C., and Ierusalimschy, R. (1999). Dynamic Component Gluing Across Different Componentware Systems. In *International Symposium on Distributed Objects and Applications (DOA'99)*.

[Clinick, 2001] Clinick, A. (2001). Script Happens .NET. Available at `http://msdn.microsoft.com/library/en-us/dnclinic/html/scripting06112001.asp`.

[DeJong and Redman, 2003] DeJong, M. and Redman, S. (2003). The Tcl/Java Project. Available at `http://tcljava.sourceforge.net/docs/website/index.html`.

[Dubois, 2002] Dubois, J. (2002). PerlNET — The Camel Talks .NET. In *The Perl 6 Conference*. Available at `http://conferences.oreillynet.com/presentations/os2002/dubois_update.ppt`.

[Finne, 2003] Finne, S. (2003). Hugs98 for .NET. Available at `http://galois.com/~sof/hugs98.net/`.

[Giacometti, 2003] Giacometti, F. (2003). JPE – The Java-Python Extension. Available at `http://jpe.sourceforge.net/`.

[Hammond, 2000] Hammond, M. (2000). Python for .NET: Lessons Learned. Available at `http://www.activestate.com/Corporate/Initiatives/NET/Python_for_.NET_whitepaper.pdf`.

[Ierusalimschy, 2003] Ierusalimschy, R. (2003). *Programming in Lua*. Lua.org.

[Ierusalimschy et al., 1996] Ierusalimschy, R., Figueiredo, L. H., and Celes, W. (1996). Lua — An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652.

[Inc., 2000] Inc., S. (2000). S#.NET Tech-preview Software Release. Available at `http://www.smallscript.com/Community/calendar_home.asp`.

[Jensen, 2003] Jensen, J. C. (2003). LuaPlus 5.0 Distribution. Available at `http://wwhiz.com/LuaPlus/index.html`.

[Mascarenhas, 2000] Mascarenhas, F. (2000). *LuaInterface: User's Guide*. Computer Science Department, PUC-Rio. Available at `http://www.inf.puc-rio.br/~mascarenhas/luainterface/manual-en.pdf`.

[Mascarenhas and Ierusalimschy, 2003] Mascarenhas, F. and Ierusalimschy, R. (2003). LuaInterface: Scripting .NET with Lua. Available at `http://www.inf.puc-rio.br/~mascarenhas/luainterface/`.

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

[Microsoft, 2002] Microsoft (2002). ECMA C# and Common Language Infrastructure Standards. Available at `http://msdn.microsoft.com/net/ecma/`.

[Microsoft, 2003] Microsoft (2003). Managed Extensions for C++ Migration Guide: Platform Invocation Services. Available at `http://msdn.microsoft.com/library/en-us/vcmxspec/html/vcmg_PlatformInvocationServices.asp`.

[Ousterhout, 1998] Ousterhout, J. K. (1998). Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30.

[Pinto, 2003] Pinto, P. (2003). Dot-Scheme — A PLT Scheme FFI for the .NET framework. In *Fourth Workshop on Scheme and Functional Programming*.

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

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

All URLs in these references are valid as of April 12th, 2004.