LuaInterface: Scripting the .NET CLR with Lua

Fabio Mascarenhas[1] **, Roberto Ierusalimschy**[1]

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

mascarenhas@acm.org, roberto@inf.puc-rio.br

Abstract. *In this paper we present LuaInterface, a library for scripting the .NET CLR with Lua. The .NET Common Language Runtime aims to provide interoperability among objects written in several different languages. LuaInterface is a library for the CLR that lets Lua script objects in any language that runs in the CLR. It gives Lua the capabilities of a full CLS consumer. The Common Language Specification is a subset of the CLR with rules for language interoperability, and languages that can use CLS-compliant libraries are called CLS consumers. Applications may also use LuaInterface to embed a Lua interpreter and use Lua as a language for configuration scripts or for extending the application. LuaInterface is part of the Lua.NET project for integration of Lua into the .NET Common Language Infrastructure.*

1. Introduction

The Microsoft .NET Framework aims to provide interoperability among several different languages through its Common Language Runtime (CLR) [13]. The CLR specification is being turned into ISO and ECMA standards [14], and implementations for non-Windows platforms already exist [17, 18]. Visual Basic, JScript, C#, J#, and C++ already have compilers for the CLR, written by Microsoft, and compilers for several other languages are under development [2].

Lua is a scripting language designed for to be simple, portable, to have a small footprint, and to be easily embeddable into other languages [8, 10]. Scripting languages are often used for connecting components written in other languages to form applications (“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 declaration of types 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 [16].

This work presents LuaInterface, a library for the CLR that allows Lua scripts to access the object model of the CLR, the Common Type System (CTS), turning Lua into a scripting language for components written in any language that runs in the CLR. LuaInterface is part of the Lua.NET project for integration of Lua into the CLR [9].

LuaInterface provides all the capabilities of a full CLS consumer. The Common Language Specification (CLS) is a subset of the CLR that establishes a set of rules to promote language interoperability. 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 CLS consumer should be able to call any CLS-compliant method or delegate, even methods named after keywords of the language; to call distinct methods of a type with the same name and signature but from different interfaces; to instantiate any CLS-compliant type, including nested types; and to read and write any CLS-compliant property and access any CLS-compliant event [14, CLI Partition I Section 7.2.2].

With LuaInterface, Lua scripts can instantiate CTS types, access their fields, and call their methods (both static and instance), all using the standard Lua syntax. CLR applications can run Lua code, acess Lua data, call Lua functions, and register CLR methods as functions. Applications can use Lua as a language for their configuration scripts or as an embedded scripting language, and Lua cripts can glue together different components. Besides these consumer facilities there is also a limited support for dynamically creating new CTS types, but it will not be covered in this paper.

Lua is dynamically typed, so it needs no type declarations to instantiate or use CLR objects. It checks at runtime the correctness of each instantiation, field access, or method call. LuaInterface makes extensive use of the reflexive features of the CLR, without the need of preprocessing or creating stubs for each object that needs to be accessed. Its implementation required no changes to the Lua interpreter: the interpreter is compiled to an unmanaged dynamic linked library and the CLR interfaces with it using P/Invoke.

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, with basic performance measurements. Section 4 presents some related work and comments on their strengths and drawbacks relative to LuaInterface, and Section 5 presents some conclusions and future developments.

2. Interfacing Lua and the CLR

As an embeddable language, Lua has an API that lets an application instantiate a Lua interpreter, run Lua code, exchange data between the application and the interpreter, call Lua functions, and register functions so they can be called from Lua [11]. 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 Lua tables' (associative arrays) fields and to call Lua functions. LuaInterface also has the capabilities of a full CLS consumer, so Lua code can instantiate CLR objects and access their their properties and methods.

Functions are first-class values in Lua, so Lua objects are just tables, and functions stored in

fields are 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 dot (.) operator is used for fields, with `obj.field="foo"` meaning `obj["field"]="foo"`, for example. The colon (:) operator is used to call methods. A method call like `obj:foo(arg1,arg2)` is syntactic sugar for `obj["foo"](obj,arg1,arg2)`, that is, the object goes as the first argument to the call.

2.1. The API wrapper

Applications start a new Lua interpreter by instantiating an object of class `Lua`. Multiple instances may be created, and they are completely independent. Methods `DoFile` and `DoString` execute a Lua source file and a Lua chunk, respectively. Access to global variables is through the class indexer, indexed by variable name. The indexer returns Lua values with the equivalent CTS value type: `nil` as `null`, numbers as `System.Double` (the Lua interpreter uses doubles to represent all numbers), strings as `System.String`, and booleans as `System.Boolean`. The following C# code shows the usage of these methods:

```
// 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";
```

The indexer returns Lua tables as `LuaTable` objects, which have their own indexers to read and write table fields, indexed by name or by numbers (arrays in Lua are just tables indexed by numbers). They work just like the indexers in class `Lua`. Lua functions are returned as `LuaFunction` objects. Their `call` method calls the corresponding function and returns an array with the function's return values.

LuaInterface converts CLR values passed to Lua (either as a global or as an argument to a function) into the appropriate Lua types: numeric values to Lua numbers, strings to Lua strings, booleans to Lua booleans, `null` to `nil`, `LuaTable` objects to the wrapped table, and `LuaFunction` objects to the wrapped function.

2.2. Loading CTS types and instantiating objects

Scripts need a type reference to instantiate new objects. They need two functions to get a type reference. First they should use `load_assembly`, which loads the specified assembly, making its types available to be imported as type references. Then they should use `import_type`, which searches the loaded assemblies for the specified type and returns a reference to it. The following excerpt shows 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")
StartPosition = import_type("System.Windows.Forms.FormStartPosition")
```

Notice how scripts can use `import_type` to get type references for structures (`Point`) and enumerations (`FormStartPosition`), as well as classes.

Scripts call static methods through type references, using the same syntax of Lua objects. For example, `Form:GetAutoScaleSize(arg)` calls the `GetAutoScaleSize` method of class `Form`. LuaInterface does lookup of static methods dynamically by the number and type of arguments. Scripts also read and write to static fields and non-indexed properties through type references, also with the same syntax of Lua objects. For example, `var=Form.ActiveForm` assigns the value of the `ActiveForm` property of class `Form` to the variable `var`. LuaInterface treats enumeration values as fields of the corresponding enumeration type.

LuaInteface converts arguments to the parameter type not the original Lua type. For example, a number passed to a C# method expecting a `System.Int32` value is converted to `System.Int32`, not to `System.Double`. LuaInterface coerces numerical strings into numbers, numbers into strings and Lua functions into delegates. The same conversions apply to fields and non-indexed properties, with values converted to the field type or property type, respectively.

To instantiate a new CTS object a script calls the respective type reference as a function. The first constructor that matches the number and type of the parameters is used. The following example extends the previous example to show some of the discussed features:

```
form1 = Form()
button1 = Button()
button2 = Button()
position = Point(10,10)
start_position = StartPosition.CenterScreen
```

2.3. Accessing other CTS types

Only numeric values, strings, booleans, `null`, `LuaTable` instances and `LuaFunction` instances have a mapping to a basic Lua type that LuaInterface uses when passing them from the CLR to Lua. LuaInterface passes all other objects as references stored inside an userdata value (an userdata is a Lua type for application-specific data). Scripts read and write an object's fields and non-indexed properties as fields of Lua objects, and call methods as methods Lua objects. To read and write indexed properties (including indexers) they must use their respective `get` and `set` methods (usually called `get_`*`PropertyName`* and `set_`*`PropertyName`*).

The same considerations about method matching and type coercion that apply for accessing static members apply for accessing instance members. The following Lua code extends the previous examples to show how to access properties and methods:

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

```
form1.StartPosition = start_position
form1:ShowDialog()
```

The three previous examples combined, when run, show a form with two buttons, on the center of the screen.

Scripts can register Lua functions as event handlers by calling the event's `Add` pseudo-method. The call takes a Lua function as the sole argument, and automatically converts this function to a `Delegate` instance with the appropriate signature. It also returns the created delegate, allowing deregistration through the event's `Remove` pseudo-method. The following Lua code extends the previous examples to add event handlers to both buttons:

```
function handle_mouseup(sender,args)
  print(sender:ToString() .. " MouseUp!")
  button1.MouseUp:Remove(handler1)
end
handler1 = button1.MouseUp:Add(handle_mouseup)
handler2 = button2.Click:Add(exit)  -- exit is a standard Lua function
```

Scripts can also register and deregister handlers by calling the object's `add` and `remove` methods for the event (usually called `add_`*`EventName`* and `remove_`*`EventName`*).

LuaInterface passes any exception that occurs during execution of a CLR method to Lua as an error, with the exception object as the error message (Lua error messages are not restricted to strings). Lua has mechanisms for capturing and treating those errors.

LuaInterface also provides a shortcut for indexing single-dimension arrays (either to get or set values), by indexing the array reference with a number, for example, `arr[3]`. For multidimensional arrays scripts should use the methods of class `Array` instead.

2.4. 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 needed capabilities.

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 method's return value, in the order they appear in the method's signature. The method call should ommit *out* parameters.

The standard method selection of LuaInterface uses the first method that matches the number and type of the call's arguments, so some methods of an object may never be selected. For those cases, LuaInterface provides the function `get_method_bysig`. It takes an object or type reference, the method name, and a list of type references corresponding to the method parameters. It returns a function that, when called, executes the desired method. If it is an instance method 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. There is also a `get_constructor_bysig` function that does the same thing

for constructors. It takes as parameters a type reference that will be searched for the constructor and zero or more type references, one for each parameter. It returns a function that, when called, instantiates an object of the desired type with the matching constructor.

If a script wants to call a method with a Lua keyword as its name the `obj:method(...)` syntax cannot be used. For a method named `function`, for example, the script should call it using `obj["function"](obj,...)`.

To call distinct methods with the same name and signature, but belonging to different interfaces, scripts can prefix the method name with the interface name. If the method is called `foo`, for example, and its interface is `IFoo`, the method call should be `obj["IFoo.foo"](obj,...)`.

Finally, to get a reference to a nested type a script can call `import_type` with the nested type's name following the containing type's name, like in `import_type("ContainingType+NestedType")`.

3. Implementation of LuaInterface

We wrote LuaInterface mostly in C#, with a tiny (less than 30 lines) stub in C. The current version uses Lua version 5.0. The C# code is platform-neutral, but the stub must be changed depending on the standard calling convention used by the CLR on a specific platform. The implementation assumes the existence of a DLL or shared library named `lua.dll` containing the implementation of the Lua API plus the stub code, and a library named `lualib.dll` 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 CLR's native code interface. Access is straightforward, with each function exported by the Lua libraries corresponding to a static method in LuaInterface's C# code. For example, the following C prototype:

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

when translated to C# is:

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

P/Invoke automatically marshalls basic types from the CLR to C. It marshalls delegates as function pointers, so passing methods to Lua is almost straightforward, for care must be taken so the CLR garbage collector will not collect the delegates. In Windows there is also a conflict of function calling conventions. The C compilers use CDECL calling convention by default (caller cleans the stack) while the Microsoft .NET compilers use STDCALL as default (callee cleans the stack), so we wrote a tiny stub C stub which exports a function that receives an explicit STDCALL function pointer and passes it to the Lua interpreter wrapped inside a CDECL function.

Implementing the `Lua` wrapper class and its methods that deal with Lua standard types was easy once the Lua API was fully available to C# programs. The API has functions to convert Lua numbers to C

doubles and C doubles to Lua numbers. It also has functions to convert Lua strings to C strings (char*) and C strings to Lua strings, and functions to convert Lua booleans to C booleans (integers) and C booleans to Lua booleans. The `Lua` class' indexer just calls these functions when numbers, strings and booleans are involved.

The indexer returns tables and functions as `LuaTable` and `LuaFunction` instances, respectively, containing a Lua reference (an integer), and CLR applications access or call them through the appropriate API functions. When the CLR garbage collector collects the instances LuaInterface removes their Lua references so the interpreter may collect them.

3.2. Passing CLR objects to Lua

Lua has a data type called userdata that lets an application pass arbitrary data to the interpreter and later retrieve it. When an application creates a new userdata the intrepreter allocates space for it and returns a pointer to the allocated space. The application can attach functions to an userdata to be called 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 list (to keep the CLR from collecting it), creates a new userdata, stores the index (in the list) of the object inside this userdata, and passes the userdata instead. A reference to the userdata is also stored, with the same index, inside a Lua table. This table is used if the object was already passed earlier, to reuse the already created userdata instead of creating another one (avoiding aliasing). This table stores weak references so the interpreter can eventually collect the userdata. When the interpreter collects it the original object must be removed from the list. This is done by the userdata's finalizer function.

3.3. Using CLR objects from Lua

When a script calls a CLR method, such as `obj:foo(arg1,arg2)`, the Lua interpreter first converts the call to `obj["foo"](arg1,arg2)`, which is an indexing operation (`obj["foo"]`) followed by a call to the value returned by it.

The indexing operation for CLR objects is implemented by a Lua function. It checks if the method is already in the object type's method cache. If it is not, the function calls a C# function which returns a delegate to represent the method and stores it in the object type's method cache.

When the interpreter calls the delegate for a method it first checks another cache to see if this method has been called before. This cache stores the `MethodBase` object representing the method (or one of its overloaded versions), along with a pre-allocated array of arguments, an array of functions to get the arguments' values from Lua with the correct types, and an array with the positions of *out* and *ref* parameters (so the delegate can return their values). If there is a method in the cache the delegate tries this method first. If the cache is empty or the call fails due to a wrong signature, the delegate checks all overloaded versions of the method one by one to find a match. If it finds one it stores the method in the cache and calls it, otherwise it throws an exception.

To read fields LuaInterface uses the same C# function that returns the method delegate, but now it returns the value of the field. Non-indexed properties and events use this same technique, but events return an object used for registration/deregistration of event handlers. This object implements the event's `Add` and `Remove` pseudo-methods.

LuaInterface uses another C# function to treat assignment to fields and non-indexed properties. It retrieves the object from the userdata, uses reflection to try to find a property or field with the given name and, if found, converts the assigned value to the property type or field type and stores it.

Type references returned by the `import_type` function are instances of class `Type`, with their own assignment and indexing functions. They search for static members only, but otherwise work just like the assignment and indexing functions of normal object instances. When a script calls a type reference to instantiate a new object, LuaInterface calls a function which searches the type's constructors for a matching one, instantiating an object of that type if it finds a match.

3.4. Performance of CLR method calls

We ran simple performance tests to gauge the overhead of calling a CLR method from a Lua script. On average the calls were five times slower than calling the same method from C# using reflection (with `MethodBase.Invoke`). Most of the overhead is from P/Invoke: each P/Invoke call generates from ten to thirty CPU instructions plus what is needed for security checking and argument marshalling [15]. One call is needed for each argument of the method plus one for the receiver, one for the delegate, one for each returned value, and one call to get the number of arguments passed to the method.

The rest of the overhead (a fifth of the call's time, approximately) is from Lua itself, as each method call is also a Lua function call which checks a Lua table (the method cache). Implementing this cache in C# just makes performance worse (by a factor of 2.5), as three more P/Invoke calls are needed to get the receiver of the method, the method's name and then returning the delegate.

Removing the second level of caching so every method call needs to match the arguments against the method's overloaded versions and their parameters worsens the performance by a factor of three. The naive implementation (no caching at all) is much worse (by about two orders of magnitude), as each method call involves the creation of a new delegate.

4. Related Work

The LuaPlus distribution [12] has some of the features of LuaInterface. It provides a managed C++ wrapper to the Lua API that is similar to LuaInterface's API wrapper, with methods to run Lua code, to read and write Lua globals, and to register delegates (with a specific signature) as Lua functions. Arbitrary CLR objects may be passed to the interpreter as userdata, but Lua scripts cannot access their properties and methods, and applications cannot register methods with arbitrary signatures as Lua functions.

LuaOrb is a library, implemented in C++, for scripting CORBA objects and implementing CORBA interfaces [5, 6]. As LuaInterface, LuaOrb uses reflection to access properties and to call methods of CORBA objects. Registering Lua tables as implementations of CORBA interfaces is done through CORBA's Dynamic Server Interface, which 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 that allows Lua scripts to access Java objects and create Java objects from Lua tables [3, 4]. LuaJava uses an approach very similar to the one in LuaInterface to access Java objects, using Java reflection to find properties and methods and Java's native code API to acess the Lua C API. It uses dynamic generation of bytecodes to create Java objects from tables, generating a class that delegates attribute access and method calling to the Lua table. This class is loaded by a custom class loader. The CLR's `Reflection.Emit` interface made this task much easier, with its utility classes and methods for generating and loading Intermediate Language (IL) code.

Microsoft's Script for the .NET Framework [7] 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 CLR's Intermediate Language (IL) before they are executed, instead of being directly executed by a separate interpreter like LuaInterface does with Lua scripts.

ActiveState's PerlNET [1] gives access to Perl code from the CLR. It packages Perl classes and modules as CTS classes, with their functions and methods visible to other objects. This is accomplished by embedding the interpreter inside the runtime, and using proxies to interface the CLR objects with Perl code. This is very similar to the approach used by LuaInterface, but the types generated by LuaInterface are kept on memory and recreated on each execution instead of being exported to an autonomous assembly on disk.

Other scripting languages have compilers for the CLR in several stages of development, such as SmallTalk (S#), Python, and Ruby [2]. When these compilers are ready these languages may also be used to script CLR applications, but only prototypes are available yet.

5. Conclusions and Future Work

This paper presented LuaInterface, a library that gives Lua scripts full access to CLR types and objects and allows CLR applications to run Lua Code, turning Lua into a glue language for CLR applications. LuaInterface gives Lua the capabilities of a full CLS consumer.

We implemented the library in C# so it is platform-neutral, except for a small C stub. Users can compile the C code (the Lua interpreter and the stub) in all the platforms where the CLR is available, with minimal changes to the stub code.

The Lua interpreter was designed to be easily embeddable, and with the CLR's P/Invoke library access to the interpreter was straightforward. We created an object-oriented wrapper to the C API functions

to provide a more natural interface for CLR applications.

Performance of method calls from Lua is still poor when compared with reflection, although LuaInterface caches method calls. They were about five times slower, on average. Most of the overhead comes from costly P/Invoke function calls.

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;
- Lua's dynamic typing and the CLR's reflection are crucial for the lightweight approach to integration that was used in this project, as the correctness of operations may be checked by the library at runtime;
- Reflection is not the performance bottleneck for the library, as we initially thought it would be;
- P/Invoke is very easy to use and very clean, but much slower than we thought, and became the bottleneck of the library. The CLR documentation could give more emphasis to the performance penalties of using P/Invoke.

LuaInterface is an ongoing project. There is room for improvements with more CLR extension features, as well as further optimization for method calls, reducing the use of P/Invoke or not using it at all. One possible optimization is to reduce the number of P/Invoke calls necessary for each operation. This requires extensions to the API (new C functions). Another optimization is to do a full port of the Lua interpreter to managed code. Both are being considered for future work.

References

[1] ActiveState. PerlNET — Build .NET components using the Perl Dev Kit, 2002. Available at `http://aspn.activestate.com/ASPN/Downloads/PerlNET`.

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

[3] C. Cassino and R. Ierusalimschy. LuaJava — Uma Ferramenta de Scripting para Java. In *Simpósio Brasileiro de Linguagens de Programação (SBLP'99)*, 1999.

[4] C. Cassino, R. Ierusalimschy, and N. Rodriguez. LuaJava — A Scripting Tool for Java. Technical report, 1999. Available at `http://www.tecgraf.puc-rio.br/~cassino/luajava/index.html`.

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

[6] R. Cerqueira, L. Nogueira, and A. Moura. *The LuaOrb Manual*. TeCGraf Computer Science Department, PUC-Rio, 2000. Available at `http://http://www.tecgraf.puc-rio.br/luaorb/`.

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

[8] L. H. Figueiredo, R. Ierusalimschy, and W. Celes. Lua — An Extensible Embedded Language. *Dr. Dobb's Journal*, 21(12):26–33, 1996.

[9] R. Ierusalimschy and R. Cerqueira. Lua.NET: Integrating Lua with Rotor, 2002. Available at `http://www.tecgraf.puc-rio.br/~rcerq/luadotnet/`.

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

[11] 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`.

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

[13] 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`.

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

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

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

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

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