A CORBA Language Mapping for Tcl

Frank Pilhofer*

June 15, 2000

Abstract

Tcl is designed to be a powerful general-purpose scripting language and is often referred to as a glue language, to glue together various parts of an application. Often, these "various parts," or components, are realized as C code that is registered with the interpreter as a custom command.

It is easy to imagine these components as being distributed – a custom command could cause some sort of request to be sent over the network, returning the reply back to the script.

CORBA is a hardware and vendor-independent de-facto standard for distributed computing. By specifying a CORBA language mapping for Tcl and providing access to an Object Request Broker, Tcl scripts can fully interoperate with other CORBA clients and servers.

With the rapid application development features of Tcl and the connectivity of CORBA, the possibilities range from testing of existing CORBA services to graphical user interfaces with the aid of Tk, or even to script-based CORBA servers.

This paper presents the CORBA language mapping for Tcl used in the Combat project, discusses its features and possible improvements, and makes comparisons with similar projects.

1 CORBA

Distributed systems have a number of advantages over centralized processing that are attractive for an equally wide range of applications. However, building distributed systems that require communication across process or host boundaries is inherently complicated, with communication overhead, multiple points of failure, security concerns and synchronization issues. Multiple attempts have been made to provide generic distribution platforms to aid the development of a distributed system and to hide the complications as much as possible, but few have stood the test of time. *Middleware* implements a "middle layer," abstracting the lower network layers and presenting a uniform interface to the application layer according to OSI terminology.

*Send Email to fp@fpx.de

```
interface Calculator {
  double calculate (in string expr);
};
```

Figure 1: Example IDL file

The Common Object Request Broker Architecture (CORBA) [4] is a particularly successful example of a middleware. Published by the Object Management Group (OMG), a consortium of more than 700 companies, it is an open and freely available specification of an infrastructure for distributed objects.

The key component of the Object Management Architecture is the Object Request Broker (ORB), often called "object bus," which allows communication among these objects by directing remote method invocations from client to server.

Part of CORBA's success is its viability in heterogeneous environments. Objects are addressed by opaque Interoperable Object References (IOR) that encapsulate addressing information for various types of networks, of which a client can select the most optimal. The abstract GIOP communications protocol (General Inter-ORB Protocol) is not limited to TCP/IP networks, even if the Internet Inter-ORB Protocol (IIOP) is, at the moment, its only specified implementation.

A remote method invocation is initiated by the client sending a GIOP request over the network. This request includes information about the target object, the method name and the parameter values, but no type information. For this to work, information about available methods and its parameter types must be published in an abstract *Interface Definition Language* (IDL), which is similar to a C++ header file (figure 1).

Clients and servers can employ the type infor-

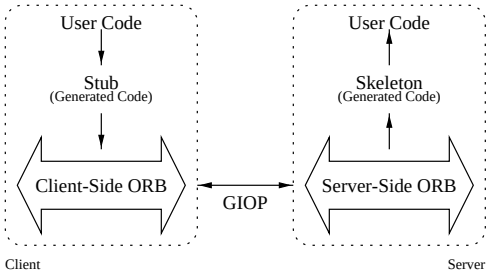

Figure 2: Performing a CORBA invocation

mation contained in the "human-readable" IDL description either by running an IDL "compiler" to create static client-side *stubs* and server-side *skeletons*, or they can load the file into an *Interface Repository* (IFR), where type information can be read from at runtime. (The IFR itself is described in IDL, so clients can use the IFR's generated stubs to access it.)

The appearance of stubs and skeletons, and the representation of basic and composed IDL types, differs between programming languages, and is defined by a "language mapping". Official language mappings [3] currently exist for C, C++, Java, Smalltalk, Ada, COBOL, Python and IDLscript (the latter two not yet finalized). Unofficial language mappings have been proposed for languages such as Eiffel, Visual Basic and Perl, to name a few.

The availability of a wide range of language mappings underlines the attractiveness of CORBA and its language-independent approach. Unlike with other middleware, the developer is not tied to a specific environment but can stick to her favorite language, as long as a language mapping exists, either an official or a self-defined one.

As already hinted at, the language mapping itself is not enough to participate in CORBA communication. The other part is the Object Request Broker, which provides some basic interfaces of its own and the necessary infrastructure for stubs and skeletons, i.e. object reference handling, connection management, the marshaling of parameters into and the retrieval of return values from GIOP messages (see figure 2).

2 Scripting Languages

Scripting languages have proven themselves to be at least as suitable for complex application development as traditional compiled languages [6]. Particularly attractive is the short turnaround time due to the missing compilation step, and the resulting possibilities for rapid application development and one-shot bug fixing. Just as interesting are interactive capabilities of loading a set of code into a running interpreter and testing its subcomponents "live" at the console or with test scripts. Another contributing factor is the usually very lax type system of scripting languages, letting the developer concentrate on the work ahead instead of on type signatures – a major time saver in areas where strict type aherence is not required.

Last but not least, scripts run unchanged on all platforms supported by the interpreter, whereas compiled languages, which are in closer contact with the operating system, often need modifications when ported to a different system.

One popular representative of the scripting language kind is Tcl, the Tool Command Language [9]. Apart from the reasons mentioned above, Tcl is also known for integrating Tk, a set of easy-to-use graphical widgets, which allow the development of attractive user interfaces with a minimal amount of Tcl code.

Tcl has been designed as a glue language; it is easily extensible in both C and in Tcl itself. C code can then itself be used as a glue for third-party packages. Over the years, an impressive set of extensions has appeared, such as Tk, Expect, data encryption, cgi scripting or database access. The fascinating part is that all such extensions can be used in parallel, resulting in synergetic effects, such as database cgi scripts being written in a few lines of Tcl.

A CORBA extension makes another interesting addition to the mosaic, with possibilities like test-scripting existing CORBA servers, graphical user interfaces accessing CORBA services or the rapid development of services themselves, maybe by exploiting existing extensions.

As mentioned, bringing CORBA to Tcl requires both an ORB and a language mapping.

3 Language Mapping

A CORBA language mapping requires basically three parts:

- Representing IDL data types
- Representing client-side stubs
- Representing server-side skeletons

For the mapping to be useful, it should integrate seamlessly into the language, so that using stubs and skeletons appears as natural as possible. In C++, for

IDL Type	Tcl Type
short, long, unsigned short	int
unsigned long, (unsigned) long long	string
float, double, long double	double
char, wchar, string, wstring, octet	string
boolean	boolean
enum	string
fixed	string

Table 1: Mapping of simple IDL types to Tcl

IDL Type	List Members
sequence	{value$_0$ ... value$_n$}
array	{value$_0$ ... value$_n$}
struct	{name$_0$ value$_0$ ... name$_n$ value$_n$}
union	{discriminator value}
TypeCode	{TCKind ...}
any	{typecode value}
exception	{repository-id {struct members}}

Table 2: Mapping of complex IDL types to Tcl

example, stubs are normal C++ objects that expose the methods defined in the IDL description, which then transparently marshal their parameters into a GIOP request, perform necessary upcalls into the ORB, extract the reply and return the server's reply to the caller – in C++, a remote invocation "feels" like a local one.

For a Tcl language mapping, the peculiarities and limitations of Tcl must be kept in mind, such as the lack of a true type system and the lack of an object system.

There is yet no official Tcl language mapping, and an effort initiated by Sun Microsystems and IONA Technologies [5] as a response to the ill-fated, components-centric CORBA Scripting Language RFP has since been abandoned.

The following sections describe the Tcl language mapping implemented in the *Combat* project [7]. Its strengths and weaknesses are explored, so that future discussion can render it suitable for standardization.

3.1 Mapping IDL Types

Earlier versions of Tcl did not provide a type system at all, all data was represented as strings and interpreted according to context. Tcl 8.0 added a backwards-compatible type system by keeping both an "internal" and a "stringified" representation of values. New types can be added by providing two-way conversion functions. Whenever a value is requested to have a certain type (i.e. an integer), first the old type is requested to update the string representation, and then the integer type is asked to scan the string – therefore, strings remain the common denominator for data representation. Tcl provides the "built-in" types integer, double, boolean, list, string, and byte array.

CORBA, in contrast, is built upon a strict type system – a requirement of the GIOP protocol, which does not include type information, so sender and recipient must both know how to interpret incoming binary data.

Table 1 shows how Combat maps simple IDL types to Tcl types. This mapping is straightforward for some types, but not for others. Some numeric types cannot be represented in Tcl, which only provides a signed integer (usually 31 bits plus 1 sign bit) type, so the 32-bit IDL type `unsigned long` could exceed its range. It is therefore rendered as a Tcl string, and errors might result if arithmetic computations are attempted at the Tcl level. The same holds for the infrequently-used `fixed` IDL type, which is represented by a string containing the value in exponential representation: it can be used as a numeric value, but may exceed Tcl's numeric range.

Precision may be lost by mapping the `long double` IDL type (112 bits of mantissa) to Tcl's floating point type, which, depending on the machine's own floating-point format, might be less precise.

Tcl's only complex data type is the list – its string-indexed associative arrays can hardly be counted as such. Arrays do not exist as a separate data type but are only hacked into the normal type system: they cannot be copied, cannot be nested, and they do not have a string representation.

Therefore, the Combat mapping does not use Tcl arrays for complex IDL types, but lists only. Table 2 shows the composition of Tcl lists for constructed IDL types. As an example, IDL sequences and arrays are intuitively mapped to a Tcl list where each list element contains one sequence or array element. For the `struct` type, a list alternating between field names and values is used, for easy access with Tcl's `array get` or `array set` commands. Since IDL types can be nested, so can the Tcl types, and each sequence or struct element can itself be of a complex IDL type. As an exception, sequences and arrays of char and octet are mapped to Tcl strings and byte arrays, so that the "opaque" type `sequence<octet>` can be handled efficiently.

Not shown is the mapping of the `TypeCode` type, whose contents completely describe an IDL type.

The representation differs for each TypeCode kind and is modeled similar to their GIOP encoding.

This mapping translates IDL types to Tcl types in a very intuitive manner, and values can be examined with standard Tcl commands (such as `lindex` and `array`). However, the necessary conversions take time, and can be considered too slow. A different solution (as proposed in [5]) would be to map complex IDL types in an opaque manner and to provide accessor functions for their contents. This would be faster, but adds a number of new functions for constructing and accessing each IDL type.

As an optimization, Combat takes advantage of Tcl's type system and registers its own `CORBA::Any` type. This native representation is unrolled according to the above mapping only upon request, if a script actually accesses its members. However, type conversion requires full unrolling of the type into a string, so accessing one list member requires updating the string representation first and then scanning of that string by the list conversion function. Combat tries to compensate by replacing this list conversion function by one that knows about the new type, but unfortunately, Tcl's implementation of the list functions bypass Tcl's type system and directly call their "known" conversion function,[1] reducing the effect.

3.2 Mapping Stubs

Stubs are client-side procedures that represent a CORBA object. A stub has a type that corresponds to an IDL-declared interface, exports that interface's operations, and incorporates an object reference. When invoked, a stub must marshal the operation's parameters into a GIOP message and call the ORB to transport the request to the target address contained in the object reference.

Combat uses Tcl procedures to represent stubs. For each object reference that a script acquires, either from ORB procedures such as `string_to_object` or as return value from a method invocation, Combat creates a new Tcl procedure, *handle* in Combat terminology, that encapsulates type information and the object reference. The procedure interprets its first argument as operation name and the remaining arguments as the operation's parameters.

IDL also supports *out* parameters that are passed from the server to the client, and *inout* parameters that are passed in both directions. For these, Tcl's usual call-by-name handling is used, i.e. the script passes a variable name that the stub accesses via `upvar`.

[1] `lindex` et al call SetListFromAny() instead of Tcl_ConvertToType().

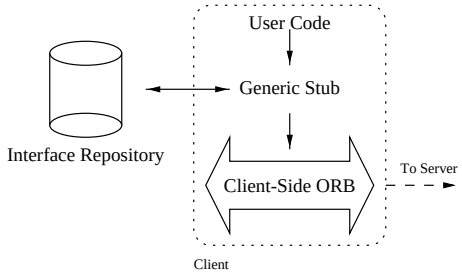

Figure 3: The generic stub needs access to an Interface Repository.

```
set calc [corba::string_to_object \
    corbaloc::rose:1234/Calculator]
set res [$calc calculate 1+2*3+4]
puts "$res"
```

Figure 4: Example of using a stub

An obvious problem is that the procedure receives weakly typed arguments from the Tcl script but must generate a strongly typed GIOP request, so the handle requires type information and must verify each argument's value against the expected type. In C++, this problem is solved by using type-specific *static* stubs for each interface, which have the necessary information hardcoded by the IDL compiler.

This approach was deemed unsuitable for the Tcl mapping, which should be dynamic rather than requiring compile-time information – else one major scripting language advantage would be lost.

Combat therefore uses a *dynamic* stub procedure that reads an object reference's type id at runtime and accesses an Interface Repository, a standard CORBA service containing type information, to verify an operation's name and signature, as shown in figure 3. Figure 4 shows the usage of a stub.

Left to the user is the "feeding" of the Interface Repository, an ORB-dependent procedure that is similar to running the IDL compiler. For ease of operation, Combat includes its own "IDL to Tcl compiler" that generates stringified type information from an IDL file, and allows this information to be loaded into a local Interface Repository, so that no external process is needed.

A problem is that there is no public interface to read the type id from an object reference. Combat's glue code knows some hacks to extract it when built upon "known" ORBs, but this fails elsewhere. Fur-

thermore, object references are not required to hold a valid type id.

The only CORBA-compliant way to receive an object's type information is to send a GIOP request[2] asking for interface information, but that requires setting up an Interface Repository at the server's end. In a worst-case scenario, where the object reference does not contain useful information, and where the server is not connected to an Interface Repository, the client is out of luck and must use a Combat-specific hack to assign type information.[3]

On the Tcl end, the handle's implementation is faced with problems, too. On one hand, a handle is a procedure, but on the other hand, it is data, since it encapsulates an object reference, and therefore blurs the distinction Tcl's type system has between data and procedures.

In particular, Tcl garbage-collects data that is not referenced by any variable, so when a local variable goes out of scope because the procedure is left, the data is automatically released. Such a mechanism does not exist for procedures: if a procedure is not referenced anywhere, is is not cleaned up.

However, garbage collection is essential for object references. Because object references can be contained in parts of a complex value that the user is not interested in (e.g. as by-product of a structure or any value), they are hard to keep track of. Requiring the user to explicitly free handles would be non-intuitive.

Tcl is not really prepared for such coexistence of value and procedure. When a handle is invoked, Tcl normally tries to compile the data and then replaces the original information with the "compiled" one, but in this replacement, the association between procedure and its data is lost. To prevent this behavior, Combat replaces handling for Tcl's internal `cmdName` type and adds its own hooks, a technology pioneered by TclBlend [10], which manages reference-counted Java code objects.

But this relies on internal Tcl interfaces that are subject to change, and even as is, the approach is not perfect. For example, if a complex value that contains an object reference is accessed as a list, the resulting list conversion updates the value's string representation and frees Combat's handle representation, so a workaround was added to completely unwind any type that contains an object reference rather than using the `CORBA::Any` type. Over the lifetime of Combat, several hacks and workarounds have accumulated in the code.

Therefore, Combat is desperate for a solution such as Feather's *command objects*, which are a framework for reference-counted procedures. But as described in [1], Feather currently uses similar hooks to those described here and does not offer any improvement in behavior.

3.3 Mapping Skeletons

On the server side, skeletons exist to receive incoming requests from the ORB, to unmarshal parameters and to perform the upcall into user code. The user must therefore be able to register custom code with the skeleton.

A server process can contain many instances of the same or of different skeletons, each corresponding to a CORBA object and called *servants*. Servants must be registered with an Object Adapter, which serves as a mediator between the ORB and the implementation, providing an interface for controlling the flow of requests and their direction to the "right" servant. As part of its registration with an Object Adapter, or *activation*, a servant is assigned its "identity" in the form of an object reference. Combat fully exposes the Portable Object Adapter's interface [8] at the Tcl level.

The requirements of a skeleton – extensibility with user code, identity, and implementation inheritance to support IDL's interface inheritance – are the same as for an object. Therefore, skeletons are mapped to objects.

Rather than re-inventing the wheel, the popular [INCR TCL] extension [2] is used. In order to implement a servant, the user writes an [INCR TCL] class that inherits the skeleton class and implements the methods according to the IDL description.

An [INCR TCL] object, i.e. an instance of that class, is a Tcl servant that can be registered with an object adapter. It is obvious that the user can easily create multiple instances of a class and register them separately to create more than one object.

Just as Combat uses a generic stub on the client side, it also uses a generic skeleton for the `PortableServer::ServantBase` base class. As on the client side, the generic skeleton accesses the Interface Repository to find the necessary type information in order to unmarshal requests. Again, the server side does not need static but only run-time type information. For this to work, the implementation must give a hint which interface a servant implements by overloading the special `_Interface` method to return the interface's type id. Figure 5 shows an example of

[2] Using the `_interface` pseudo operation, which returns a pointer into an Interface Repository.

[3] If an `_is_a` pseudo operation succeeds, Combat uses that knowledge to update its type information for the object.

```
class Calculator {
  inherit PortableServer::Servant
  public method _Interface {} {
    return "IDL:Calculator:1.0"
  }
  public method calculate {expr} {
    return [expr $expr]
  }
}
```

Figure 5: Servant implementation example

a servant implementation, but not the necessary initialization and registration code.

A different approach that has been proposed [5] is to implement servants as Tcl namespaces. This would eliminate the dependency on [INCR TCL], but then, multiple instances of the same servant would have to cope with identity problems themselves, having to put their individual state into an array indexed by a to-be-defined identifier key. For an [INCR TCL] object, this is a built-in capability. Also, implementation inheritance is impossible with namespaces.

On the other side, [INCR TCL] is not perfect for two reasons. One, its objects are not garbage-collected, and two, it does not yet support diamond inheritance.[4] Since each implementation class must inherit the generic skeleton as its base, multiple inheritance on the IDL level would result in diamond inheritance at the [INCR TCL] level. Implementing interfaces that use multiple inheritance is possible using different inheritance techniques or delegation, but not with intuitive implementation inheritance.

3.4 ORB Interface

Not really a part of the language mapping is the ORB interface. The ORB is basically a layer above the network but below the stubs and skeletons and for the most part invisible, but it does provide a couple of interfaces on its own, such as the conversion between stringified object references handles, and methods to bootstrap initial object references, such as a reference to the Naming Service or to the Root POA.

In contrast to the CORBA specification, Combat does not use ORB objects, a design that would allow multiple ORBs to exist in an interpreter, but an ORB singleton that can be explicitly initialized with `corba::init` or implicitly by using any other ORB method. It might be sensible to change this approach in the future, as having an ORB object is a more flexible design, as it does not require introducing a new Tcl command for each ORB method.

Standard CORBA services, such as the Naming or Event Service, are specified in IDL and are therefore subject to the mappings as described above, so they can be used as long as their interface information has been loaded into an Interface Repository.

The Combat mapping also includes convenience commands to deal with exceptions. Basically, CORBA exceptions are mapped to Tcl errors and can be handled with the native `catch` command. `corba::throw` is, by name, more descriptive than `error` but otherwise identical, and `corba::try` adds Java-style exception handling based on the exception's repository id (see also 3.1). It might be reasoned that these commands are redundant, yet they make code, especially for error handling, much more readable.

Another feature allows for asynchronous requests that are processed from within Tcl's event loop, using the `-async` or `-callback` flag upon a method invocation. The script then receives a *handle* that can be passed to `corba::request` to check if the operation has finished, and to retrieve the result. Asynchrony is important, for example, to maintain a GUI's responsiveness.

4 Discussion

The previous section has outlined the Tcl language mapping used in the Combat project. Several design choices have been made that are subject to discussion, as already mentioned in the text above.

1. Mapping of complex IDL types to native Tcl types (string/list) versus mapping them to opaque values with accessor functions.
2. Usage of generic stubs/skeletons with run-time IFR access versus static stubs/skeletons with hardcoded type information, generated by an IDL compiler.
3. Garbage-collection of handles versus requiring the user to delete them.
4. Using [INCR TCL] servants versus a namespace-based solution.

The first item is a choice between intuitive usability and speed, which ought to be resolved favoring the former. By using generic stubs, CORBA scripts

[4] The case where a base class is inherited more than once via different inheritance paths.

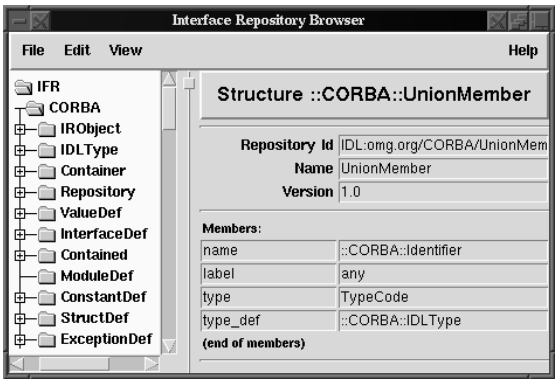

Figure 6: Interface Repository Browser written in Tcl using Combat

remain fully dynamic and do not require compile-time knowledge. However, as e-mail discussion has shown, other developers might set other priorities.

There is an open conflict between the third and fourth issue, because it is inconsequential to garbage-collect handles, but to not garbage-collect servants, a limitation imposed by [INCR TCL]. The choice of [INCR TCL] also seems poor because of its failure to allow diamond inheritance, which is available in IDL. But [INCR TCL] provides exactly what is needed in a servant implementation. Therefore, the author hopes that garbage collection and diamond inheritance will be implemented in future versions of [INCR TCL]. Until then, they remain a nuisance.

A further question is whether introspection features are necessary, such as a list of active handles, their types etc. The only introspection currently available in Combat is `corba::type` to retrieve type codes from the Interface Repository and to type-check values against them.

One major point in discussing a language mapping should be its implementability. Combat serves as proof-of-concept that the mapping presented here in fact works, and that it works well. But then, Combat is a glue package written in C++ and has access to features not available in Tcl. A requirement on a CORBA language mapping for Tcl should be that it can also be implemented in pure Tcl.

There is enough reason to request a pure-Tcl ORB. While it would almost certainly be slower than a glue package, which does all communication in a compiled language, it would not depend on a third party package, and it would not require compilation as Combat does. This is attractive not only in heterogeneous environments – it has proven much harder to compile the C++ code of the ORB and Combat than it is to compile Tcl, – but also in restricted environments such as a browser's sandbox, where compiled code is not acceptable.

With present Tcl, implementing the presented language mapping is only possible to a limited extent, because of the garbage collection issues – here, the C++ glue code uses hooks not accessible in Tcl.

But this is, in fact, the only limitation preventing a pure-Tcl ORB. As another proof of concept, the author has written an experimental ORB in Tcl (utilizing [INCR TCL]). So far, only the client side is implemented with some shortcomings on error handling, but it is indeed source compatible to Combat except for the non-garbage collected handles and passes its client-side tests.

The demand for a pure-Tcl ORB favors the choice of mapping IDL types to native Tcl types over opaque values with accessor functions, because if these accessor functions were to be implemented in Tcl themselves, they too would need plain Tcl input data to manipulate.

As a side note, a pure-Tcl ORB needs facilities for static stubs and skeletons as well as for the default generic ones – else, it would be impossible to access or implement an Interface Repository.

5 Conclusion

The CORBA language mapping presented in this paper is a sensible extension to the Tcl language. In intuitiveness and useability, it wins over past suggestions as in [5] or over other implementations as TclDii[5], tcliop[6] and Torb[7]. All three are client-side solutions only, and use a "dumb" stub that has neither static nor run-time type information, so that the user must pass type information for each invocation.

Before the Combat mapping can be accepted as standard, the mentioned issues must be resolved, some of which depend on future developments in the Tcl core. If reference-counted procedures, garbage-collected [INCR TCL] objects and diamond inheritance of [INCR TCL] classes are not implemented, or if support for the third-party extension [INCR TCL] was dropped, the mapping would need to be adjusted; potential alternatives have been mentioned.

Recent discussions of adding an object package to the Tcl core and the work on the Feather package give reason to believe that these issues will be dealt with in a future Tcl version.

Objects by Value, introduced by CORBA 2.3, have not been mentioned by this document and are

[5] `http://www.cerc.wvu.edu/iss/TclDii.htm`
[6] `http://leo.cs.uiuc.edu/~galmasi/`
[7] `http://pythontech.co.uk/torb/`

yet unhandled in Combat. Their mapping to Tcl needs further discussion.

Another reason for concern is compatibility between different packages implementing the same mapping, or in other words, the portability of Tcl CORBA scripts. It is therefore suggested to introduce explicit versioning. Combat provides the `corba::feature` command to probe for various features of the language mapping that might be unimplemented by other packages (e.g. `core` for basic method invocations, `async` for the asynchrony feature, and `poa` for the POA-based server-side mapping). By assigning version numbers to each feature and the usual Tcl policy that minor updates (e.g. from 1.1 to 1.3) are compatible, scripts will be able to check if their requirements are supported. Commands that are specific to a package or to an ORB must not be placed in the "shared" standard `corba` namespace (e.g. `combat::ir`, `mico::bind`).

References

[1] Paul Duffin, *Feather: Teaching Tcl objects to fly*. Proceedings of the 7th USENIX Tcl/Tk Conference, February 2000.

[2] Michael J. McLennan, *Object-Oriented Programming with [incr Tcl]*. `http://www.tcltk.com/itcl/`

[3] Object Management Group, *CORBA Language Mapping Documentation Index*. `http://www.omg.org/library/clangindx.html`

[4] Object Management Group, *CORBA/IIOP*, `http://www.omg.org/corba/cichpter.html`

[5] Object Management Group, *CORBA Component Scripting Revised Joint Submission*. `ftp://ftp.omg.org/pub/docs/orbos/98-07-02.pdf`

[6] Lutz Prechelt, *An Empirical Comparison of C, C++, Java, Perl, Python, Rexx and Tcl for a Search/String-Processing Program*. `http://wwwipd.ira.uka.de/~prechelt/Biblio/#jccpprtTR`

[7] Frank Pilhofer, *Combat*. `http://www.fpx.de/Combat/`

[8] Frank Pilhofer, *Design and Implementation of the Portable Object Adapter*. Sulimma, 1999.

[9] Scriptics, *Tcl Developer Xchange*. `http://dev.scriptics.com/`

[10] Scriptics, *Tcl Java Integration*. `http://dev.scriptics.com/software/java/`