CLANGER : an interpreted systems programming language

Timothy Roscoe
Computer Laboratory,
University of Cambridge.
(tr@cl.cam.ac.uk)

October 14, 1994

Abstract

CLANGER is a powerful, yet simple, command language for the Nemesis operating system. It uses runtime type information to interface directly with operating system components. CLANGER is a combination of command-line interpreter, scripting language, debugger and prototyping tool. This paper describes why such a language is possible, how it is being implemented, and outlines the language as it currently stands.

1 Background

Nemesis is the operating system being developed as part of the Pegasus ESPRIT Project [3]. It is based on a Quality of Service (QOS) paradigm for resource management which locates as much operating system functionality as possible in the application, thereby reducing QOS crosstalk between application domains.

This is achieved using a structure and linkage mechanism based on the notion of *interfaces*, described in [8]. An interface is the point at which a service is offered. It is implemented as a *closure*, i.e. a pair of pointers: one to a state record opaque to the client, and one to an array of function pointers called a method suite. The number and signatures of the functions in the method suite make up the *type* of the interface, and interface types are defined in an interface definition language (IDL) called MIDDL. Invocation between protection domains and between address spaces is performed through surrogate interfaces.

Interfaces are the basic linkage mechanism in Nemesis. The operating system code is composed of a set of *modules*, whose only externally visible symbols are a small number of closure addresses. To give an idea of the size of modules

(and thus the granularity of linkage units), the system as it stands for Digital Alpha/AXP machines comprises about 25 modules, with sizes ranging from about 100 to 2,500 lines of C source. Most modules are about 300-400 lines long.

At a level above virtual addresses, all computational objects in Nemesis can be named in a way modelled on [9]. An interface of type `Context` provides a flat name space in which textual strings can be bound to objects of arbitrary types. Naming graphs can be built by binding names in one `Context` to other `Contexts`. Name spaces can also be composed in an ordered way in the manner of Plan 9's "union directories" [7]. Interfaces conforming to type `Context` are used extensively within Nemesis.

Nemesis also provides a module called `TypeSystem`. This component of the system encapsulates data structures representing every interface type known to the system together with all operation signatures and concrete types defined within each interface. The information is presented as a collection of interfaces. The Type System is analogous to the Cedar Abstract Machine [10] or the CORBA Interface Repository [4]. In addition, the Type System provides a tagged `Any` type plus associated runtime type checking and narrowing. It is this `Any` type to which textual names are mapped by `Context` interfaces.

These two facilties make CLANGER possible: ubiquitous typing within the operating system (coupled with a means to interrogate the type system at runtime), and a uniform naming model based on pathnames, flat name spaces and runtime typing. Given an object and its type, an instance of the interpreter can perform any legal operation on it in a typesafe manner without any prior knowledge of its structure. Any linkage-level interface in the operating system can potentially be manipulated from the command line. CLANGER can perform any non-time-critical operation that a piece of C code can.

2 Variables in CLANGER

All variable names in CLANGER are Nemesis pathnames, and all values are `Any`s. An instance of the interpreter is associated with a root naming context. This context is used to resolve any variable name. It is also the context in which a new name is bound when a variable is encountered for the first time. New variables are created simply by assigning to them, and their type is inferred from the type of the value they receive. Since variables are nothing more than entries in some naming `Context`, any value in any name space reachable by a pathname from the interpreter's root is a CLANGER variable.

For manipulating instances of MIDDL's concrete types (integers, records, etc.) CLANGER has a full set of operators modelled on C and providing essentially the same functionality. Since each variable carries its own type around

with it as part of its value, any operation on a concrete type can be typechecked at invocation time.

Similarly, a CLANGER value which is a pointer to an interface closure (also known as an *interface reference*) can have any valid method invoked on it (see section 3 below).

This embedding of the command language within the operating system's linkage structures gives the language its expressiveness and power. It also results in a very simple base language. For instance, unlike many embedded interpreters, CLANGER does not provide an associative array type. There are plenty of these in the operating system, indeed the most commonly used one is `Context`, which is the very mechanism used for variable binding anyway. Using associative arrays in CLANGER is just like using any other part of the operating system.

This also permits an interpreter to be embedded in almost any application with next to no effort—certainly without the need for the *ad-hoc* C wrappers required for languages such as Python [11] or Tcl [5]. Furthermore, when a new type is introduced to the system, CLANGER will be able to manipulate its values with no new code whatsoever.

3 Method Invocation

CLANGER's integration with the operating system is achieved through the ability to invoke methods on interfaces. A full method invocation is a statement a bit like this[1]:

```
[ res1, res2, res3 ] <- iref$method[ arg1, arg2, arg3 ];
```

On parsing this statement, the interpreter finds the variable whose name is `iref` and checks that it is an interface reference whose type has an operation called `method`. It then checks that the operation has three arguments, that the types of arguments `arg1`, `arg2` and `arg3` can be narrowed to the appropriate argument types, and that the operation returns at least three results.

The interpreter then actually synthesizes a Nemesis method invocation on the interface and binds to the names `res1`, `res2` and `res3` the first three results, changing the types of `res1` etc. in the process to reflect their new values. Extra results are discarded.

Various shorthands exist. Intermediate results from the call can also be discarded, for example in the statement:

[1] the syntax is somewhat influenced by Cedar and CLU.

```
[ res1,,, res4 ] <- iref$method2[ arg1, arg2, arg3 ];
```

All results (if they exist) can be ignored with a statement like:

```
iref$method3[ arg1, arg2, arg3 ];
```

A single result can be extracted. The value of the expression:

```
iref$method4[ arg1, arg2 ].result3
```

is the result whose formal parameter name is `result3`.

Finally, to invoke a method with no arguments and discard all its results, one can just use the syntax:

```
iref$method4;
```

Exceptions raised by method invocation calls are caught and translated into CLANGER exceptions.

4 Function Definition

One can define functions in CLANGER. A function definition looks like this:

```
def myFunction[ arg1, arg2, arg3 ]
    returns [ res1, res2, res3 ]
    { ... }
```

This statement defines a new function with three formal parameters called `arg1`, `arg2` and `arg3`, and which executes the code inside the curly brackets in an environment almost identical to the caller's except with the actual arguments bound to the names of the formal parameters. The function itself is an Nemesis interface of type `ClangerFunction` and is inserted into the name space when it is defined. Thus it can be passed around and manipulated in much the same way as anything else.

A function thus defined can be invoked in a similar way to a MIDDL method invocation, for example:

```
[ res1, res2, res3 ] <- myFunction[ arg1, arg2, arg3 ];
```

The body of the function is just plain CLANGER code. The function must return by using the `return` statement:

```
return [ 1, 4, "Hello" ];
```

The `return` statement must include the right number of return values. As with normal assignment, the types of values returned will be deduced at run time.

It should be noted that invoking a function call in CLANGER is simply a matter of creating a suitable name space and passing it to the `ClangerFunction` interface. The name space (in effect the scope for the function) can trivially be constructed by creating a new `Context` for the variables, and then creating a union of it and the current root. This operation is sufficiently lightweight to allow functions to be used as syntactic sugar for commonly-used invocations.

A possible future extension would allow the user to define complete implementations of MIDDL interfaces, rather than isolated functions not belonging to any interface. While aesthetically appealing, it is unclear how useful this facility would be.

5 Control Flow

As well as function and method invocation, CLANGER provides the usual rich set of control flow structures:

- `while` loops
- `for` loops
- `if` statements
- `case` statements
- exceptions with `raise`, `try`, `catch`, `catchall` and `finally`.

A block is delimited by curly brackets ("{" and "}") and can be used in place of a statement.

6 Implementation

The basic interpreter for CLANGER is very simple. It consists of a parser which generates a syntax tree, and code which takes a syntax tree and a `Context` interface and executes the code. This is the same mechanism as that used for function invocation. Indeed, the parser returns a Nemesis interface of type `ClangerFunction` just like the `def` statement.

The interface type `ClangerFunction` provides a method called `Execute`, which takes as its sole argument a `Context` interface which provides the name space and execution environment of the call.

7 Examples

The provision of associative arrays in CLANGER has already been mentioned. In addition, all the modules providing language run-time support in Nemesis have interfaces specified in MIDDL and so are usable by CLANGER. Here we give three examples of further facilities to which an interpreter has access:

7.1 Filing Systems

Directory services of filing systems in Nemesis are just another part of the name space. Implementing an `ls` command in CLANGER is a matter of calling the `List` method on a name space, then printing the list of results by calls on the `Print` method of a `Console`. If the notion of a "current working directory" is required, it is simple to have a variable in the root name space called `cwd` or somesuch. Its type is, of course, `Context`.

7.2 Inter-domain communication

Some command languages have extensions to provide inter-process communication via RPC, but require the programmer to create stubs for each RPC interface. The Nemesis inter-domain communication mechanism uses surrogate interfaces and dynamically creates such surrogates at bind time. The interface to the binding mechanism is (naturally) defined in MIDDL. Thus RPC support in CLANGER is already there.

7.3 Threads

Support for concurrency is becoming more widespread in programming languages and operating systems, but shells are some way behind. Access to the thread primitives and synchronisation mechanisms of Nemesis user-level schedulers is via interface calls and is therefore trivially easy from CLANGER, regardless of which particular abstraction is provided by the current domain. Many interfaces in a Nemesis system have concurrency control; those that do not can be protected in the command language.

8 Related Work

CLANGER is, we believe, unique. We do not know of any interpreted language which lives sufficiently low in the system to be able to do anything that can be done in C and still be usable as a general command line interpreter.

Plan 9 [6] allows similar interaction with anything in the machine that implements a filing system. However, it requires that all interfaces in the system are untyped and use a command language. By contrast in CLANGER typed interfaces are down at the linkage level and the command language can use all interfaces. Nemesis interfaces can be (and usually are) much more fine-grained than Plan 9's filing systems.

Software development environments often implement similar functionality: XDE [14], Cedar [10], SmallTalk [2] and Oberon [13] are all examples. However, they tend to be quite specialised. None provide both the power to interactively invoke any system interface together with the tight integration with a true general-purpose operating system supporting many programming languages.

General-purpose scripting languages such as Python [11], Perl [12] and Tcl [5] can only be extended by writing compiled code as a wrapper around some operating system functionality. Furthermore, they are geared towards usage in an environment with poor definition of interfaces and almost no modularity (though this is less true of Python). For our requirements, namely a command line interpreter for a new operating system very different from Unix, implementing CLANGER is much less work than porting one of these languages.

9 Conclusion

CLANGER is the "shell from hell". It allows arbitrary bits of the system to be tweaked from the command line. A CLANGER interpreter running as a privileged Nemesis domain with a full root name space has immense power over every component in the operating system. It can be used to prototype quite low-level system components.

By contrast, by restricting the interfaces available to an interpreter at startup, the same implementation can function as a safe command-line interpreter. Since the interpreter need not understand system interface types *a priori*, it does not need to be changed as new components are added or as interfaces change.

This flexibility extends to embedding the interpreter in applications, by means of specialised name spaces. It fits in naturally with the "build packages, then tools" philosophy of application building [10]: with CLANGER, the easiest way to build a tool is to provide code with well defined linkage-level interfaces in the first place. Once a programmer has done this, the command language really does come for free.

References

[1] BAYER, R., GRAHAM, R. M., AND SEEGMULLER, G., Eds. *Operating Systems: an Advanced Course*, vol. 60 of *LNCS*. Springer-Verlag, 1979.

[2] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[3] LESLIE, I. M., MCAULEY, D. R., AND MULLENDER, S. J. Pegasus—Operating System Support for Distributed Multimedia Systems. *ACM Operating Systems Review 27*, 1 (January 1993), 69–78.

[4] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*, Draft 10th December 1991. OMG Document Number 91.12.1, revision 1.1.

[5] OUSTERHOUT, J. K. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[6] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. The Use of Name Spaces in Plan 9. Tech. rep., AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1992.

[7] PRESOTTO, D., PIKE, R., THOMPSON, K., AND TRICKEY, H. Plan 9, a Distributed System. In *Proceedings of the Spring 1991 EurOpen Conference, Tromsoe* (May 1991), pp. 43–50.

[8] ROSCOE, T. Linkage in the Nemesis Single Address Space Operating System. *ACM Operating Systems Review 28*, 4 (October 1994), 48–55.

[9] SALTZER, J. H. Naming and Binding of Objects. In Bayer et al. [1], ch. 3.A, pp. 100–208.

[10] SWINEHART, D., ZELLWEGER, P., BEACH, R., AND HAGEMANN, R. A Structural View of the Cedar Programming Environment. Tech. Rep. CSL-86-1, Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, June 1986. (published in ACM Transactions on Computing Systems 8(4), October 1986).

[11] VAN ROSSUM, G. *Python Tutorial.* Dept. CST, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, July 1993.

[12] WALL, L. *Perl Man Page.*

[13] WIRTH, N. From Modula To Oberon and The Programming Language Oberon. Tech. rep., Instiut für Informatik Fachgruppe Computer-Systeme, Eidgenössische Technische Hochschule, Zürich, September 1987.

[14] XEROX CORPORATION INFORMATION SYSTEMS DIVISION. *XDE User Guide*, December 1986.