type information from the interfaces and deposit that information in the file. The type information is generated in the form of static data structures connected via pointers.

Extending Tcl

The Dish extension to the Tcl interpreter can be divided into two parts: one that adds predefined Dish variables and commands and a second part that modifies the handling of unknown commands to perform dispatching on Fresco objects. The predefined Dish variables and commands are straightforward to define as Tcl extensions.

The second part is more complicated. Our model called for all object references returned from a Dish command to be potential Dish commands themselves. A straight-forward implementation would define a new Dish command for every object reference known by Dish. This is not only expensive, is it also wasteful, since most object references are not used as commands. Our solution is to not generate Dish commands for object references at all. Instead, we redefined the mechanism for handling unknown commands in Tcl to perform a lookup in our object table that contained all object references. If a match is found, then we perform a dynamic invocation on the object. Otherwise, Dish uses the standard Tcl unknown command handling mechanism.

Conclusions

Dish is a command interpreter based on Tcl that allows users to create and manipulate Fresco objects. Any object reference can form a Dish command for executing an operation. Dish can have a variety of uses, including as an interactive shell, a rapid-prototyping environment, and a testing tool.

The advantage of our approach to scripting is that the connection between the scripting language and the operations implemented in C++ is automatic. New operations that are added to Fresco are immediately accessible to Dish without any effort (other than re-running the interface translator). In addition, scripting code can easily be evolved into C or C++ code because the same operations are also available from a programming language.

Bibliography

[1] P. Calder and M. Linton. Glyphs: Flyweight Objects for User Interfaces. *Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology*, 1990.

[2] M. Linton and C. Price. Building Distributed User Interfaces with Fresco. *Proceedings of the Seventh X Technical Conference*, Boston, Massachusetts, January 1993, pp. 77-87.

[3] J. Ousterhout. Tcl: an Embeddable Command Language. *Proceedings of the 1990 Winter USENIX Technical Conference.*

[4] J. Ousterhout. An X11 Toolkit Based on the Tcl Language. *Proceedings of the 1991 Winter USENIX Technical Conference.*

```
interface RequestObj {
    enum CallStatus {
        initial, ok, unknown_operation,
        bad_argument_count, bad_argument_type
    };
    void set_operation(in string s);
    void add_arg(in string value);
    CallStatus invoke();
    string result_string();
    boolean op_info(out TypeObj::OpInfo op);
};
```

The RequestObj interface is similar to the Request interface defined by CORBA. Our current interface is string-oriented, but we expect to switch to a binary interface in the near future. This approach will move the string-to-binary conversion code from the library into Dish. This change is especially desirable for aggregates, which may have different representations in different scripting languages. It is therefore appropriate to move the language-dependent implementation out of the Fresco library and into the Dish implementation.

Run-time type information

The TypeObj interface describes type information pertaining to each Fresco object. TypeObj provides the name of the type, whether or not it is an interface (only interface objects are entered into the Dish object table), and details on each operation. Operation information include its name, the return type, and a list of parameters. Parameter information include its name, mode, and type. The TypeObj interface is as follows:

```
interface TypeObj {
    enum ParamMode { param_in, param_out, param_inout };
    struct ParamInfo {
        string name;
        ParamMode mode;
        TypeObject type;
    };
    struct OpInfo {
        string name;
        long index;
        TypeObject result;
        sequence<ParamInfo> params;
    };
    string name();
    boolean is_interface();
    boolean op_info(out TypeObj::OpInfo op, in long index);
};
```

Interface translator

The type information provided by TypeObj is generated by Ix, the Fresco interface translator. Ix generates C++ classes corresponding to the interface definitions and also can edit C++ source files to assist with the coding of implementation classes. The type information is generated as part of the filtering process.

The filtering process takes one or more interfaces and modifies a C++ implementation file according to annotations embedded in the file. Some of these annotations instruct Ix to generate

```
g1 request r
puts $r
```

Note that the Dish variable ‘r’ does not have to exist before using it as an out parameter. In the case of an inout parameter, the variable used as the argument must already exist. Dish evaluates the variable just like any other Dish variable and passes the value as the actual argument. Upon return from the operation, the variable is updated with the return value corresponding to the inout parameter. For example, to transform a vector by the transformation object t1, we can use the code:

```
set vector “(1 1 1)”
t1 transform vector
puts $vector
```

Callbacks

The Dish action command allows users to specify callbacks. The syntax for the action command is,

```
action command
```

where *command* is any valid Dish command. The action command returns a Fresco action object, which can be used any place in the interface that accepts an action. For example, the following Dish program creates a push button with the label “Push me” that prints “pushed” whenever the button is clicked:

```
proc push_me {message} {
   puts $message
   flush stdout
}
set button [
   widget_kit push_button [
      figure_kit label [figure_kit default_style] “push me”
   ] [
      action push_me “pushed”
   ]
]
main button [layout_kit margin button 10.0]
```

Implementation

Most of the work implementing Dish has really been in the Fresco request interface that supports dynamic invocation and the Fresco interface translator that generates run-time type information. The design of the manipulation of object references in Tcl also has some associated subtle issues.

Request interface

A request object may be created from any Fresco object. The interface to a request object allows one to set the operation to a string, assign arguments to strings, and finally try to invoke the request. The invoke operation parses the arguments, converts them from strings to binary form, and, if there are no errors, invoke the operation on the object. An error code is returned if the invocation fails. The result of the operation is returned as a string. Dish must also use the op_info operation to determine whether there are any out parameters and if the operation returns a value or not. The IDL for the request interface is as follows:

still refer to addresses in shared memory, so the user can interactively send commands to objects that are on the screen.

Strings

The Fresco interface defines strings as objects and requires the programmer to convert C++ string literals to string objects explicitly. Rather than force a Dish user to perform the same operations, Dish implicitly converts Tcl strings to Fresco strings. For example, the following C++ code implements a window containing the label “hello world”:

```
FigureKit* figure_kit = fresco->figure_kit();
fresco->main(
   nil, figure_kit->label(figure_kit->default_style(), Fresco::string_ref(“hello world”))
);
```

The equivalent Dish program is as follows:

```
main 0 [
   figure_kit label [figure_kit default_style] “hello world”
]
```

Note that in the Dish version the string “hello world” is used directly as an argument to the label operation. The Dish implementation automatically creates the string object to be passed.

Aggregates

Aggregates such as sequences and structs are specified in Dish using pairs of parentheses. Each aggregate is surrounded by a pair of parentheses. Aggregates may be nested and can be assigned to Dish variables. For example, suppose we want to scale a glyph g1 by a factor of 0.5, then we can use the following code:

```
set t [g1 transform]
if { $t != 0 } {
      t scale “(0.5 0.5 0.5)”
}
```

The double quotes are necessary to indicate that the aggregate should be treated as a single argument. The use of an open brace at the end of a line as a line continuation symbol. This example also illustrates the duality of Dish variables: they are evaluated by default in a Dish command (g1), while evaluation is not automatic in a Tcl command (puts).

Out parameters

Certain Fresco operations call for out parameters or inout parameters. Dish expects the name of a Dish variable as the argument for these cases. When an argument is passed to an out parameter, the name in the argument does not have to correspond to an existing Tcl variable. Dish will create a variable and assign the out parameter value to the variable upon return from the operation. For example, to print the requisition of a glyph g1, which describes the desired geometry for the glyph, one can use the code:

For example, consider the following partial interface description for the Fresco Glyph interface:

```
interface Glyph : FrescoObject {
    attribute StyleObj style;
    void append(in Glyph g);
    void prepend(in Glyph g);
    void need_redraw();
}
```

Now suppose that g1 and g2 have been bound to Fresco glyph objects. The following Dish commands are possible:

```
g1 append g2
g1 prepend g2
g1 need_redraw
g2 need_redraw
g1 _set_style [ g2 _get_style ]
```

The brackets (“[]”) denote a nested command. To avoid potential ambiguities, IDL attributes have separate commands for assigning and retrieving a value.

Predefined commands and variables

Fresco defines a set of top-level objects, called “kits,” that provide operations for creating other objects. For example, the drawing kit has operations for creating fonts, colors, and images; the figure kit has operations for creating circles, polygons, and labels; and the widget kit has operations for creating push buttons, menus, and scroll bars.

Dish defines these top-level objects as variables. For example, to create a circle one could give the command

```
figure_kit circle 0 [figure_kit default_style] 0 0 100
```

The arguments to the circle operation are the figure mode (where 0 means filled), the figure style (in this example we retrieve the default style), the origin, and the radius. Similar commands are possible to create other kinds of objects.

Dish defines one more important command, “main,” that corresponds to the main operation defined by the top-level Fresco interface. This operation takes two arguments, a viewer and a glyph, that define the input and output behavior of a window, respectively. Either of these arguments may be zero (but not both). If the viewer is zero, then the window is not interested in events. If the glyph is zero, then the viewer defines both input and output.

The Dish implementation of the main command spawns a new thread, opens a new display connection, maps the window, and executes a Fresco event loop for the display. This approach means that several scripts may be started and controlled interactively.

For example, here is the Dish script to create a circle, put it in a window, and map the window to the default display:

```
main 0 [figure_kit circle 0 [figure_kit default_style] 0 0 100]
```

The current implementation does not create a new Tcl interpreter, though it probably should to avoid concurrency problems. However, even if a new interpreter is created, object references will

Background

Fresco covers a broad range of functionality—buttons, menus, rectangles, white space, text editors, and movie players all can be Fresco objects. For the purposes of Dish, what matters most is that Fresco is specified by IDL interfaces. Connecting Dish to other IDL interfaces would require very minor changes. We therefore do not present here an overview of Fresco. The overall goals and the motivation for using CORBA are described in [2].

Tcl (Tool Command Language) is an interpretative language with strings as the common data type. Tcl commands are generally of the form "command arg1 arg2 ..." where the command and arguments are (or evaluate to) strings. Tcl also provides variables, procedures, and allows recursive evaluation of procedures or commands.

The common usage of the Tcl interpreter is as a library bound into a program. Application-specific commands can be bound to program functions, allowing the user to write scripts that access the application functions.

Tk is a set of widgets that define Tcl commands for the creation, customization, and manipulation of user interface objects. Tk widgets and applications must manually register the Tcl commands, check the parameter types for each command, and perform conversion on the parameter strings to the appropriate program binary form. The Tk button command, for example, must process about 20 argument options.

A second problem with the manual approach used by Tk is that there is no guaranteed correspondence between a command available from a Tcl script and a function available from a programming language. Thus, one may not easily be able to rewrite portions of a script in C or C++ to improve performance.

In contrast, Dish is able to provide the desirable features of Tcl without the problems associated with manual registration and command parsing. The combination of IDL and dynamic invocation gives Dish automatic access to all the Fresco operations, and any operation accessible from Dish is also accessible directly from a programming language using a function call.

Using Dish

When a user starts Dish interactively, a "%" prompt will appear, and the user can start entering Dish commands. A newline normally terminates a command. To end a Dish session, one uses the "exit" command.

Any Fresco object may be used as a Dish command. An object reference may be bound to a Tcl variable by using the Tcl"set" command. This approach allows commands in the following format:

```
    ObjectRef operation argument-list
```

ObjectRef refers to the target object, which is represented as a string (normally the address in ASCII), and *operation* is the name of the operation to invoke as defined in IDL. *Argument-list* is zero or more arguments separated by spaces. Dish currently requires the argument list to match the interface specification both in number and position (one could imagine using the parameter names rather than position).

Dish: A Dynamic Invocation Shell for Fresco

Douglas Pan and Mark Linton

Abstract

Dish is a command interpreter that allows the user to create and manipulate Fresco objects interactively or with a script. Dish uses the Tcl[3] language and interpreter for the basic script control flow and object naming. Because Fresco operations are specified in the CORBA Interface Definition Language (IDL), Dish can automatically translate a Tcl command line into the appropriate call using the CORBA dynamic invocation mechanism. Unlike Tk[4], Dish requires no special registration of commands or explicit argument parsing, and commands in Dish scripts can easily be moved into compiled code.

Introduction

Scripting is a popular form of implementing portions of an interactive application. A command interpreter can either read commands interactively or from a script, and is generally more responsive and easier to use than a compiled environment. Furthermore, scripting languages tend to be simpler than traditional programming languages and therefore appeal to users who do not wish to learn all the details associated with a language such as C or C++.

Scripts are not typically self-contained; some statements may call functions implemented in a traditional programming language for greater efficiency and robustness. Unfortunately, integrating scripts and program functions is often awkward. The program code typically must register the functions to be available from the scripting interface, check the types of parameters, and convert parameters from the scripting representation (typically strings) to the program form (binary).

Fresco is an API for graphical user interfaces in which all operations are specified using the CORBA Interface Definition Language (IDL). To make a scripting interface available automatically, we leverage the dynamic invocation mechanism defined by CORBA. This mechanism allows one to create a request object at run-time and fill in the operation name and parameter list. CORBA also defines a type repository so that an interpreter can check the types of parameters and perform conversions automatically, instead of requiring that each operation do so.

To test the validity of our approach, we have implemented a script interface to Fresco called Dish (Dynamic Invocation SHell) using the Tcl language and interpreter. Dish is a relatively small application (about 1,000 lines of C++ code) that uses dynamic invocation to evaluate commands that create and manipulate Fresco objects.

Douglas Pan is a Ph.D. student at Stanford University and a consultant for Fujitsu, Ltd.
Mark Linton is a Principal Scientist at Silicon Graphics.