Wafe - An X Toolkit Based Frontend for Application Programs in Various Programming Languages

Gustaf Neumann – Wirtschaftsuniversität Wien
Stefan Nusser – Wirtschaftsuniversität Wien

ABSTRACT

Wafe provides a flexible and easy to use interface to the *X Toolkit* (Xt) and the *Athena* widget set (Xaw) using the embeddable command language *Tcl* [1]. It allows access to Xt's functionality from all compiler and interpreter languages, provided that they can communicate over *stdout* and *stdin* via unbuffered I/O. A typical Wafe application consists of a frontend process and an application program, which is executed as a child process of the frontend. Wafe provides a relatively high level interface to the X Toolkit and widget programming, where the user interface can be interactively developed without any need to program in C. Wafe can be used as a rapid prototyping tool and allows easier migration from existing ASCII based programs to X Window applications.

Introduction

When we started to work on the Wafe project in Summer 91 we had the need to provide decent user interfaces for applications in various (mostly interpreted) programming languages. As a matter of fact, at this time most of our applications were running with ASCII based user interfaces under terminal emulators like xterm - which is a practical but suboptimal way of using the graphical user interface of our equipment, which consists mostly of X Window based workstations. We found out that for most (if not even all) of our application programs a small set of X Toolkit commands and the Athena widget library with its programmatic interface was completely sufficient to provide easy to use graphical interfaces.

On the one hand, it seemed impractical to implement widget functionality in all different programming languages used for our applications, on the other hand we did not even consider to port our existing programs to C. Therefore we chose a frontend approach where all widget functionality is incorporated in one separate program. We called our frontend Wafe, standing for *W*idget[*A*thena]*F*ront*E*nd. Wafe was implemented using the embeddable command language Tcl [1], which was augmented with widget specific facilities. Tcl is an interpretative language using strings as the only data type and provides a collection of built-in utility commands as well as user defineable subroutines.

Given the situation described so far, we decided after a thorough analysis of the existing products to implement our own solution using the following design goals:

- Our frontend approach must be able to collaborate with a broad variety of programming languages, using a handy communication mechanism. This implies that we cannot presume that the backend application will support certain libraries (eg. sockets or pipes), which are actually not available under some of the programming languages used for the examples presented in the last section.
- To support smoothly the different stages of the developing and prototyping process, we want our frontend application to provide three different modes of operation: There is an *interactive mode*, where Wafe can be used as a single process reading commands from standard input, which are interactively interpeted. The user sees how the widget tree is built and modified step by step. The interactive mode offers the possibility to examine the effects of different commands or to easily compare different approaches to accomplish a certain task.
  Furthermore, our frontend has to support the possibility to execute command files (*file mode*). The file mode offers two main usages: First, this mode can be used to provide simple user interfaces just by writing scripts in the Tcl language, where Tcl's built-in commands or the commands provided by Wafe can be used. Typically such a script will start with the `#!` magic supported by most of the shells. This script can also be used later as a frontend. The user interface (the frontend) can be developed mostly independent from the application program (the backend).
  Finally, Wafe provides the so-called *frontend mode* which uses interprocess communication facilities as described in the sections below to support the separation between the backend application and the frontend process.
- Another requirement for the application development was the extensibility of the chosen widget set. This made us choose the X Toolkit as the basis for our program, granting access to the broad range of

commercial or freely available widgets based upon this toolkit.

• Finally, as mentioned above, we want to use Wafe as a prototyping tool as well, for developing and testing applications which will be implemented finally in another programming language (mostly C). This requires the incorporation of a widely available widget set. We chose the Athena widgets as the basis for our project, since they are part of the MIT standard distribution of the X Window System. Accepting that they do not offer a very exciting appearance, a version supporting the commercial OSF/Motif widget set is under development (at the time of this writing).

The first section starts with a short comparison of Wafe and Tk [2] which was one of the ancestors (motivations) of the Wafe project. The following section presents an overview of the components, followed by a summary of the design principles and basic features of Wafe. After that we will discuss how Wafe can be used as a frontend for application programs in arbitrary programming languages. This section contains a programming example in Perl [3]. The summary of our experiences and an availability note end this paper.

Comparison between Wafe and Tk

The regular USENIX conference visitor who is confronted with the terms ''Tcl'' and ''user interface'' will associate immediately John K. Ousterhout's work on Tcl [1] and Tk [2]. Therefore we want to give a short comparison of Wafe and Ousterhout's work before we concentrate on the details of Wafe.

In some of its components Wafe looks similar to the Tk toolkit [2]. Tk comes with a Tcl shell (*wish*) which allows to read in command sequences in Tcl from a file. Wafe's equivalent is the file mode mentioned above.

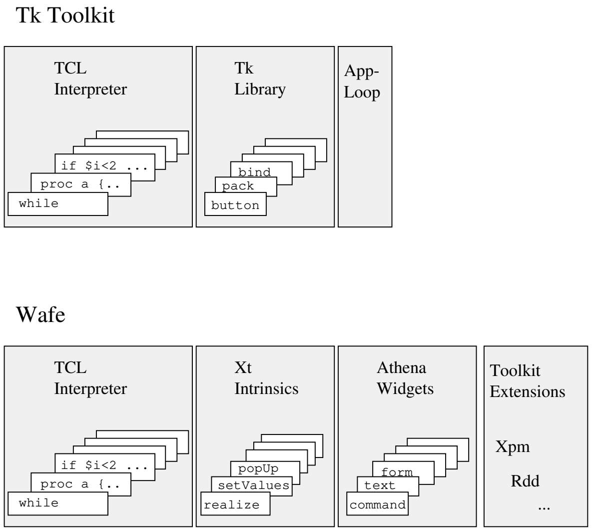

Figure 1: Tk and Wafe components

The Tk intrinsics and Tk widgets have been implemented by John Ousterhout since 1989; Tk offers three dimensional appearance of its widgets, its implementation compares favorably with the Motif counterparts in terms of size (see [2]).

Wafe is - on the other hand - based on the standard X11R5 Xt Intrinsics [4] and the Athena widget set [5] (see Figure 1). As a consequence it was easy to extend Wafe with other Xt based widgets, widget sets or libraries such as Xpm [6] or for example a drag and drop library (Rdd [11]). A user interface designer can use the standard X11R5 literature (knowledge, support) in order to develop Wafe applications. It is straightforward to replace the Athena widgets by any other Xt based widget set (such as Motif) or to augment Wafe with special purpose widgets. The current Wafe distribution contains support for the Plotter widget set (which supports bar graphs and line graphs [12]) and the *XmGraph* widget (a graph layout widget for OSF/Motif used in Figure 2 [13]). Kaleb Keithley's three dimensional Athena widget library (Xaw3d) [10] can be used simply by relinking Wafe.

In order to write larger applications in practically arbitrary languages, Wafe provides its frontend mode. Current versions of Tcl and Tk do not provide any comparable facility.

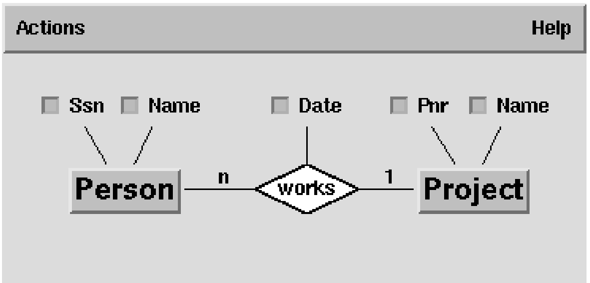

Figure 2: Sample Wafe application using the *XmGraph* widget based on OSF/Motif

The Components of Wafe

This chapter is intended to present the most important implementation issues and to explain some design principles which build the basis of all user level commands.

Wafe's structure can be described globally by the following formula:

$$\textit{Wafe} = \textit{Tcl} + (\textit{Intrinsics} + \textit{Widgets} + \textit{Converters} + \textit{Ext}) + (\textit{Memory Management} + \textit{Communication})$$

The three main components of this formula are described in the following sections.

Tcl

Wafe uses the embeddable command language Tcl (first part of the formula) as a host language and extends Tcl's basic programming capabilities with additional X Toolkit and widget specific commands.

Tcl provides a parsing mechanism as well as a procedural framework for the generic Wafe commands and offers advantages to users already familiar with other Tcl based tools.

In our point of view Tcl offers the following advantages as a host language:

- Tcl has a simple syntax: Every command is simply a list of words.
- Tcl is highly extensible, since it has a simple and well documented interface to C where each argument is only a string.
- Tcl has a clean memory management, where it is possible to specify whether Tcl should copy variables or where special routines can be specified to free memory.
- Tcl uses only one type of argument - the string. Since the string representation is also used to specify information in resource files, the standard Xt converters can be used to convert from string to the variety of Xt or widget specific data types.

Of course, the use of Tcl imposes some limitations too (see also [2]):

- It is not suitable for more complex programs, since it was designed to be a command language.
- The string representation of all data types is a disadvantage, when repetitious calculations have to be made in Tcl.

Besides *Tcl*, two different groups of components can be distinguished in the *Wafe* formula above. For the following part we assume a certain familiarity with the X Window programming tools, which are extensively described in [7] or [8].

X Toolkit Specific Components

The second unit in the above formula represents all functions and commands actually implementing the programmatic interface to the X Window System. These commands provide access to the functionality of the X Toolkit, which comprises all commands necessary to manage a widget's life cycle, the selection mechanism and some information retrieving functions as well as the basic widget classes. This functionality is commonly called the X Toolkit *Intrinsics*. In most cases, each toolkit function is represented by a corresponding Wafe command.

In addition to the X Toolkit a suitable widget set (*Widgets*) is needed. Examples are the Athena widget set, the OSF/Motif widget set, or the widgets of the Open Look Intrinsic Toolkit (Olit). Every widget set typically has a series specific functions, which are called the ‘‘programmatic interface’’. In general, the whole functionality of the programmatic interface is accessible via corresponding Wafe commands.

Converters are an Intrinsics based concept to set and read resources. The X Toolkit provides mechanisms which allow to register additional application specific converters. Wafe registers several converters to ease the handling of certain resources or to implement some additional functionality. This will be discussed in detail later.

Internals

The third unit of the formula above comprises the necessary internals which put Wafe to work: Since the creation of a widget always implies the dynamic allocation of memory for the associated resources, *Memory Management* is a topic of special importance. Wafe has its own memory management: every time a string resource, a callback - or other objects larger than one word - are updated, the old value is freed. If a widget is destroyed the associated resources in Wafe's memory are disposed too.

The other part of the internals deals mainly with the *Communication* mechanism and its different options, which are described in detail later when Wafe's frontend mode is discussed. Note that most of these internals are hidden from the user, although some commands offer the possibility to extend and customize the communication mechanism.

Design Principles

We tried to present a consistent interface, which is based upon certain design principles. We will present them in the following section.

Transparency

Internals should be hidden from the user as much as possible. Interfaces to the corresponding X Toolkit functions are simplified wherever possible. For example, widgets and even windows are therefore referenced by the widget's name, or as another example, function pointers to certain handlers or callbacks are just executable Tcl string expressions, which are evaluated at the time the handler or callback is invoked.

Naming Conventions

Naming conventions are kept as follows: Wafe commands corresponding to X Toolkit functions (eg. *XtDestroyWidget*) have the same name except that the prefix ‘‘Xt’’, ‘‘Xaw’’ or ‘‘X’’ is stripped and the first letter of the remaining string is translated to lower case (in our example, the resulting command is called `destroyWidget`). The same principle is applied to all commands associated with the Athena widget set. For example *XawFormAllowResize* is called `formAllowResize` in the Wafe framework). It should be noted that the Athena widget set is the primarily supported library. On the contrary, OSF/Motif commands stripped by the rules above result in Wafe commands starting with the letter *m*. The OSF/Motif command *XmCommandAppendValue* is therefore called `mCommandAppendValue` in the Motif version of Wafe.

It should be noted that all Wafe commands are generated automatically from a high level description (for code generation, see below). During the code generation the naming rules above are applied. If one prefers other naming conventions, or does not like the prefix stripping at all, it is fairly easy to change the names. In addition Tcl allows to register the same command under various names.

Following the X Toolkit Programming Philosophy

In order to build a frontend Wafe offers commands which should be explained in the X Toolkit documentation. Widgets are created and configured, then the widgets are realized, and during the run of the application the execution flow is triggered by actions and callbacks.

In general one Xt or widget specific call of a C procedure corresponds to one Wafe command. In certain cases, Wafe provides convenience procedures, which group several commands together and help to hide internals.

The *widget creation commands* are an exception from this rule: Instead of implementing one command to instantiate a widget from a certain class (namely *XtCreateManagedWidget*), Wafe provides a different command for each widget class to create an instance. The names of these commands are derived from the corresponding classes in an analogous manner. To create an instance of the Athena Toggle widget class, the command "`toggle` *Name Father*" is provided. In order to create an OSF/Motif *XmCascadeButton*, the creation command is called `mCascadeButton`, and so on.

Command Line Arguments

When command line arguments are passed to Wafe, it has to be determined, for which part of the application these parameter are relevant. In general, there are three candidates:

- The X Window Toolkit,
- the frontend (Wafe), or
- the application program.

We have choosen the following approach: Command line arguments starting with a double dash (like "`--f`") are always handled by the frontend. The remaining arguments are passed to the X Toolkit (to interpret arguments like "`-display hostname:0`" or "`-xrm` application program, if Wafe runs in the frontend mode.

Argument Style and Value Passing Conventions for Wafe Commands

Wafe's argument style is similar to other Tcl based tools: Arguments are separated by spaces, can be grouped as described in [1] and are all of data type *string*. Xt function calls returning a single value are implemented using the standard Tcl method of return value passing. In C programs Xt functions returning several values receive a pointer to a free memory area, where the return value will be placed. The Wafe counterparts of these functions take a name of a Tcl associative array as an argument (instead of a pointer) and create entries in the associative array corresponding to the C-structure's components. We did not have the intention to implement all components of a structure, since some components are rather meaningless in the Wafe context (for example: a display pointer). If an Xt command is used which returns a structure, it should therefore be checked with the Wafe documentation which members are supported. When a C procedure returns a list of a certain type and its length, we return the number of elements as a function value and provide a variable name for the list.

This principle can be illustrated by the following example. The toolkit function `XtGetResourceList` has the following syntax:

```
void
XtGetResourceList(
      WidgetClass,
      XtResourceList* /*return*/,
      Cardinal*       /*return*/
      );
```

The corresponding Wafe function is named `getResourceList`, accepts two arguments (`widget` and `varName`) and returns as function value the number of elements in the list named in the second argument. Since Wafe applications do not deal with the structure *WidgetClass*, we use a widget instance to refer to the class. The string containing the widget name is used to refer to a widget instance and is passed to `getResourceList` as first argument. Therefore `widget` references a previously created widget by its name. The second argument `varName` is the name of the Tcl variable to be created. Let's consider the following example, which can be issued interactively by using Wafe in interactive mode:

```
label l topLevel
echo [getResourceList l retVal]
echo Resources: $retVal
```

The first command creates an instance of the Athena label widget class named `l` as child of the `topLevel` widget (which is a top level shell automatically created in every Wafe program). When the second command is executed, the output of the command between square brackets is printed on standard output. Thus, if the command is executed, the number of resources available for the *Label* widget class is printed, which is 42 using the X11R5 Xaw3d libraries. In addition, a list of the *Label* widget's resources is passed to a variable named in the second argument. A Tcl variable will be created containing the desired information as a Tcl list structure. In the code example above the third command prints the contents of this variable. (Note that the dollar sign is used for variable substitution in Tcl). The output of the third command looks as follows:

```
Resources: destroyCallback
ancestorSensitive x y width
height borderWidth sensitive
screen depth colormap background
(...)
```

The *XtResource* structure actually contains more members than just the resource's name (such as the default value or the data type for example), but Wafe currently supports only the resource names.

Code generation

As noted above, all Tcl commands provided by Wafe are generated automatically from a high level description. The code generation is performed by a Perl program, which takes as argument the specification file and outputs the necessary C code for conversion, argument passing, error messages, storage management, interpretation of percent codes for callbacks (see section about callbacks below) and registrations of commands. In addition the code generator outputs TeX source for the short reference guide. The main advantages of the code generator are that it (a) provides consistency in documentation and interface code, (b) eases changes that effect code changes on various different places, and (c) makes Wafe easily extensible.

The following example of the specification suffices to provide a `mCascadeButton` command in Wafe.

```
~widgetClass
XmCascadeButton
     include <Xm/CascadeB.h>
```

The specification below creates the Wafe command `mCascadeButtonHighlight` with two input arguments. The command can be used to toggle the state of a OSF/Motif cascade button widget.

```
void
XmCascadeButtonHighlight
     in: Widget
     in: Boolean
```

The Wafe source is currently about 13000 lines of C code. About 60% of the code is generated automatically from specifications like the two examples above. For widget sets with highly regular patterns in their man pages (like OSF/Motif), it is even possible to derive a first draft version of the specification directly from the manual pages.

Basic Features of Wafe

This section presents an overview of Wafe's functionality. In order to obtain a complete documentation of all available Wafe commands refer to the Wafe distribution referenced at the end of this paper.

Creating Widgets

The creation of a widget is certainly the most fundamental task to accomplish with Wafe. It can easily be done using the widget creation commands presented in the last section. Note that these commands correspond to the configuration of a specific Wafe binary - if you choose to install the OSF/Motif version, the command to create the Athena text widget, `asciiText`, won't be available, since in the current version it is not possible to mix Athena and OSF/Motif widgets and converters freely.

All widget creation commands take - after the widget's and the parent's name - any number of attribute-value pairs as additional arguments, which are used to set resources at the widget's creation time.

Consider the following example creating an instance of the OSF/Motif *XmPushButton* widget class under the top level shell:

```
mPushButton pressMe topLevel
```

This command will create a managed *XmPushButton* widget named *pressMe* as a child of the application's top level shell widget. The creation of unmanaged widgets is easily accomplished by an optional argument.

When a Wafe application wants to display widgets on multiple X servers it can create several application shells where the display is specified instead of the father widget.

```
applicationShell top2 dec4:0
```

The children widgets under top2 will be mapped to the specified display.

Setting and Retrieving Resource Values

Resource Values are public variables of a widget instance, which are intended to be set by the programmer or to be configured by the user. Wafe provides several ways to set resource values:

- Using a resource description file, which is evaluated at startup time of the application.
- Using the command `mergeResources`.
- With arguments to the widget creation commands at creation time.
- With the command `setValues` after a widget's creation.

Note that the order of these possibilities to set resource values above corresponds to their precedence. All of the commands will be described in the next paragraphs.

The resource file mechanism

The resource file mechanism, extensively documented in [7], can be used by any Wafe application. Note that Wafe provides some additional converter procedures for the types Pixmap, Callback or XmString. Such resources can be set in the current version of Wafe only during widget creation or via `setValues`.

The mergeResources command

An extension to the resource file mechanism is provided by the Wafe command `mergeResources`. Whenever a widget is created, the per display database of resource specifications is searched for entries relevant for the new widget instance.

By using `mergeResources` the resource database can be extended with additional specifications. The specified resources can refer to widget classes as well as to instances. For short Wafe scripts it is often preferable to have the code as well as the resource specifications together in one file.

This possibility is illustrated by the following example, which could be part of a Wafe script as well as part of a front end application.

```
mergeResources \
    *Font       fixed \
    *foreground blue \
    *background red
 (...)
label hello topLevel
```

The resource specifications are used as if they were specified in an application defaults file. The label widget created afterwards will use the three values specified, but they apply as well to every other widget created in this application. The `mergeResources` command can be used at arbitrary places in a Wafe application.

Arguments to widget creation commands

All widget creation commands take any number of additional attribute-value pairs as arguments. Since Wafe uses the standard Xt resource file mechanism in order to convert the specified values to their corresponding data types, you can as well use the features provided by the additional type converters, which will be described below.

Consider the following example, which creates an instance of the Athena `Label` widget class using red background and blue foreground colors.

```
label label1 topLevel \
    background red \
    foreground blue
```

As already explained these specifications override any settings in resource files or settings made with `mergeResources` and therefore reduce the configurability of the application via resource file.

The setValues command

The `setValues` command is used to change a resource value after the widget has been created. Note that there are some resources which cannot be set after creation time or after the widget is realized. For detailed information refer to the documentation of the specific widget class.

In order to change the resource value of `background` and `label` of the previously created widget `label1` the following statement can be used

```
setValues label1 \
    background tomato \
    label "Hi Man"
```

For convenience the command `setValues` is registered as well under the name `sV`.

The Wafe command analogous to `sV` to retrieve values from resources is the command `getValue` (or `gV` for short).

```
echo [gV label1 label]
```

The Wafe command above outputs the content of the label resource of the widget `label1`.

Callbacks and Actions

The X Toolkit provides two mechanisms to link widgets to application code: Callbacks and Actions. Since Wafe's interface is slightly different from the original Xt functions it is described in detail in this section.

Callbacks

Callbacks are used to invoke a function whenever certain predefined requirements are satisfied. Callbacks are defined by the widget itself, which declares a callback resource. An application programmer cannot configure a new callback, she/he can just decide whether to use the callbacks provided by the widget class or not. Actions are more flexible to use since they can be bound to an arbitrary event but they require a more complicated handling.

The most common use of callbacks in Wafe applications will be of the form

```
command hello topLevel \
    callback "echo hello world"
```

where the callback procedure is set via resources. This converter will be discussed in the next section. Using the converter an arbitrary Wafe command can be provided.

In addition to this facility special purpose callback functions offered by the X Toolkit can be used as well. These predefined callback functions can be bound to a widget's callback resource by using the Wafe command `callback`. The different predefined functions available are summarized in the table below:

Predefined Callbacks	
Type	**Description**
none	realize shell, grab none
exclusive	realize shell, grab exclusive
nonexclusive	realize shell, grab nonexclusive
popdown	unrealize shell
position	position shell
positionCursor	position shell under pointer

All of these callback functions concern the handling of popup shells, which are used for menus, dialog boxes and the like. Wafe's access to the predefined callback functions is illustrated by the following code segment for the OSF/Motif version of Wafe, which presumes a previously created *Shell* widget called `popup`.

```
mPushButton b topLevel
callback b armCallback none popup
```

The Motif *PushButton* widget's *armCallback* resource triggers the specified function whenever the button is pressed. In the example the specified predefined function with the name *none* realizes the popup shell `popup` without constraining user events to it.

Actions

Wafe's interface to actions is essentially the command `action`, which is used to override, augment or replace the translation table of a widget with translations specified as arguments. Note that a widget's translation table is actually maintained as a resource called `translations`.

Consider the following example: The Athena *MenuButton* widget provides a simple mean to realize and place a popup shell on a button press. To modify the translations of this widget in order to let the menu pop up whenever the pointer enters the button the following Wafe command can be used:

```
menuButton mb topLevel
action mb override \
   "<EnterWindow>: PopupMenu()"
```

The first command creates an instance of the *MenuButton* widget named `mb`. The second command binds the enter window event to the action `PopupMenu`, which is provided by the *MenuButton* widget class. `PopupMenu` is a built-in action of the X Toolkit.

In addition to the built-in actions provided by Xt and the used widget sets, Wafe provides the possibility to bind the execution of an arbitrary Wafe command to an event. Wafe registers a global action `exec` which accepts any Wafe command as argument. When the action is activated, the Wafe command is executed.

One of the big advantages in using actions instead of callbacks is the possibility to access information from the event which triggered the execution. This feature is supported in a restricted fashion by the `exec` action with printf-like percent codes. The event types supported in this way are:

- Button Press, Button Release
- Key Press, Key Release
- Enter Notify, Leave Notify

Since the information passed to an action depends on the type of event that triggered it, only the following combinations of percent codes and event types are valid:

Event Types and Percent Codes of Actions		
Code	**Information**	**Events**
%t	event type	all of the above
%w	widget	all of the above
%b	number of button	BPress, BRelease
%x	x-coordinate	all of the above
%y	y-coordinate	all of the above
%X	x-root-coordinate	all of the above
%Y	y-root-coordinate	all of the above
%a	ascii-character	KPress, KRelease
%k	keycode	KPress, KeyRelease
%s	keysym	KPress, KeyRelease

It is the programmer's responsibility to ensure by a correct binding in the translation table that a percent code substitution occurs only with a valid event type. The `%t` code will expand to `unknown`, if the event is not included in the list above.

Let us consider as an example an Athena *Label* widget. With the following translation, the key-code, character and keysym will be printed any time a key is pressed in the label widget called `xev`.

```
label xev topLevel
action xev override \
   {<KeyPress>: exec(echo %k %a %s)}
```

If the input “`w!`” is typed on the label widget `xev`, Wafe prints the following output to the associated terminal:

```
198 w w
174 Shift_L
192 ! exclam
```

Converter Procedures

Converters are an Xt Intrinsics based concept which is used to implement conversion for the resources of a widget. In Wafe, a converter always converts a string to a certain target data type; the X Toolkit provides easy mechanisms to provide additional converters.

We tried to use converter procedures whenever we decided to extend the standard Xt mechanism. Some of Wafe's additional converter procedures will be described in this section.

The Callback Converter

We have already introduced Wafe's `callback` command in the last section; the callback converter is used to bind the execution of a Wafe command to a widget's callback resource. Since this feature is implemented as a converter, the standard `setValues` command can be used to set the resource, or the resource can be provided in the resource list in a widget creation command.

The following example shows how to provide the callback resource in a widget creation command

```
command quit topLevel \
    callback quit
```

or to set (or to alter) it later using `sV`:

```
command quit topLevel
sV quit callback quit
```

In this example `callback` is the name of the Athena *Command* widget's callback resource and `quit` a simple Wafe command used to terminate an application.

Some widgets pass additional information to certain callback functions. To access this so-called *clientData*, Wafe uses again printf-like percent codes. Note that these percent codes are only interpreted for certain *Callback* resources in certain widget classes.

The complete list of percent codes for each widget class can be found in the Wafe short reference manual. Below is a table of the percent codes for the `callback` resource of the Athena *List* widget class as an example:

Athena List Widget Callback	
Percent Code	**Description**
%w	widget's name
%i	index
%s	active element

The X Toolkit passes the widget pointer referring to the invoking widget to every callback function. This widget pointer is evaluated by using `%w`. Since this information is available for each callback function in Wafe, `%w` can be used in any callback function to obtain the widget's name. The following example shows a statement to set a previously created Athena label widget named `confirmLab` to the selected item of a list widget named `chooseLst`. Selecting an item of a *List* widget activates the specified callback procedure.

```
sV chooseLst callback \
   "sV confirmLab label %s"
```

Opposite to the X Toolkit it is possible in Wafe to obtain the value of a callback resource. The following Wafe script creates a *Form* widget with two *Command* widgets as children. The callback of the second command widget (`c2`) is set to the content of the callback resource of `c1`. When the widget tree is realized and the callback of `c1` is activated, the string "i am c1." is printed; if the callback for `c2` is activated, the output is "i am c2.".

```
#!/usr/bin/X11/wafe --f
form f topLevel
command c1 f \
    callback "echo i am %w."
command c2 f \
    callback [gV c1 callback] \
    fromVert c1
realize
```

The XmString Converter

The Wafe OSF/Motif version provides a converter to *XmString*, which is Motif's compound string data type. A compound string is an extended string format, which additionally contains font information and the string's writing direction. The converter procedure allows to provide compound strings in a user friendly way in a widget creation command or in a `sV` or `gV` command.

Please refer to [9] or any other OSF/Motif book for a complete description of compound strings; the following example using the OSF/Motif XmLabel widget should illustrate the point:

```
#!/usr/bin/X11/mofe --f
mLabel l topLevel \
  fontList \
   "*b&h-lucida-medium-r*14*=ft,\
    *b&h-lucida-bold-r*14*=bft" \
  labelString \
   "I'm^bft bold^ft and^rl strange"
realize
```

The syntax of Wafe's compound string interface is straightforward and similar to TeX's text formatting commands. A special character (we are using "^" instead of TeX's "\") is used for layout commands which are either used to change the font or to change the writing direction. The output of the sample script is shown in Figure 3.

Figure 3: An OSF/Motif widget with compound strings

The Pixmap Converter

The X Window pixmap format (Xpm [6]) is a graphical image file format similar to the standard X11 bitmaps, but it supports colored images and shape masks. Wafe provides an extended String-to-Bitmap converter which checks additionally whether the specified file is in Xpm format, when the attempt to read the file in the standard X bitmap format failed. This converter can be used to set all resources of type *Pixmap*, such as for example the background pixmap of the Athena *Label* widget.

Using Wafe as a Frontend

In our framework a typical Wafe application consists of two parts, the frontend (Wafe) and an application program, which typically run as separate processes. The application program talks to the frontend via stdio. Each output line from the application process starting with a certain prefix character is interpreted as a Wafe (or pure Tcl) command. So an application program can dynamically submit requests to the frontend to build up and modify the graphical user interface; the application can even down load application specific Tcl procedures to the frontend, which can be executed in the frontend without interaction with the application program. At the same time the application program reads from stdin, which is connected to Wafe, and awaits ASCII strings to control its actions.

Starting Applications in Wafe's Frontend Mode

When Wafe is used in the frontend mode, an application program is started as a subprocess of Wafe. After the fork the necessary connections of the I/O channels are established (see Figure 4, left hand side). Note that in interactive mode or in file mode no subprocess is spawned, and Wafe behaves like a shell.

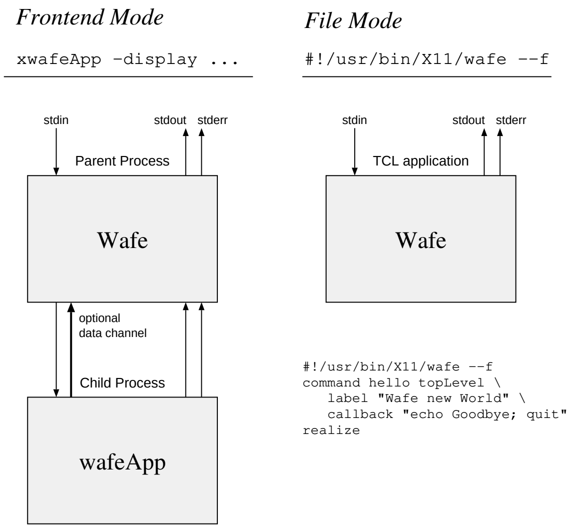

Figure 4: Wafe's Communication Mechanism

The first question, however, was to figure out, what application program should be launched as subprocess. Although Wafe provides a command line option to specify the name of the application program, it is in many cases not convenient to be forced to specify this argument. Therefore we chose the following naming scheme:

Suppose an application program is named `wafeApp` (see Figure 4). If a link like `ln -s wafe xwafeApp` is established and `xwafeApp` is executed, the program `wafeApp` is spawned as a subprocess of `wafe` and connects its stdio channels with the frontend.

Lines written from the application program to *stdout* are read by the Wafe process. If the line received by Wafe starts with a certain character (such as %) Wafe tries to interpret the remainder of the line as a Tcl command. Note that each command issued that way has to fit in a single line (which can be pretty long depending on a preprocessor variable specified at compilation time; the default length is 64KB).

The commands submitted to Wafe can be issued from arbitrary programming languages provided that they are able to write to *stdout* unbuffered (the application program must at least be able to flush the buffer) and to read from stdio. The frontend is programmed by the application program to send back string messages whenever certain events (like button presses, etc.) occur. This way the application program determines the syntax in which Wafe talks back.

Using Wafe's Mass Transfer Mechanism

As indicated above, output lines from the application program starting with a certain prefix character are parsed and interpreted as Wafe commands. Other lines from the application are printed by Wafe to stdout. In some larger applications it is necessary to transfer a bulk of data from the application program to the frontend. In this case it is preferable to establish an additional (optional) data channel (see Figure 4), where no parsing or interpretation is performed. If an application program wants to use this data channel, it has to figure out first, on which file number Wafe is listening. The application program can obtain this information by sending the command

```
echo listening on [getChannel]
```

to Wafe which writes back for example "listening on 5". The data transferred will be stored in a Tcl variable in the frontend. If the application program issues the command

```
setCommunicationVariable \
     C 100000 \
     {sV text type string string $C}
```

the data transferred over the mass channel (5) will be stored in the Tcl variable named `C`. After 100000 bytes are read, the Tcl command specified in the last argument will be executed. In this example it will set the string resource of the Athena *asciiText* widget to the transferred content.

Typical Structure of Application Programs using Wafe as a Frontend

Throughout this section we assume that Wafe is used in frontend mode and an application program is performing some meaningful computations that we do not want to program in Tcl, or that we do not want to bind to Wafe. When an application program is started using Wafe as a frontend we can distinguish three phases (see also Figure 5):

1) Wafe starts the application program as a subprocess.
2) The application program creates and configures the widget tree, submits Tcl procedures and realizes the widget tree.
3) In a read loop the application program accepts commands in the form of ASCII strings from the frontend. The commands are triggered by callbacks or actions.

For some interpretative programming languages it is preferable to send an initial command from the frontend to the application process after the fork to initiate step 2. For instance in Prolog, it is convenient to send a startup goal "`[myapp], widget_tree, read_loop.`" in order to load the application "`myapp`" and to cause Prolog to print the commands necessary for 2 and to continue with 3. For this purpose the resource `InitCom` is provided, which can be specified in a resource file or by using the "`-xrm '*InitCom: ..`" command line option.

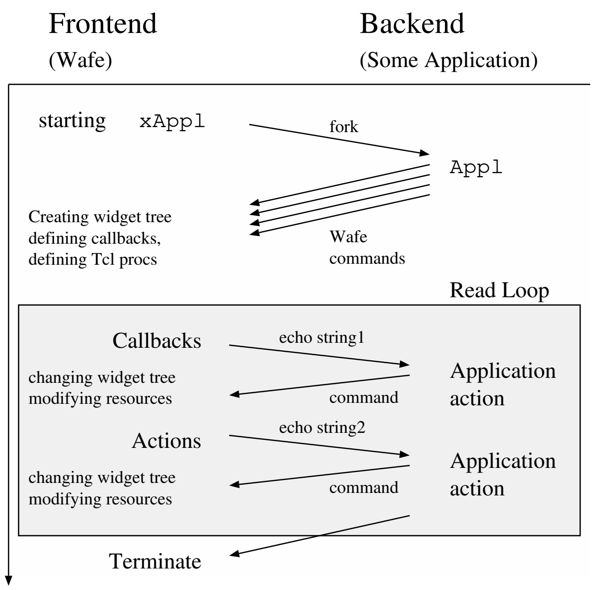

Figure 5: Using Wafe as a Frontend

The following short sample program written in Perl demonstrates steps 2 and 3. The program computes prime factors for integers typed into an Athena asciiText widget.

```
#!/usr/local/bin/perl
$|=1; # set output unbuffered

# build widget tree
print
  "%form top topLevel\n"
  ."%asciiText input top editType edit"
  ."  width 200\n"
  ."%action input override"
  ."  {<Key>Return: exec("
  ."  echo [gV input string])}\n"
  ."%label result top label {}"
  ."  width 200 fromVert input\n"
  ."%command quit top fromVert result"
  ."  callback quit\n"
  ."%label info top fromVert result"
  ."  fromHoriz quit label {}"
  ."  borderWidth 0 width 150\n"
  ."%realize\n";

# read loop
while(<STDIN>) {
  chop;
  if (/^\d+$/) {
    print
      "%sV info label thinking...\n";
    $starttime = time;
    for($d=2,@result=();$d<=$_;$d++){
      while (!($_ % $d)) {
        unshift(@result,$d);
        $_ /= $d;
      }
    }
    print "%sV result label {"
     .join('*',@result)."}\n"
     ."%sV info label {"
     .  (time-$starttime)
     .  " seconds}\n" ;
  } else {
    print "%sV info label"
      ." {invalid input}\n";
  }
}
```

Demo Applications of the Wafe Distribution

We have developed sample application programs in Perl, GAWK, Prolog, Tcl, C and Ada talking to the same Wafe binary. The following demo applications are among the programs distributed together with the Wafe sources:

- `xwafedesign`: interactive design program for Wafe applications (see Figure 6)
- `xwafeftp`: FTP frontend
- `xwafemail`: Mail user frontend with faces, using elm aliases
- `xwafenews`: NNTP based news reader, using elm aliases
- `xwafegopher`: a simple gopher frontend
- `xdirtree`: tree directory browser
- `xbm`: bitmap and pixmap viewer
- `xwafemc`: multiple choice test answering program
- `xruptimes`: rwho monitor like xnetload
- `xnetstats`: network statistics, frontend for `netstat -i <interval>`
- `xvmstats`: system statistics, frontend for `vmstat -i <interval>`
- `xiostats`: I/O statistics, frontend for `iostat -i <interval>`
- `xwafeping`: pings several machines and shows up-status
- `xwafecf`: a simple read-only card filer
- `xwafetel`: a simple read-only Oracle front-end for looking up telephone numbers
- `xwafeora`: a more elaborated Oracle frontend with updates, capable to model an entity type with distinct attribute defined subtypes, allowing multi valued attributes. The sample program supports field completion and other funky stuff. `xwafeora` is configured via a parameter block containing the sample applications “Filing Management” and “Paper Base”.
- `perlwafe`: an example program calling Wafe as a subprocess of the application program (normally, it is the other way round).

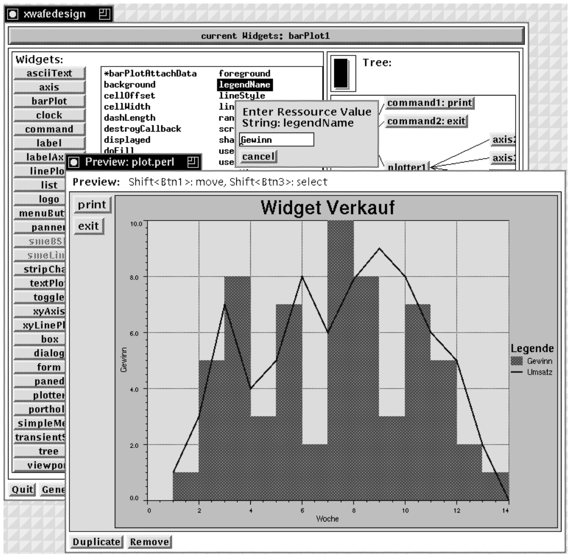

Figure 6: Sample Screen Shot of `xwafedesign` using Xaw3d and the Plotter Widget

Experiences

Our experiences proved that

- Wafe applications can be written in a wide range of programming languages,
- Wafe provides a relatively high level interface to widget applications,
- a single Wafe binary serves multiple applications,
- Wafe achieves a better refresh behavior when the application program is busy,
- click ahead is possible due to buffering in the I/O channels,
- Wafe allows better separation between user interface and application program matters,
- from its performance a user cannot distinguish whether a widget application was developed using C or Wafe,
- there is no need to program in C in order to develop widget frontends, and
- migration from existing ASCII based programs to X Window applications is easier using Wafe.

For the click ahead feature mentioned above it is questionable whether this is a desireable feature. It can be deactivated by setting widgets insensitive or by writing a small Tcl procedure which checks for each interesting callback procedure whether the program is in a busy state or not and writes accordingly friendly messages to the user.

The main disadvantage of Wafe is - when compared to widget programming in C - the higher resource consumption, because every Wafe application needs an additional process (the frontend). Frequently it is necessary to duplicate data (such as a text to be displayed in a text widget), since one copy has to be available in the frontend and another copy in the application process.

Availability

Wafe was developed on DECstations 5000/200 under Ultrix 4.2 using X11R5, and has been compiled on SparcStations under SunOS 4.1, RS6000/320 under AIX and on HP 9000/720 under hpux 8.05. Wafe can be compiled for X11R5 and X11R4. The preferred program-to-program communication is done via socketpair. Support for PIPES and SYS V streams is included for systems without the socketpair system call. The actual Wafe version and the sample applications mentioned above can be obtained via anonymous FTP from

```
ftp.wu-wien.ac.at:
pub/src/X11/wafe/*
```

(ip address: 137.208.3.4). At the time of the conference at least version 0.93 will be available. Since Wafe was announced first in May 92, about 2200 FTP-requests for Wafe were issued at the mentioned server.

References

[1] John K. Ousterhout, *Tcl: An Embeddable Command Language*, Proc. USENIX Winter Conference, January 1990.

[2] John K. Ousterhout, *An X11 Toolkit Based on the Tcl Language*, Proc. USENIX Winter Conference, January 1991.

[3] Larry Wall, Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Sebastopol 1991.

[4] Joel McCormack, Paul Asente and Ralph Swick, *X Toolkit Intrinsics - C Language Interface*, Massachusetts Institute of Technology, 1990.

[5] Ralph Swick, Terry Weissman, *X Toolkit Athena Widgets - C Language Interface*, Massachusetts Institute of Technology, 1990.

[6] Arnaud Le Hors, *The X PixMap Format*, Part of the xpm distribution, export.lcs.mit.edu, 1991.

[7] Adrian Nye, Tim O'Reilly, *X Toolkit Intrinsics Programming Manual*, Second Edition, O'Reilly and Associates Inc., Sebastobol 1990.

[8] *X Toolkit Intrinsics Reference Manual*, Third Edition, O'Reilly and Associates Inc., Sebastobol 1992.

[9] Thomas Berlage, *OSF/Motif, Concepts and programming*, Addison-Wesley, Wokingham 1991.

[10] Kaleb Keithley, *Three-D Athena Widgets (Xaw3d)*, export.lcs.mit.edu, 1992.

[11] Roger Reynolds, *Rdd2 - Drag and Drop Library*, export.lcs.mit.edu, 1992.

[12] Peter Klingebiel, *AthenaTools Plotter Widget Set, Version 6-beta*, export.lcs.mit.edu, 1992.

[13] Doug Young, *XmGraph, A Motif Graph Widget*, iworks.ecn.uiowa.edu, 1992.

Author Information

Gustaf Neumann is Assistant Professor at the Vienna University of Economics and Business Administration, Department of Management Information Systems, in Vienna, Austria. His main research interests are centered around the intergration of heterogenous systems like the integration of different information analysis methods, the integration of various language layers (esp. in the field of logic programming and program transformation), applications of deductive databases and user interface issues. He has developed several free packages spread over the internet such as dvi2xx (a TeX dvi converter for HP LaserJets and IBM 3812 printers) and diac (conversion program for ASCII umlauts). Gustaf Neumann can be reached electronically as neumann@wu-wien.ac.at.

Stefan Nusser is writing his master's thesis at the department mentioned above. He can be reached over the network as nusser@wu-wien.ac.at.