Overview of GMW+Wnn System

Masami Hagiya Takashi Hattori Akitoshi Morishima
Reiji Nakajima Naoyuki Niide Takashi Sakuragawa
Takashi Suzuki Hideki Tsuiki Taiichi Yuasa

Summary. A window system GMW and a Japanese inputting system Wnn have been developed together to serve as flexible and network-extensible infrastructures for building workstation environments especially with use of Japanese languages. GMW is an general-purpose overlapping window system which is intended to facilitate construction of a wide range of application software for which graphical workstation environment with user-friendly interface is essential. It mainly features a clean and high-level imaging model as well as high user-programmability based on the virtual-machine-server-type implementation. Wnn is a highly user-customizable kana-to-kanji conversion system of the converting-several-sentences-by-one-request-type. It is the first Japanese inputting system with functions more advanced than or at least comparable to any commercially available ones, whose entire internal structures, including source codes and dictionaries, are completely open to the users. It serves as a flexible and powerful Japanese inputting front-end module for a variety of workstation application software. It is not a coincidence that both GMW and Wnn adopted a server-based implementation scheme because it is invaluable for achieving efficiency, network-extensibility, modifiability, and portability at the same time. Wnndesk, a Japanese-inputting window environment, has been developed to take full advantage of the availability of the both systems. This paper overviews the workstation environment which GMW and Wnn together provide.

1 Introduction

Successful development of a sophisticated and friendly user-interfaced workstation environment depends largely on well-prepared infrastructures of basic software which support various levels of applications. The GMW+Wnn system is intended to provide a powerful platform for graphical interface and Japanese input.

Common desirable features of such basic software include functionality, extensibility, efficiency, network transparency, and portability as well as user customizability. In user interface systems in particular, user customizability is one of the most important features because everyone has his own needs and taste in operation. Both GMW and Wnn are carefully designed to be fully customizable. In addition, GMW adopts a user interface management system that makes it easier to construct

ISBN 0-12-037101-4

applications with a customable user interface. In practice, so far, window systems have been often used for very limited purposes, e.g., as a device to provide multiple character terminals. It is intended to be a flexible and powerful infrastructure for constructing a wide-range of applications for which a graphical user interface is essential, though much still depends on its further development.

Effective use of non-European languages is another essential objective to be considered at the initial design stage of the basic software. Regrettably, this point is almost entirely missed in the designs of most system programs developed in the USA and Europe, which greatly reduces the adoptability of these systems. In many cases such difficulties are just barely avoided only by additional rewriting of the system which damages many important goals in the original design. (For example, an X window user must know how to distinguish XDrawString from XDrawString16 in order to output a text if it contains both Japanese and English words. In addition, X supports no standard way for implementing Japanese text input. The only possible way to get around this is to let each application do input by a front-end module, and this can be done only through a terminal emulator.)

These are the main considerations adopted in the design of GMW and Wnn that the authors of this paper at Kyoto University have produced in cooperation with ASTEC Inc. and Omron Tateisi Co. in Japan. Due to space limitation, this paper gives only a brief sketch. For a comprehensive description, see [7, 15, 16]. More documents in English are planned to be written (and include a translation of the textbook [15]). Both GMW and Wnn, with their source programs are distributed free of charge.

2 GMW, an Extensible Window System with a Flexible Imaging Model

Since Alto first showed the usefulness of multiple windows, a number of window systems have been developed and are considered to play a central role in workstation environments. Although there is no fixed definition as to how a window system should behave, the authors of this paper requires several characteristics that are absent in many of existing window systems, in particular those on UNIX[1]; this is the main motive for introducing GMW. (Section 2.5 compares GMW with other systems.)

The main features of the GMW window system include:

- A clean and flexible imaging model (Section 2.1).
- Functions for Japanese characters (See Wnndesk in Chapter 4).
- A virtual-machine implementation schema (Section 2.2).
- A user interface management system (Section 2.3).

In this chapter, description of those features are followed by sections about implementation and comparison. As for the user interface management system, Hagiya, *et al.* [8] give a more detailed description.

[1] UNIX is a trademark of AT&T.

2.1 Imaging Model of GMW

2.1.1 Display Object

There are various items that appear on the screen, where all of them are physically represented by a set of pixels. However, it is, of course, not flexible or efficient to deal with them in such a way that application programs directly manipulate pixels which depend on the particular hardware. Therefore, one of the most important features of window systems is to provide a clear and abstract imaging model. GMW supports high level operations corresponding to logical structures of images. The primitive unit of such a structure is a *display object*, which is capable of producing an image on a bitmap display. For example, a character code alone is not a display object, but a code combined with particular font information and a procedure to be used to draw it onto the frame buffer forms a display object. The types of display objects currently supported by GMW are: *text objects* for an array of character code and font information, *bitmap objects* for a pixel buffer, *menu objects* for showing a pop-up menu, and *display desks* which can contain several windows (see 2.1.4). Extension of GMW is under way.

For each type of display object, creation, deletion, and a set of primitive operations are provided. For example, in case of the text object, we can write and read character strings, and we can also set and change its attributes, such as the size of the object, the font of characters, underlined or reversed characters, etc.

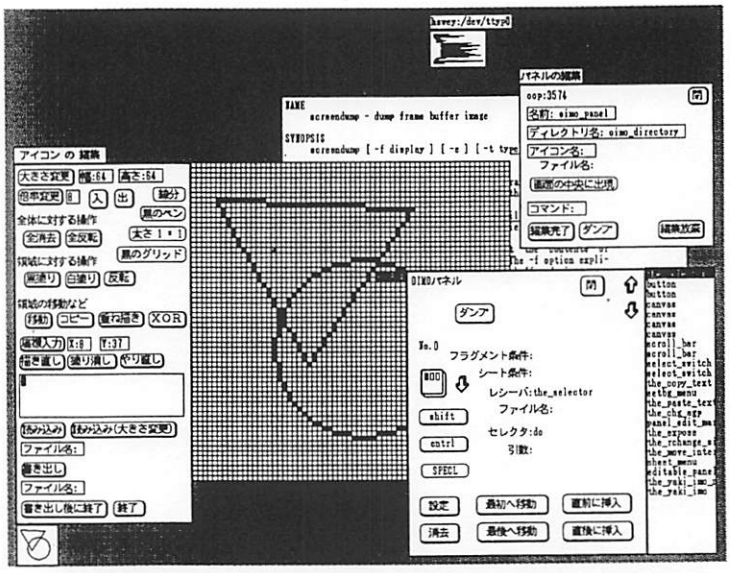


Fig. 1 An example of GMW display.

2.1.2 Display Fragment

A display object does not correspond directly to a region of the screen. The image that a display object produces is called a *display image*, and a rectangular area extracted from a display image is called a *display fragment* (Fig. 2). In contrast to a display object which represents an abstract model of the image to be displayed, a display fragment is a device to project the image on the screen. What the user sees are display fragments whose contents always reflect the display images from which they are extracted. If the user performs some operations on a display object, the window server will keep display fragments consistent with the corresponding display objects so that changes will immediately appear on the screen.

Separation of the display object and the display fragment gives a clear and flexible imaging model. For example, more than one fragment can be extracted from a display image so that different parts of a display object can be shown in several different windows simultaneously.

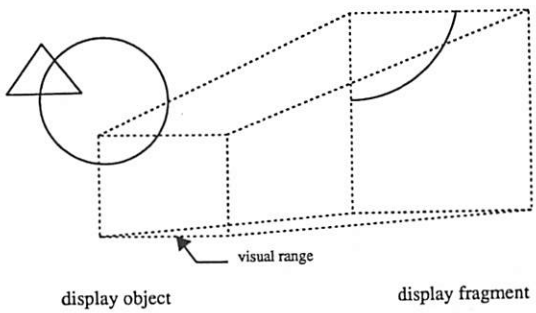


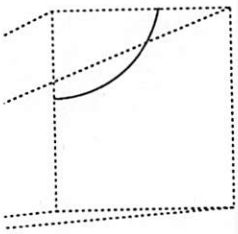
Fig. 2 Display fragment.

2.1.3 Display Sheet

An application usually uses several display fragments simultaneously. For example, a terminal emulator creates two display fragments, one for its body and the other for its title. When the user moves one of these fragment with a mouse, he naturally expects that the other will move in parallel. Therefore a group of several display fragments are tied together to form a *display sheet*, which corresponds to what has been conventionally called "window". The sheet itself does not have any specific shape, and each fragment is located relative to the origin of the sheet. The user interface manager provides the "move" operation which changes the origin of a sheet so that all display fragments belonging to the sheet move together.

Display fragments in a display sheet can be divided into several *fragment groups*, only one of which is shown on the display at a time. The typical application of this arrangement is an icon which is implemented using two fragment groups. One group contains usual window fragments, while the other group consists of a fragment for the icon. To open and close the window means simply to switch between these two

to a region of the screen. The image
isplay image, and a rectangular area
play fragment (Fig. 2). In contrast to
; model of the image to be displayed,
image on the screen. What the user
lways reflect the display images from
s some operations on a display object,
ıts consistent with the corresponding
tely appear on the screen.
e display fragment gives a clear and
than one fragment can be extracted
of a display object can be shown in



ge

display fragment

fragment.

ragments simultaneously. For example,
gments, one for its body and the other
se fragment with a mouse, he naturally
l. Therefore a group of several display
y sheet, which corresponds to what has
sheet itself does not have any specific
'e to the origin of the sheet. The user
eration which changes the origin of a
ıg to the sheet move together.
be divided into several *fragment groups*,
; a time. The typical application of this
l using two fragment groups. One group
e other group consists of a fragment for
ans simply to switch between these two

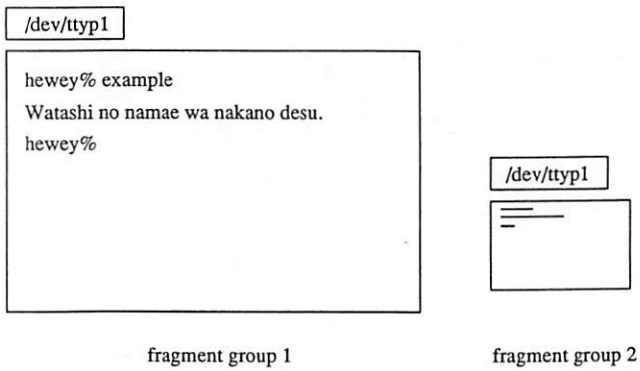


Fig. 3 Display sheet.

groups (Fig. 3). The origin of the location can be specified independently for each fragment group.

2.1.4 Display Desk

GMW has a display object called a *display desk*, which corresponds to the whole screen in other systems. A display desk contains a set of display sheets, and produces a display image in the same way as the physical screen (Fig. 4). In most other window systems, windows and the whole screen are completely different notions, or the screen is a special window defined by the system. However, since a display desk is itself a display object, we can extract a display fragment which is included in another display desk (Fig. 5). Consequently display objects, fragments, sheets, and desks form a recursive structure called a *display tree* (Fig. 6).

The *full screen desk* is a desk currently displayed on the screen. The user can choose any desk to be the full screen desk and make the image of the chosen desk appear on the screen. The physical bitmap display can be regarded as a special fragment, called the *full screen fragment*, which has a fixed size and is extracted from the desk or the full screen desk at the moment. GMW also allows a display sheet to be moved from one desk to another.

The recursive structures in display trees mentioned above can allow a group of several windows to be placed inside another window. This does not merely mean the parent-child relations which are already seen in some other window systems, but recursively nested windowing structures in which a user can move around from one window to another. To be concrete, a desk can be nested and user can make an inside desk the full screen desk. Although this function resembles the *project* of Smalltalk-80 [5], it is more general and powerful in that one can view and manipulate the contents of another desk as a window in the current full screen desk.

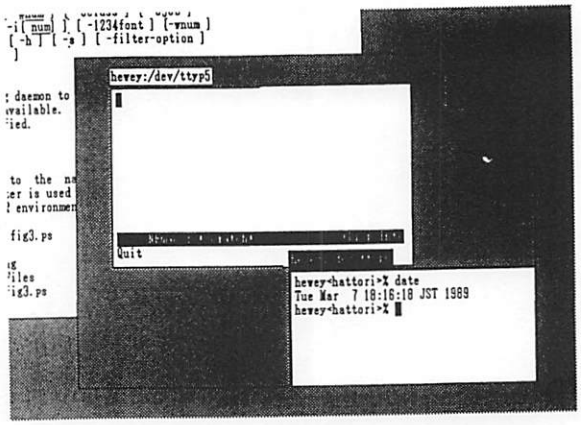
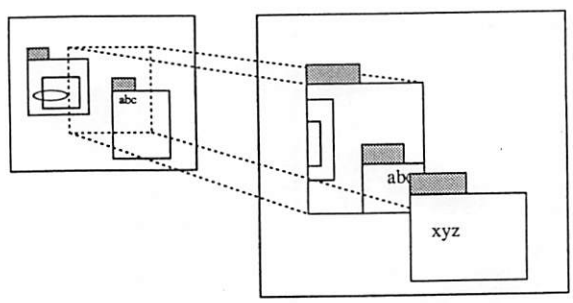
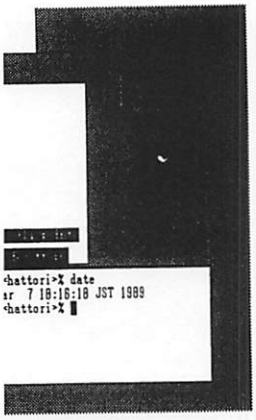


Fig. 4 Display desk.

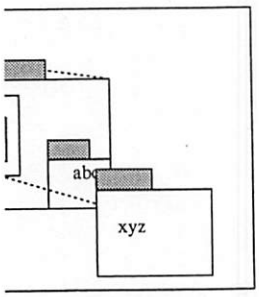


desk 1 desk 2

Fig. 5 Nesting of display desks.



desk.



desk 2

isplay desks.

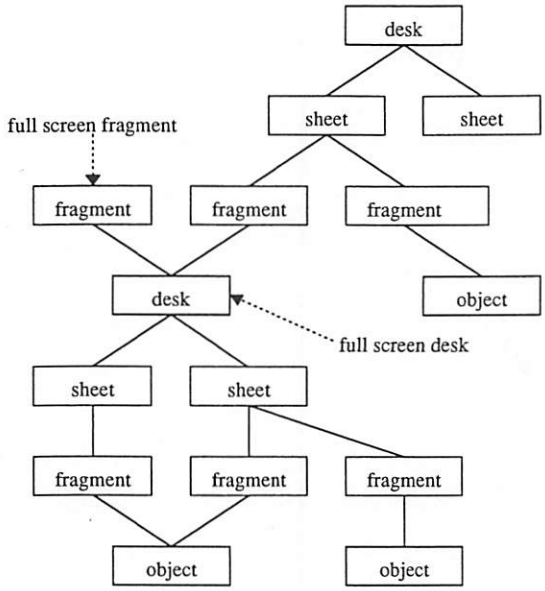


Fig. 6 Display tree.

2.1.5 Event and Redraw

In GMW, inputs from the user through such a device as a mouse and keyboard are taken as *events* which are usually sent to application programs. Another kind of event is a request to redraw fragments, which are generated in the window server and sent to application programs. Events are internally transmitted in the form of messages (see Sections 2.2 and 2.4 for description of messages).

An event is usually bound to a particular fragment: a redraw event corresponds to the fragment required to be redrawn (see the paragraph below). Events caused by keys and mouse buttons are directed to the fragment to which the mouse cursor points. For each fragment, the user can specify how to process the events directed to the fragment by setting a flag called *event behaviors*, i.e., to send a message to an associated application program, to discard them, or to pass them to the upper fragment in the display tree. If an event is passed to the upper fragment, it will be processed again according to the state of the event behavior of the fragment, and this procedure will be repeated until it is discarded or sent somewhere. An event behavior is specified for each kind of event. Usually, it is not necessary to specify event behavior because the default operator interface manager specifies at the full screen fragment level so that it can perform appropriate responses.

For a fragment extracted from a display desk, there is a special event behavior called *event intercept*, which makes the desk intercept all events directed to all fragments that belong to a subtree whose root node is the desk. This facility is,

for instance, used by Wnndesk (see Section 3), which intercepts key events, then performs kana-kanji conversion, whose results in turn are sent to the corresponding application.

Since GMW is an overlapping window system, a display fragment may be hidden by other fragments. If one of them is removed or the priority is changed, the newly exposed fragment should be redrawn onto the frame buffer. It must also be redrawn if the size of a fragment has changed. GMW provides two ways to perform this: One is to keep images of fragments in the window server. The window server automatically redraws the fragment when it is required. In this case, the user need not redraw it explicitly. The other way is that the application program redraws the fragment itself. In order to notify the application when redraw is required, the window server generates a redraw event. The former is faster and easier, but the latter is also necessary because the former requires a huge memory in the window server process when the number of fragments is large.

Each fragment has a *retained flag* to specify which way of redraw should be applied. If the retained flag is set, the image of the fragment is kept in the window server. If the retained flag is not set, the application program should observe a redraw event in order to do necessary operations. As for fragments extracted from text objects, however, the window server keeps the array of the character code and does not issue redraw events, but it automatically redraws them even if the retained flag is not set. This is possible because GMW has a feature that allows primitive operations to be performed at the level of display objects.

2.2 Virtual Machine

2.2.1 M and G

A server-based window system is one in which a special user process, called a *window server*, exclusively manages all operations on particular resources. The window server serves primitive functions for window operations, which are invoked by application programs. In the multi-tasking environment (e.g., in UNIX), such systems can take advantage of high portability and efficiency, compared to the systems which require special kernel calls or a huge library to be linked with user programs. However, the simple server-based system suffers such weaknesses as

- the protocol set must be fixed and the user cannot extend or modify it; and
- it may suffer from the overhead of interprocess communications, especially when applications are highly interactive.

GMW gets around these difficulties by the use of a virtual machine which runs in the window server process. The virtual machine allows the user to dynamically extend the protocol set between the client and the window server by loading programs into the window server, which can reduce the overhead of interprocess communications. (Independently, Sun NeWS [6, 18, 19] adopts a similar approach, where the virtual machine is a PostScript [1] interpreter.)

The other reason why a virtual machine is invaluable is that GMW adopts a user interface management system (UIMS) style [9]. Because the user interface depends heavily on the features of the window, it is most efficient for UIMS to be included in the window server. Therefore the window server must provide a virtual

machine on which the user can implement his own interface. (See Section 2.3.)

GMW provides a virtual machine called M which is based on concurrent objects and message sending. Objects send messages to one another which invoke their internal state transition. Input events from the mouse and the keyboard are sent to objects as messages. Activity often begins with an input event—causing a chain of message-sending toward objects that perform the job—which is proper to the application program.

A high-level language called G has been designed for writing programs that are compiled into the virtual code of M. G is a typeless concurrent object-oriented language with multiple inheritance which supports the facilities of M. See Hagiya, *et al.* [8] for details.

2.2.2 Delegate and Stub Description

If one could always write an application as an object in the window server, there would be no problems about communication. However, it is intended to place the user interface part of an application in the window server as an object of M, while the remaining part is implemented as a UNIX process in conventional languages such as C, because of efficiency and the need for library links and system calls. It is desired that internal objects communicate with external parts in a uniform way. In order to solve this situation, we introduce the idea of *delegates*. One delegate is automatically generated when an application process connects to the window server. Messages to delegates are converted into packets to the corresponding application processes and vice versa.

An application implemented as a UNIX process must include a routine to associate messages with procedures, which is a function of server stub in [2]. GMW provides a stub generator for C language which reads a stub description consisting of names of message selectors, types of arguments, and optionally, names of C functions. The stub generator produces both server stubs and client stubs written in C language. By linking server stub, message packets from delegates are associated with corresponding function calls. By linking client stub of another object, an application process can send a message to the object through a delegate (in other words, the application becomes a client of the object). Using a stub generator, what must be written by an application programmer is a set of C functions and a stub description which specifies how to call these functions from other objects. From a programmer's point of view, control is directly transferred from the user interface part to the corresponding C function.

2.3 User Interface Management System (UIMS)

Generally, an application on a window system consists of two parts: one is the user interface part and the other is the application proper. UIMS is the method of software management that divides applications into these two parts and manages the user interface parts under a unified software. The merits of UIMS are the following:

- It is possible to provide a user interface in a uniform manner. The user can work under the same user interface environment for different applications.

- It is possible to develop the application proper and the user interface separately. Therefore, prototyping of the user interface is possible.
- It is possible to design more than two different user interfaces for a single application according to various user's demands.

The GMW window system provides an infrastructure for user interface development environment and is designed to be suitable for adopting UIMS. GUIDE is a UIMS which includes graphical development facilities. (It is described in Subsection 2.3.2.)

2.3.1 Software Model

The former version of GMW [10], SUN NeWS [18], X10, X11, and some other window systems are designed according to the server-client model. In those systems, an application is a client, and it calls or sends messages to the window server which manages all the windows, pointing devices, and keyboard. In the current version of GMW, however, UIMS has the main control in calling an application, i.e., the application proper can be thought to be a server which processes messages sent by the user interface part. This is the "external control" type UIMS according to Ref. [9]. See Refs. [9] and [11] for the reasons why external control is better.

The virtual machine window system, like GMW, can naturally include UIMS in the window server. Therefore, the server-client relation of applications and the window server in external control UIMS is the opposite of conventional window systems. However, the drawing part of the window is a server for applications and UIMS. (Note that, as illustrated in Fig. 7, the function of the window system consists of UIMS and the drawing part.)

One of the most suitable computation models of UIMS seems to be the concur-

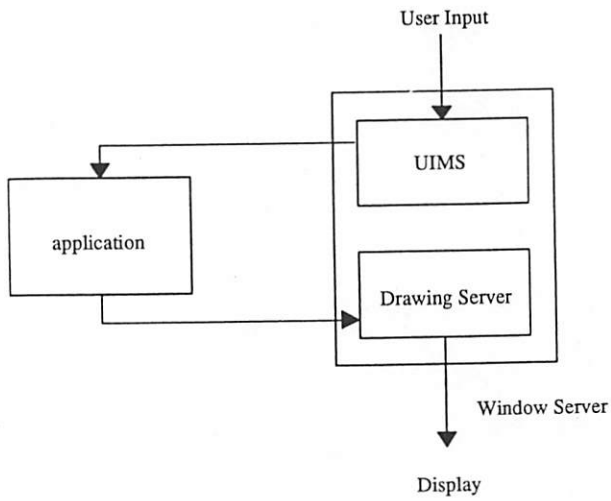


Fig. 7 UIMS, application, and drawing server.

rent object oriented model. In GMW, interaction techniques such as menus and buttons are concurrent objects implemented by the virtual machine M. Besides, GMW recommends writing an application as a server, i.e., the main routine is a message-awaiting loop. Considering the loop to be a state in which objects wait for messages, with uniform message transmission through delegates, we can virtually treat an application as an object.

2.3.2 GUIDE (GMW User Interface Design Environment)

GUIDE is a user interface management system aimed at a graphical user interface design environment. Its characteristics are the following:

- Basic interaction techniques are provided from which the user interface part of an application is constructed without coding.
- The attributes and layout of interaction techniques can be changed interactively even while the application is being executed. The result of a change can be saved for the next use.

More exactly, GUIDE consists of an operation interface manager called OIMO, primitive interaction techniques, and other utility objects, such as cut buffer. Currently, GUIDE provides the following tools for implementing interaction techniques: *button, switch, alternate switch, label, string holder, scroll bar*, and *message box*. An application usually uses several instances of tools, tied together to form a *panel* used as a controlling panel, a confirmer, etc. (Fig. 8). A panel can include other panels as its components. A pop-up menu is also implemented as a panel.

It is possible to change appearance of tools and panels. For example, its size and position of a button in a panel can be changed by selecting the menu and dragging the mouse. Sometimes a more complex operation is needed, e.g., to change a label printed on a button, which requires key input. In such a case, an editing panel is

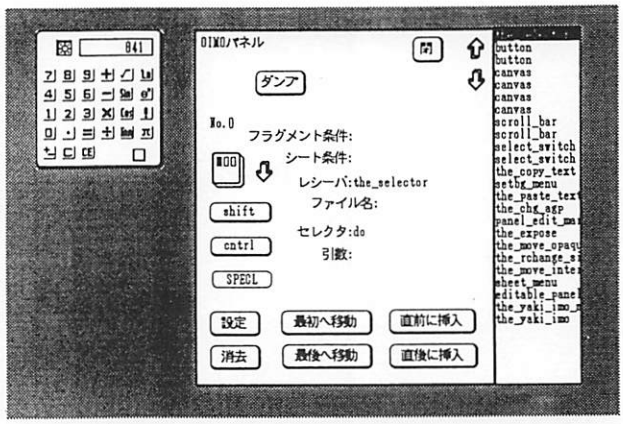
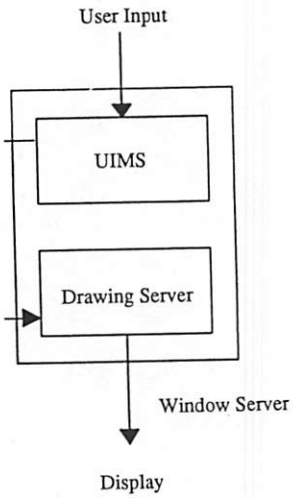


Fig. 8 Examples of panel.



invoked in which the user can edit the label. It is also possible to save the state of a panel after a change. The panel makes a G program and writes it into a file. The next time, loading of the G program restores the panel to the new state. Note that editing and saving are possible at any time regardless of the application program.

OIMO (Operator Interface Message Organizer) corresponds to a window manager in other window systems. It is included in GUIDE because it is one of the main components of UIMS and closely related to tools where OIMO mediates messages between the user and tools. OIMO charges itself with mouse events and common window operations such as opening and moving windows. All the events about mouse buttons are received by OIMO in general, and OIMO converts each event into a message and sends it to the corresponding tool so that it is possible to provide a uniform user interface among applications. It can be specified that a particular action should take place according to information attached to the fragment on which the event occurs. This feature is useful for making a reasonable response if some information is given by an application. For example, fragments of buttons have a common identifier for buttons. When the mouse button is pressed, OIMO sends a message suitable for buttons if the fragments have an identifier for buttons.

2.4 Implementation of GMW

The virtual machine M implements concurrent objects, where there are two different message-sending styles, SEND and RPC. While SEND only sends a message, RPC waits for a return value. When an RPC message has been processed, a return message is sent to the sender of the RPC. An object can process only a single message at a point in time.

Under a basic concurrent object-oriented model, recursive RPC is prohibited because such RPC waits for itself forever. M allows recursive RPC, however, in order to implement recursive algorithms and for efficient inheritance. This is possible by adopting the following *colored message model*.

A message sent by SEND is painted a new color. The object which processes the message is painted the same color. A message sent by RPC has the same color as its sender. When an object receives a message of the same color as its own, the current state of the receiving object (which must be waiting for a return value) is pushed and execution of the message starts. After the message has been processed, the former state is popped and execution resumes.

M also supports some functions of operating systems such as memory management. Memory management is required in M, because data for the window system are created dynamically and their sizes are not constant. Objects are referred to by indirect pointers in order to prevent problems with garbage collection.

The objects in M form a tree structure, each of whose nodes is called a *directory*. Directories are introduced in order to (1) protect objects of one client from other clients, and (2) manage *name spaces* so that one can refer to an object by its name. Each directory keeps a name space that binds each name to a certain object. All the name spaces in the system make a tree structure according to that of the directories, and they are searched from the current directory to the

l. It is also possible to save the state of
3 program and writes it into a file. The
:s the panel to the new state. Note that
regardless of the application program.
ganizer) corresponds to a window man-
led in GUIDE because it is one of the
:ed to tools where OIMO mediates mes-
charges itself with mouse events and
ıg and moving windows. All the events
O in general, and OIMO converts each
orresponding tool so that it is possible
applications. It can be specified that a
ng to information attached to the frag-
ıture is useful for making a reasonable
application. For example, fragments of
)ns. When the mouse button is pressed,
ıs if the fragments have an identifier for

rent objects, where there are two differ-
'C. While SEND only sends a message,
'C message has been processed, a return
:. An object can process only a single

ted model, recursive RPC is prohibited
M allows recursive RPC, however, in or-
for efficient inheritance. This is possible
model.
new color. The object which processes
nessage sent by RPC has the same color
ıessage of the same color as its own, the
:h must be waiting for a return value) is
s. After the message has been processed,
resumes.
rating systems such as memory manage-
M, because data for the window system
re not constant. Objects are referred to
:oblems with garbage collection.
each of whose nodes is called a *directory*,
protect objects of one client from other
that one can refer to an object by its
ace that binds each name to a certain
:em make a tree structure according to
rched from the current directory to the

root. The directory tree can be regarded as corresponding to a file system of an operating system. In addition, it is also used for better memory use by reducing unused cells as follows: When a connection between the application process and the window server is closed, the delegate object corresponding to the connection is automatically deleted. Its current directory and all the objects belonging to that directory are also freed. This can considerably defer the invocation of the garbage collector.

The GMW server includes the G compiler, which supports a higher interactive environment together with the debugging facility built in M. Since M code includes all information about the source program, it is possible to debug at source level.

The window server is written in the C language and is designed for UNIX or its equivalence. It was carefully designed to have high portability to UNIX machines. However, GMW requires some functions 4.2BSD has but System V is lacking, which, according to BSD terminology, are the select system call, sockets (at least in the UNIX domain), pseudo-tty, and FIONREAD ioctl. Demand-paging memory management is also needed, because the GMW server loads all the data, including kanji character fonts, to its address space (which amounts to more than 1MB).

GMW currently runs on Sun 3, 4, Sony NEWS, Omron SX 9100, LUNA, and Apple Macintosh. Those parts that depend on the workstation hardware are isolated to ease porting.

2.5 Comparison with Other Window Systems

GMW is essentially an independent work from other noted existing windows such as X, Andrew, SUN NeWS, although, in the late stage of its development, GMW was benefited from the experience of some of them. Here, we compare GMW with X and SUN NeWS in order to make clear the *raison d'être* of GMW.

Those features of GMW that are not seen in X are its abstract imaging model, the facility of display desks, and the virtual machine.

There are several advantages in the approach of GUIDE over library-based UIMS's such as the X toolkit. Since Widgets in the X toolkit are implemented by simple C structures, the X toolkit lacks the facility for multiple inheritance, a very useful tool for organizing the objects in the graphical user interface. Concurrency gained by the virtual machine is obviously an advantage over library-based UIMS's. Finally, in GUIDE, the interface between the application and the user interface is realized by an automatically generated stub and coincides with the interprocess communication interface, so that the user interface for an application can be changed without recompiling the application or even relinking it.

SUN NeWS is a virtual-machine-server-type window system whose virtual machine is the PostScript interpreter augmented with the facility of multi-tasking. In SUN NeWS, the communication interface between the window server and UNIX processes is provided by *ctops* [19], which generates a C program to send PostScript code and to receive a return value. However, the delegate and the stub description provide a more general and clean implementation scheme in that the C program can be considered as an object and can be treated in entirely the same way as objects in M. The imaging model of SUN NeWS is exactly that of PostScript, while

GMW supports a more abstract model based on display objects. The facility of display desks is not seen in SUN NeWS either.

3 Wnn, a Highly User-Customizable Japanese Text Inputting System

Customizability is one of the most important features of Japanese text inputting system since each user insists on his own way of inputting. On the other hand, it is also essential for such a system to be customizable in order that it may be widely used, in that everyone shares an essentially equivalent software without losing his own "identity" so to speak. It is regrettable that Japanese speaking people still have no common basic software for processing their own mother tongue. The available software are commercial products whose internal structures are concealed, so it is not possible for users to cooperate with one another in extending, refining, and strengthening a common inputting system.

In this respect, we expect Wnn, together with GMW, to serve as an infrastructure for a common computing environment for Japanese language use.

Before getting into a detailed description of Wnn, we give a quick introduction to the methods of inputting Japanese sentences into computers and make brief reference to the Japanese grammar. We also emphasize some of the crucial problems specific to workstation environments.

3.1 Japanese Text Inputting

3.1.1 Classification of Inputting Methods

It has been a challenging problem to work out an ingenious scheme for inputting Japanese sentences into computers. Since the Japanese character set consists of 88 phonetic hira-kana-characters, the same number of kata-kana-characters, and more than 1,000 frequently used kanji-characters, it is not possible to enter these characters using an ordinary keyboard in the ordinary way.

So, some attempts have been made at inventing a simple way of inputting Japanese characters.

Briefly, there are three ways of coping with the problem:

- tablet: a number of small buttons and some (or no) shift keys. Each combination of a button and a shift key specifies a unique character. Though easy to understand, this method needs a special key tablet and is quite time-consuming.
- two stroke method: a couple of key strokes on usual ascii or kana keyboard decides a unique character. It is necessary to memorize all the combinations of keys. Special training is indispensable, but this method is very efficient after training is completed.
- conversion from pronunciation: input sentences as sequences of phonetic symbols and convert them into the intended sequences of characters. Kana-characters are usually used as phonetic symbols, which can be inputted from Japanese keyboards directly or from ascii keyboards by romaji-to-kana conversion described below. It is necessary to select the intended kanji-words

among homonyms. This method is easy to understand and easy to use, and no special hardware is necessary, although the quality of the software counts.

The third method is called kana-to-kanji conversion and is currently the most frequently used.

3.1.2 Kana-to-Kanji Conversion

Japanese words are divided into two categories, *jiritsugo* and *fuzokugo*. A jiritsugo is like a noun or a verb in English. A fuzokugo, like prepositions or articles in English, can only be used together with a jiritsugo and adds a functional meaning to it. A *bunsetsu*, roughly speaking, consists of a jiritsugo and several fuzokugos attached to it and forms an element of a sentence which is equivalent to a subject or object in English. (It roughly coincides with the word *phrase* in linguistics.) A Japanese sentence is a sequence of several bunsetsus.

The grammar of Japanese determines how a fuzokugo can be connected to jiritsugo or other fuzokugos. Let us consider a similar situation in English. For the sentence "I can swim in the river," "I", "can swim," and "in the river" correspond to a bunsetsu, and "can", "in", and "the" correspond to a fuzokugo. The English grammar determines that "can" and "swim" can be connected but "can" and "river" cannot. In English, functional elements such as articles, prepositions etc., are usually prefixed. In contrast, in Japanese, a fuzokugo is postfixed. That is, a fuzokugo usually comes after a jiritsugo to which it is attached. A fuzokugo is usually written with one or more kana-characters, and a jiritsugo is often written with one or more kanji-characters.

As briefly mentioned in Subsection 3.1.1, there are many kanji-characters which have the same pronunciation. For example, the number of the kanji-characters which have the same pronunciation as the English letter "I" is 16 or more. In some cases, over 100 kanji-characters have the same pronunciation. The same situation occurs in the case of jiritsugos and in the case of bunsetsus. This is one of the problems that makes kana-to-kanji conversion difficult.

Kana-to-kanji conversion algorithms are classified according to the level of grammatical constructs to which they are applicable. The oldest and simplest one is called the single kanji-character conversion that converts a sequence of kana-characters to a single kanji-character which has the specified pronunciation. Because many kanji-characters have the same pronunciation (which is written in kana-characters) as mentioned in previous paragraph, the kanji-character produced from conversion may not be the one desired. So the operation of selecting one among many candidates is inevitably needed. The next level is the idiom conversion which can deal with kanji-idioms. (The word "idiom" roughly coincides with the word "jiritsugo".) On these levels, the connectivity of words are not used. Only the dictionaries are used.

The third level is the bunsetsu conversion that converts kana string to a sequence of kana and kanji-characters which form a single bunsetsu. The fourth level, which is currently the most advanced and the most popular, is called the multi-bunsetsu conversion, which attempts to convert the entire sentence represented by kana-characters to the corresponding normal Japanese sentence which has both kana and kanji-characters. The difference between the third level and the

きょうはいしゃにいった

Above is the kana string to be converted. By the start-conversion command, we get the following.

京　　歯医者に　いった
(きょう　はいしゃに　いった)

The resulted string is composed of three bunsetsus. Each bunsetsu is separated by a space here. By the change-bunsetsu-boundary command, we get the following string in which the first bunsetsu boundary is changed.

京は　　医者に　いった
(きょうは　いしゃに　いった)

By the next-candidate command, we get the following.

今日は　　医者に　いった
(きょうは　いしゃに　いった)

Fig. 9 Examples of kana-to-kanji conversion.

fourth level is that the fourth-level algorithm should punctuate the sentence into a sequence of bunsetsu. Since it is the custom of Japanese to write sentences without punctuation symbols or spaces between words, the system must guess the beginning and ending of the bunsetsu while converting a kana string into a kana-kanji string. (Imagine a situation in which one tries to parse an English sentence without any spaces.) The ambiguity of the separation into bunsetsu stems from this. So the method of determining the intended separation is required. Wnn implements this fourth-level conversion algorithm.

Through using each level conversion, a single kana string corresponds to more than a single kanji string. It is not possible to select automatically the intended kanji string without understanding the meaning of the input sentence; realization of such an approach is, more or less, a target of AI research and will require years to be developed for practical purposes. Therefore, a kana-to-kanji conversion system should have a good user interface by which a user is able to select the intended kanji string from among many candidates in a short time. In most of such systems, the candidates are presented in turn by the next-candidate-command which is bound to an appropriate key. Usually, the order by which the candidates are presented is determined according to some simple heuristic.

In general, the user interface for kana-to-kanji conversion consists of commands such as start-conversion, next-candidate, change-bunsetsu-boundary, and register-word-into-dictionary (Fig. 9). Each of these commands is usually bound to some particular function key or control key. Some systems also include simple editing facilities with which a user can edit kana string and/or kanji string without invoking a text editor.

3.2 Problems of Japanese Inputting on Workstations

There are problems that are particularly important with workstation environments.

Firstly, it is necessary to devise an interface which takes advantage of window systems, such as menus and variable-sized windows. Wnn has a window-oriented

and Technology **1**, 1989

いった

By the start-conversion com-

いった
いった)

nsetsus. Each bunsetsu is sep-
ısetsu-boundary command, we
)unsetsu boundary is changed.

いった
いった)

the following.

いった
いった)

to-kanji conversion.

ı should punctuate the sentence into a
of Japanese to write sentences without
s, the system must guess the beginning
a kana string into a kana-kanji string.
arse an English sentence without any
ıto bunsetsu stems from this. So the
ion is required. Wnn implements this

ingle kana string corresponds to more
ɜ to select automatically the intended
ning of the input sentence; realization
of AI research and will require years to
ore, a kana-to-kanji conversion system
user is able to select the intended kanji
ort time. In most of such systems, the
:t-candidate-command which is bound
which the candidates are presented is
tic.
kanji conversion consists of commands
ange-bunsetsu-boundary, and register-
ɜ commands is usually bound to some
ıe systems also include simple editing
ıg and/or kanji string without invoking

on Workstations

)ortant with workstation environments.
face which takes advantage of window
windows. Wnn has a window-oriented

interface which utilizes the merit of the notion of desks in GMW. (See Section 4.)

In a single-task environment, there are no difficulties about dictionaries because only a single task accesses them. On the other hand, mutual exclusion is needed and sharing of dictionaries have to be considered in a multi-tasking environment. In addition, if workstations are connected by a network, and dictionaries should be able to be shared network-wide, the problem gets more complicated. In order to get around these difficulties, Wnn adopts a server-based implementation. (See Section 3.5.)

3.3 Kana-to-Kanji Conversion in Wnn

3.3.1 Conversion Algorithm

Here, we describe the method for kana-to-kanji conversion that Wnn adopts. Since, as we have already seen, a Japanese sentence is a sequence of several bunsetsus, in order to convert a kana-string to a kana-kanji string, one must convert the kana-string to a sequence of bunsetsus. The method consists of two routines. One is a routine to cut off all candidates of a single bunsetsu from the tail of a given kana-string, using knowledge of words and their connectivity, and is invoked by the other routine, which, by the help of the former routine, determines which should be the most plausible sequence of bunsetsus for the given kana-string. We call the former routine the *get-a-bunsetsu* and the latter one *convert-a-sequence*. We explain first the *convert-a-sequence*, since the *get-a-bunsetsu* depends on what kind of knowledge is used about words and their connectivity and is rather complicated.

We need a method of evaluating sequences of bunsetsus in order to choose the most plausible one. For this purpose, we first introduce an evaluation function on the bunsetsus. The value of the function of a bunsetsu is defined in terms of what words the bunsetsu is composed of, the length of the bunsetsu, the frequency counter of the jiritsugo included in the bunsetsu, and how recently the jiritsugo was used. The user can customize the weights of these factors, and therefore, can define his own evaluation function. The valuation of a sequence of bunsetsus is defined in terms of the sum of the value of each bunsetsu in the sequence.

Since it is not realistic to enumerate all the possible sequences of bunsetsus and evaluate each of them to find the most plausible one, the *convert-a-sequence* finds in some way an optimal one. To do so, it first parses the kana-string from the end and enumerates all the possible sequences of n bunsetsus from the end of the given kana-string by repeatedly calling the *get-a-bunsetsu*. Here n is a customizable parameter whose default value is 2. Then it chooses one of the sequences which have the highest value. Now it fixes the last bunsetsu of the chosen sequence and removes it from the sequence. It repeats this process to the remaining kana-string until the sequence becomes null (i.e., until all bunsetsus in the sequence are fixed).

Next, we consider *get-a-bunsetsu*. The *get-a-bunsetsu* cuts off all candidates of a single bunsetsu from the tail of a given kana-string, using knowledge of words and their connectivity. As mentioned before, a bunsetsu is considered to consist of a jiritsugo and several fuzokugos which satisfy the grammatical rules for connectivity.

Firstly, we describe what the dictionaries contain to serve the *get-a-bunsetsu*. There are two kinds of dictionaries, the jiritsugo-dictionaries and the fuzokugo-

dictionaries.

A jiritsugo-dictionary contains a list of jiritsugos, J_1, J_2, ..., J_n, and for each J_i, a tuple ($yomi(J_i)$, $kana\text{-}kanji(J_i)$, $cat(J_i)$), where $yomi(J_i)$ is the kana-string and $kana\text{-}kanji(J_i)$ is the kana-kanji string composed of a mixture of kanji and kana which corresponds to the J_i itself. $Cat(J_i)$ is explained below. A fuzokugo-dictionary, on the other hand, contains a list of fuzokugos, and, for each fuzokugo F_i, a tuple ($yomi(F_i)$, $cat(F_i)$, $con(F_i)$), where $yomi(F_i)$ is the kana-string which corresponds to the $fuzokugo(F_i)$ itself. Note that fuzokugos do not have a kanji string $kana\text{-}kanji(F_i)$; they are always written in kana characters. So they need not be converted. The explanation of $con(F_i)$ follows.

Japanese words (which are in fact quite different animals from the word in European languages) are classified according to their backward connectivity, i.e., two words belong to a same class when the sets of words by which the two words are followed coincide.

Let q be a jiritsugo or fuzokugo. $cat(q)$ is the class to which q belongs. On the other hand, for a fuzokugo q, $con(q)$ is the set of the classes, where each class in the set consists of the words which q can follow. Each verb and adjective, which is a jiritsugo, changes its inflections according to what fuzokugo follows it. Inflections are written in kana-characters. So we only put the unchanging part of such a jiritsugo in a jiritsugo-dictionary, whereas inflections are regarded as fuzokugos.

A bunsetsu is a jiritsugo j followed by the fuzokugos f_k ($k = 0, 1, \ldots, l$) which satisfy $cat(f_l) \in fin$, $cat(f_k) \in con(f_{k+1})$ ($k = 0, 1, \ldots, l-1$), and $cat(j) \in con(f_0)$. (Here fin is the set of classes, where each class in the set consists of words which can be the last word of a bunsetsu.) The kana string of the bunsetsu is $yomi(j)$ followed by $yomi(f_k)$, and the kana-kanji string of the bunsetsu is $kanji(j)$ followed by $yomi(f_k)$.

Wnn adopts an efficient algorithm to cut off all the bunsetsus from the end of the given kana string, which needs no backtracking. This method is an extension of Kotodama [14]. The reason why we parse backward is to reduce the number of possible alternatives. The number of fuzokugos is by far smaller than that of jiritsugos.

In fact the situation is more complicated with *get-a-bunsetsu*, since we must take care of the prefixes (settou-go) and suffixes (setsubi-go) properly. Especially with Japanese, the problem becomes more difficult since some prefixes can proceed jiritsugos only when a particular bunsetsu comes after the bunsetsu. As an example, consider the sentence "o-yo-mi kudasa-i" which means "please read". "Yo-mi" stands for "read", where "yo" is a jiritsugo and the fuzokugo "mi" is an inflection of "yo" The prefix "o" and the bunsetsu "kudasa-i" together add politeness to the sentence, like the English word "please". In order to reduce the possibility of mis-conversion, we must use the fact that "o" and "kudasa-i" are always used together.

Therefore, we introduce a modified notion of bunsetsus and add another entry into fuzokugo dictionaries, etc. With those revisions, the capability of the *get-a-bunsetsu* is enhanced to cope with these difficulties. Although this improvement of the conversion algorithm is extremely important, we leave the details to other documents (see, e.g., [15]).

:itsugos, J_1, J_2, ..., J_n, and for each
)), where $yomi(J_i)$ is the kana-string
composed of a mixture of kanji and
(J_i) is explained below. A fuzokugo-
of fuzokugos, and, for each fuzokugo
ere $yomi(F_i)$ is the kana-string which
: that fuzokugos do not have a kanji
in kana characters. So they need not
lows.
: different animals from the word in
; to their backward connectivity, i.e.,
ets of words by which the two words

the class to which q belongs. On the
of the classes, where each class in the
Each verb and adjective, which is a
what fuzokugo follows it. Inflections
put the unchanging part of such a
flections are regarded as fuzokugos.
e fuzokugos f_k $(k = 0, 1, \ldots, l)$ which
$= 0, 1, \ldots, l-1)$, and $cat(j) \in con(f_0)$.
ass in the set consists of words which
ana string of the bunsetsu is $yomi(j)$
ng of the bunsetsu is $kanji(j)$ followed

off all the bunsetsus from the end of
racking. This method is an extension
se backward is to reduce the number
okugos is by far smaller than that of

d with *get-a-bunsetsu*, since we must
fixes (setsubi-go) properly. Especially
fficult since some prefixes can proceed
nes after the bunsetsu. As an example,
which means "please read". "Yo-mi"
and the fuzokugo "mi" is an inflection
'kudasa-i" together add politeness to
". In order to reduce the possibility
t "o" and "kudasa-i" are always used

n of bunsetsus and add another entry
revisions, the capability of the *get-a-*
fficulties. Although this improvement
portant, we leave the details to other

The first conversion algorithm was written in Kyoto Common Lisp, under whose interactive environment various experiments and prototyping were performed. As the development advanced, stable parts were gradually rewritten in C. Finally, the whole system was rewritten and was separated from the lisp environment.

3.3.2 Dictionary

It is very useful to allow a user to modify, extend, and edit a jiritsugo-dictionary, or even to build his own dictionaries, especially when one uses technical terms. Furthermore, it is extremely useful for a group of users to share or exchange dictionaries. In Wnn, each user can use his own dictionaries and common dictionaries with his own frequency files. A user can also specify which fuzokugo-dictionary to use. A user can change the set of dictionaries, frequency files, and a fuzokugo-dictionary to use in conversion even during a conversion session.

A new word can be registered into a dictionary by specifying part of the kanji string together with its reading and the grammatical category during the conversion.

The frequency files contain the frequency of use of each jiritsugo which is used in the evaluation of a bunsetsu. When a jiritsugo is selected during conversion, its frequency is automatically incremented. In Wnn, the frequency of each jiritsugo is stored in a 7-bit counter but the counter does not always count up, i.e., it is incremented with a certain probability. The probability is reduced according to the counter value and is equal to zero when the counter reaches the maximum value so that the counter never overflows. The frequency files also contain the flag for each jiritsugo, which is set up when the jiritsugo is selected and is reset when the jiritsugo is included in the mis-converted candidates.

We can convert dictionaries to human-readable text files. The method is also provided to convert human-readable text files to dictionaries, so one can easily create or update dictionaries without using Wnn. Utilities are provided in order to merge two dictionaries.

A user can also modify the content of a fuzokugo-dictionary by converting it to a human-readable text file, editing it, and then converting this modified text file to a fuzokugo-dictionary.

3.4 Romaji-to-Kana Conversion

Many users are accustomed to using *romaji* (roman letter writing for Japanese) to express kana characters. However, there are several ways of spelling kana in romaji, none of which are standard. Wnn was designed to support any way of spelling. The input from the keyboard is first passed to a finite-state automaton whose transition table can be defined by a user. It can be described as a simple table in which the left column is for romaji and the right is for kana. In order to write more sophisticated rules, one can define variables ranging over characters and also switch from one table to another among multiple tables.

3.5 Implementation of Wnn

To avoid the overhead of mutual exclusion which is needed when many processes access shared dictionaries, Wnn adopts a server-based implementation scheme like

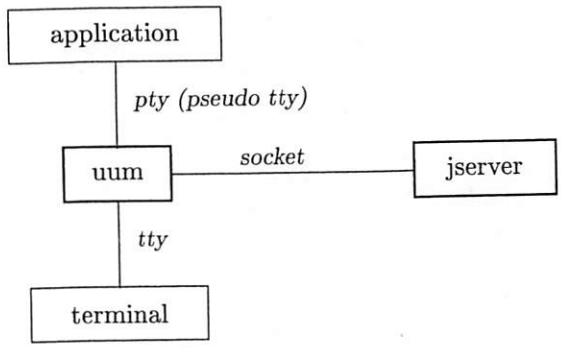


Fig. 10 The flow of *uum* frontend processor.

GMW.

Kana-to-kanji conversion and update of dictionaries and frequency files are exclusively done by the kana-to-kanji conversion server called the *jserver*. It reads an entire dictionary into its address space when the dictionary is needed. The connection between the jserver and clients is implemented by a UNIX socket so that a single jserver in network can serve all the machines, which enables us to use the same set of dictionaries on every machine.

There are two ways in which application programs utilize kana-to-kanji conversion facilities. One is to be mediated by a front-end which is a client for the jserver. The application program and the front-end are connected as if kanji characters are directly input from the keyboard so that no changes are required to the application program. The other is to become a client of the jserver. Libraries for interaction with the jserver are prepared.

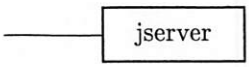
A Romaji-to-kana conversion facility is also available to application programs as a library.

So far, at least 3 front-ends—uum, wterm and Wnndesk—and one application program—egg—have been developed. Wnndesk is explained in Section 4.

Uum (wnn in the early version) is a front-end processor which is applicable on Japanese character terminals. Uum uses pty (pseudo-terminal) to send the converted string to the application program which requires Japanese input. When uum is invoked, the terminal display is divided into two parts. One is dedicated to kana-to-kanji conversion, which consists of the bottom line. The other is used by the application, and the result of a conversion is sent to this part as the tty input.

Though this kind of front-end has the advantage that it runs on usual Japanese terminals, it has two difficulties. One is that a user must perpetually move his viewpoint between the bottom line and the cursor position of the application program. Another problem is that the front-end stands between the terminal and the application, and the terminal is shared by both the front-end and the application. So the user must assign some keys to be used to start/end the kana-to-kanji conversion mode.

Wterm is a terminal emulator of the X window with a Wnn kana-to-kanji



frontend processor.

f dictionaries and frequency files are
sion server called the *jserver*. It reads
when the dictionary is needed. The
is implemented by a UNIX socket so
the machines, which enables us to use
ne.
programs utilize kana-to-kanji conver-
nt-end which is a client for the jserver.
are connected as if kanji characters are
changes are required to the application
f the jserver. Libraries for interaction

also available to application programs

m and Wnndesk—and one application
desk is explained in Section 4.
ont-end processor which is applicable
es pty (pseudo-terminal) to send the
which requires Japanese input. When
led into two parts. One is dedicated to
the bottom line. The other is used by
on is sent to this part as the tty input.
vantage that it runs on usual Japanese
hat a user must perpetually move his
cursor position of the application pro-
end stands between the terminal and
by both the front-end and the applica-
be used to start/end the kana-to-kanji

X window with a Wnn kana-to-kanji

conversion facility, developed by Ishizone with SRA Inc. Using wterm, conversion is performed at the cursor position. As wterm receives all the key-events directly, conversion facilities can be bound to keys which are not used by usual terminal applications such as meta-characters.

Egg is an extension of gnu-emacs with a Wnn kana-to-kanji conversion facility, developed by Tomura and Ishikawa with Electrotechnical Laboratory MITI. Egg is an application which does communicate with the jserver by itself. Egg makes it possible to preform the conversion at the cursor position. (Since it is an application itself, one need not be worried about key binding conflict.)

4 Wnndesk: GMW Window-Oriented Kana-to-Kanji Conversion Front-End

To use Wnn together with GMW, one can invoke uum in a terminal emulator window without taking any advantage of the sophisticated GMW window environment. Moreover, window-oriented applications that do not work in a terminal emulator window, such as, toolkits or desk-top publishing systems cannot be used in this way.

Wnndesk is a GMW window-oriented kana-to-kanji conversion front-end processer. Wnndesk consists of a desk (see Subsection 2.1.4) together with a kana-to-kanji conversion window. Keyboard inputs to fragments located under the desk in the display tree are directed to the conversion window, where conversions can be performed. When the conversion is done, the converted string is sent to the application program as if it were directly inputted from the keyboard. In case the application program runs on a terminal emulator the result of the conversion is first sent to the terminal emulator, which in turn sends it to the application.

Wnndesk utilizes the features of the display tree (see Subsection 2.1.4) and events (see 2.1.5) of GMW. Since the connection between Wnndesk and the application is not implemented by pseudo-tty but by the events of GMW, the connection can be dynamically established or destroyed as the display tree is modified. For example, if a terminal emulator is moved into the desk on the display, say by the mouse, even while an application is running on the terminal, it becomes possible to input a converted kanji string into the terminal. Conversely, if it is moved outside of Wnndesk, it becomes a "non-kanji terminal" for which only alpha-numeric symbols can be input and for which the key bindings for the conversion are freed. Wnndesk takes advantage of GMW environment during the conversion. If the conversion is done on a text object, the conversion window appears at the position of the cursor of the text object. Therefore, unlike uum, the user need not move his viewpoint between the cursor position and the bottom line. As the input string gets longer, the conversion window grows larger accordingly. Wnndesk creates another window when a list of candidates is shown. Panels are used in order for the user to register new words and to select the dictionaries needed for conversions.

As with other parts of GMW+Wnn system, the user interface of Wnndesk can be customized by a user interactively.

Though, at present, Wnndesk is implemented as a C program which calls the Wnn library directly, it will be rewritten as objects of a GMW window server in the

G language in the near future, in order to increase flexibility. The communication with the jserver will be performed through a delegate of a process which connects to the jserver. The delegate also makes it possible to utilize the jserver facilities from other objects.

Although GMW and Wnn can be used separately, it is greatly advantageous to have both of them united in a single environment as the above role of Wnndesk demonstrates.

Acknowledgments. There are a number of people other than the authors of this paper who made important technical and nontechnical contributions to the project. We express deep appreciation to them for all their effort. We would like to thank Mr. Hiroji Iwasaki for his constant support of the project.

References

[1] Adobe Systems: *PostScript Language Reference Manual*, Addison Wesley, 1985.

[2] Birrell, A. D. and Nelson, B. J.: Implementing Remote Procedure Calls, *ACM Trans. Computing Systems*, Vol 2, No. 1 (1984), pp. 137–143.

[3] Gettys, J.: Problems in Implementing Window Systems in UNIX, *Winter USENIX Proceedings*, Denver, pp. 89–97 (1986).

[4] Gettys, J., Newman, R. and Fera, T. D.: *Xlib—C Language X Interface, Protocol Version 10*, 1986.

[5] Goldberg, A. and Robson, D.: *Smalltalk-80: the Language and Its Implementation*, Addison-Wesley, 1983.

[6] Gosling, J. A.: "SunDew—A Distributed and Extensible Window System," in *Methodology of Window Management*, F. R. A. Hopgood, *et al.* (eds.), Springer-Verlag, 1986.

[7] Hagiya, M.: Introduction to the GMW Window System, *Technical report in RIMS 565*, Kyoto University, 1987.

[8] Hagiya, M., Hattori, T., Kakuno, H., Kojima, A., Ohtani, K. and Liu, S. L.: Developing Applications Based on GMW Window System, *Proc. of 4th Software Workshop* (1988), Japan Society for Software Science and Technology, pp. 23–29 (in Japanese).

[9] Hayes, P. J., *et al.*: Design Alternatives for User Interface Management Systems Based on Experience with Cousin, *SIGCHI'85: Human Factors in Computing Systems*, pp. 169–175.

[10] KABA Software Research Group: Overview of GMW+Wnn System, *2nd IEEE Conference on Computer Workstations*, 1988.

[11] Koivunen, M. and Mantyla, M.: HutWindows: An Improved Architecture for a User Interface Management System, *IEEE Computer Graphics Applications*, 1988.

[12] *MIT Project Athena: X Protocol Version 10*, 1986.

[13] Nagao, M., ed.: *Information Processing in Japanese Language*, Korona-sha, 1984 (in Japanese).

[14] Ookouchi, M., Fujisaki, T., and Morohashi, M.: Grammar Analysis for Kana-Kanji Conversion, *Keisan-Gengogaku*, Vol. 25, No. 4 (1981), (in Japanese).

[15] RIMS Software Research Group: *Introduction to GMW+Wnn*, Iwanami-Shoten, to appear (in Japanese).

[16] Sakuragawa, T.: Wnn—An Open Japanese Inputting System, *bit*, Vol. 19, No. 10 (1987), pp. 13–23 (in Japanese).

[17] Scheifler, R. and Gettys, J.: The X Window System, *ACM Trans. on Graphics*, Vol. 5, No. 2 (1986).

[18] Sun Microsystems: *NeWS Preliminary Technical Overview*, 1986.

[19] Sun Microsystems: *NeWS Manual*, 1987.

Received Sept. 1988.

Masami Hagiya
Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto
606 Japan

Takashi Hattori
Keio University
2–15–45 Shiba-Mita
Minato-ku, Tokyo
158 Japan

Akitoshi Morishima
Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto
606 Japan

Reiji Nakajima
Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto
606 Japan

Naoyuki Niide
Educational Center for Information Processing
Kyoto University
Sakyo-ku, Kyoto
606 Japan

Takashi Sakuragawa
Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto
606 Japan

Takashi Suzuki
Advanced Software Technology & Mechatronics Research Institute
Shimogyo-ku, Kyoto
600 Japan

Hideki Tsuiki
Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto
606 Japan

Taiichi Yuasa
Department of Information and Computer Science
Toyohashi University of Technology
Tempaku-cho, Toyohashi
440 Japan