Canae: A Platform for Constructing Graphical User Interfaces with Editors

JUNICHI REKIMOTO*, HIROYUKI TARUMI*, MASARU SUGAI*, GO YAMAZAKI**, KANEMITSU IGARI***, TAKESHI MORI*, TAKAHIRO SUGIYAMA*, ATSUKO UCHIYAMA* and CHUZO AKIGUCHI*

As window systems become popular, there is a growing need for a Graphical User Interface (GUI) that allows users to manipulate objects on the screen directly. However, development of GUI is not an easy task. We have developed a graphical user interface constructing environment which supports editing facilities of six media types (including text, image, diagram, graph structure, table, and hierarchical structure). This system, called *Canae*, is intended to be a general platform for several interactive and graphical applications. Canae provides various customization methods and an extension language for application developers to use editor parts as components of user interface. For modifying and extending these editors to accommodate to an application's needs, we use the MVC paradigm and object oriented approach in designing Canae. Application programmers can modify editor's keyboard handling and mouse handling by creating application specific *Event Maps* and *Key Maps*. Canae also provides a mechanism that enables applications to associate application's data and data in Canae. Canae is widely used for building several product-level applications. By evaluating three CASE applications developed with Canae, we have found that Canae reduces the amount of newly developed program codes by about 50% or more.

1. Introduction

As window systems become popular, there is a growing need for a Graphical User Interface (GUI) that allows a user to manipulate objects on the screen directly. Today, the use of workstation is not limited to engineers or computer specialists. Application programs for novice users must have graphical and intuitively understandable user interfaces.

However, the development of GUI is not an easy task, because it requires highly specialized programming techniques. Although GUI part is not a central part of the application, the cost of developing GUI tends to be more expensive than application specific parts, such as data management parts in a database application.

GUI part should be developed in an incremental fashion, because it is difficult to design a "good" user interface without feedbacks from users.

To achieve these requirements, window systems usually provide a "toolkit", which is a collection of interactive techniques, such as buttons, menus, and scroll bars. Toolkit frees application programmers from low-level programming of user interface. Although toolkits are useful, they do not help the most cost-intensive part of GUI development, that is a construction of direct-manipulation interface which allows the user to manipulate graphical objects on the screen directly. For example, it is quite difficult to construct a module structure editor in a CASE system, or to construct a page-layout user interface in a DTP (desk top publishing) system, but toolkit does not help a programmer in these areas.

There is another way to support GUI development. Emacs and Hyper Card have script languages which provide methods to extend and customize their user interface. Emacs, the screen oriented text editor, has a Lisp interpreter (called Emacs-Lisp) to extend its editing functionality. By using Emacs-Lisp, the user can create various applications beyond ordinary text editors, such as an electric news reader. Therefore, Emacs is used as a general platform for constructing various text-oriented applications. These systems reduce the costs of user interface construction dramatically than traditional toolkits.

However, Emacs is limited to text based user interfaces; Hyper Card is also limited to texts and bitmaps. Window oriented applications tend to deal with various types of graphical representations (or *media*) other than

This is a translation of the IPSJ 30th Anniversary Award Paper that appeared originally in Japanese in Journal of IPSJ, Vol. 31, No. 5 (1990) pp. 602-611.
*NEC Corporation.
**NEC Microcomputer Technology, Ltd.
***NEC Scientific Information System Development Corporation.

text or bitmap. For example, a module configuration system might use a network chart to represent module interconnections, and a form sheet to represent the properties of the modules. At this time, there is no system that supports multiple media as well as extension languages.

According to the above discussion, it would be quite useful if there were a system which provides a customization mechanism and language with various types of media. We have developed such a system, called *Canae*[1]. Canae is a platform for development of various applications, especially for CASE systems. We have investigated various applications including CASE systems, and selected six types of editor parts (text, diagram, image, graph structure, hierarchical structure, and image) as building blocks of user interface.

This paper describes Canae's architecture and our experiences of applying it to CASE systems. In Section 2 through 4, we give the model and the design of the system. In Section 5, we evaluate example applications developed with Canae. Finally, in Section 6, we will discuss the model proposed by Canae.

2. The Architecture of the Canae System

Figure 1 depicts the system architecture of the Canae system. Canae is a client process of the X window [3] system running on a Unix workstation. Below, we will explain the main components in the Canae system.

2.1 Dialog Parts

Dialog parts provide basic interaction techniques such as buttons, menus, lists (scrolling menu), volumes, palettes (icon based menu), fields (for 1-line text editing), and panels which compose other dialog parts on the dialog box. Dialog parts correspond to toolkit described in Section 1. The Dialog parts are implemented as subclasses of the X-Toolkit [4, 5] *Widget*[2].

2.2 Editor Parts

The editor parts provide basic editing functionality for six types of media. We did not select an application specific media such as a music score to keep generality of Canae. Although six might not be enough to handle *all* kinds of applications, we expect that these media will widely cover many application domains. We will discuss the domain of Canae in Section 6.

The features and usage of each media are as follows:

(1) **Text.** Text media handles multi-font string (including Japanese Kanji character). It also provides Kana-Kanji conversion input. Text media is used as an *embedded* editor to other media (to edit strings in other

[1]We named our system "鼎(Canae)", which means a tripod in Japanese, as a hieroglyph that illustrates a system supporting a window.

[2]The word "widget", which means a small gadget on the screen, is the term used in the X-Toolkit.

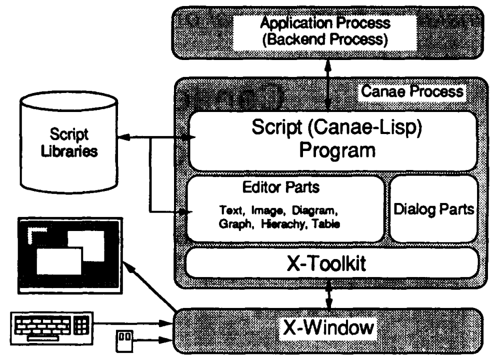


Fig. 1 The architecture of the Canae system.

media, such as cell strings in the table editor), as well as a stand-alone text editor.

(2) **Diagram.** Diagram is a composition of primitive pictorial elements, such as line, circle, rectangle, or text. This medium is useful for representing various kinds of charts, as well as ordinary diagrams.

(3) **Image.** Bitmap data, and raster data which are useful for handling scanned data.

(4) **Graph Structure.** Graph structure is a medium that consists of nodes and arcs. To represent a domain specific chart such as Petri-net, node and arc shapes are customizable by defining a scheme file. This media is useful for representing various CASE charts, such as module interconnection charts, process flow charts, and state transition diagrams.

(5) **Hierarchical Structure.** Hierarchical structure is a medium representing a tree structure. A node in the tree structure holds a title text, its contents data, and its subordinate structure. The contents data can be any kind of other media. This medium is useful for representing structured flowcharts, or structured documents such as manuals.

(6) **Table.** Table is a medium representing matrix data. The data can be any kind defined in other media. This medium is useful for creating form-based user interface, such as database query or data entry.

Canae allows an application programmer to create an application specific user interface by using these editors as building blocks. The application programmer also can use a Canae specific interpreter language (Canae-Lisp, which is discussed later), for customization, as well as C.

We have selected these six media types based on our experiences of developing several CASE tools, such as SDMS [6]. Media in Canae are selected not only from their appearances, but their internal data structures. For example, diagram editor and graph editor might create data having same appearances, but the internal structures of each is different. Graph medium handles node connections. For example, it is possible to create an application following nodes through their connection information. On the other hand, diagram editor can create more unconstrained pictures, though it does not support network structure. We believe that Canae is especially useful in developing CASE tools, which use various graphics representations as input methods and presentation methods.

2.3 Back-End Process

There are two ways to build applications using Canae. One is to use Canae as a library, and the other is to use Canae as a front-end process. The latter case, an application is built as a separate process called a *back-end process*. A back-end process is responsible for a semantic part of an application's task such as data- management in database systems. When using a back-end process, the front-end process, called Canae process, is responsible for user interface part of the applications. Canae process is controlled by Canae-Lisp, which is an interpretive language. A back-end process could be an existing (text oriented) application. This means that back-end approach can also be used as a method of adding a graphical user interface to existing tools.

Canae process has an event-driven control structure. There are two types of event: events from the X window, and events from the back-end processes. Events from the X window are classified into keyboard and mouse events. Upon receiving these events, Canae looks up Key Maps or Event Maps installed on the editor parts or dialog parts to find an appropriate action (a callback function written in C or Canae-Lisp). The Callback functions can send a request to the back-end process if it is needed. We will discuss KeyMap and EventMap later.

3. The Design of Editor Parts

In this section, we describe the architecture of the editor parts.

3.1 Design Goals

As described above, we aim to use editor parts as components of user interface. Therefore, editor parts should not be closed applications. They should have "open" architecture to constitute a user interface platform. Editor parts should also handle various media types. To accomplish these requirements, we set several design goals described in the following.

- It should provide a rich set of customization methods to meet the needs of applications. For example, keyboard and mouse event handlings should be customizable. It is also necessary to provide a mechanism that maintains a connection between data in the application and in Canae.
- Each editor should share a common architecture to minimize media dependent implementation.
- Editor parts should be fast enough to respond to user operations. Redisplay time might not be negligible even if Canae is running on high performance workstations available today. It is important to design a redisplay algorithm which makes possible for Canae to serve user operations as soon as possible even while Canae is executing the redisplay process.
- It should be possible to create a compound document, which is composed of two or more media. For example, it should be possible to place a diagram on the text media. We term this mechanism as *media-mixing*. Media-mixing should be achieved without affecting the implementation of each media.
- It should be easy to add more media types in the future.

3.2 Improved MVC Paradigm

To accomplish above design goals, we employ the *MVC paradigm* [7] as Canae's editor architecture. The MVC paradigm, which was developed at Xerox to implement the user interface of Smalltalk-80 [8], is a framework to construct a visual and interactive application by using triples of objects—Model, View, and Controller. A model holds information of an application. A view is responsible for rendering the contents of the model to the screen. A controller handles user operations. When the contents of the model have been modified, the view redisplays the screen. The Controller receives input events from the user, dispatches them to the model and the view.

Editors in Canae also consist of M-V-C triplet. A controller is implemented as EditorWidget, by defining a new class in the X-Toolkit widget class hierarchy. EditorWidget has pointers into a model and a view. Although model and view classes are defined for each media type, one EditorWidget class is used for all media types. A view is an instance of ViewClass, which holds various functions to display model objects. EditorWidget sends a message for displaying, and for scrolling media to the view object. It is possible to implement two or more ViewClasses for one medium. This means that there could be several visualization schemes for one medium type. A model has a pointer into MediaDescriptor, which holds media common editing operations (such as Cut, Copy, and Paste) and file I/O functions.

The view defines a drawing area called *canvas*. A canvas is a 2- dimensional area through which a view visualizes the contents of the model. EditorWidget defines an area called *viewport* in the canvas. It corresponds to the window of EditorWidget (see Fig. 2).

Separating views and models has several benefits. It is quite easy to set two or more views onto one model, which allows the user to manipulate different portions of the model simultaneously. It is also easy to set different types of views onto a model, which is essential if the application wants to display different aspects of the model, such as an outline view and a precise view.

Canae improves the MVC paradigm by adding the no-

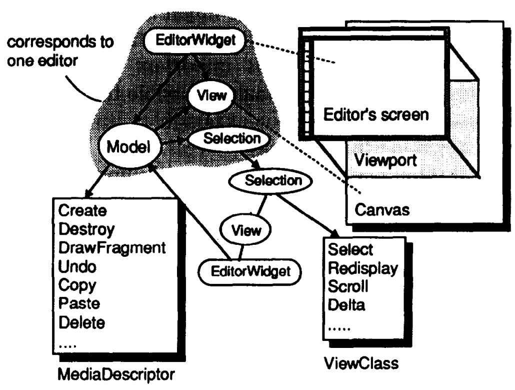


Fig. 2 The arcitecture of the Canae editor parts.

tion of *lazy redisplay*. In the traditional MVC paradigm, when a model has been modified, the corresponding view starts its redisplaying process immediately. However, if the model is changing continuously, the view must update the screen frequently (and wastefully). On the other hand, in our MVC scheme, a view does not start its redisplaying process immediately. Instead, the view is notified *delta* information from the model upon modification. The view stores the delta information to prepare for the future redis-playing. When input from the user stops[1], the top-level loop of Canae requests all views that need redisplaying to update its window.

The delta information is a trace indicating how a model has been changed. In many cases, the cost of holding delta information is very inexpensive. For example, a text editor in Canae uses three numbers (start and end position of modification in the text model, and the length of the newly replaced text) as delta information. The contents of the modified text are not stored in the delta. When the redisplaying process gets invoked, a text view can update the screen efficiently from (1) delta information, (2) updated model, and (3) view data which hold old information about the model.

The lazy redisplay technique also applies to other situations in addition to the model modification. For example, a view must redraw their screen, when a portion of the screen area maintained by the view is altered because of the change in the window's priority or size, or when a user requests to scroll the contents of the view. In such cases, the view remembers those events as deltas instead of updating the screen immediately. While the user moves the slider of the scroll bar very fast, the screen is not updated. When the user stops or moves slowly the slider (without releasing the mouse button), the redisplay process is invoked.

[1]It can be determined by checking the contents of the event-queue in the Canae process.

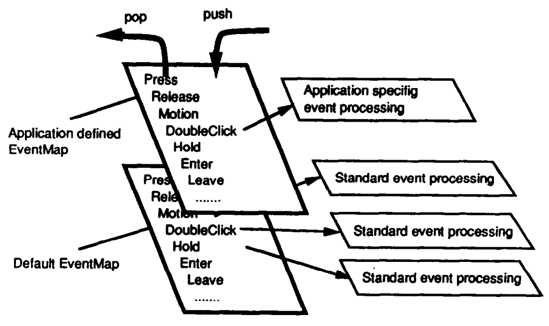

pop
push
Press
Release
Motion
DoubleClick
Hold
Enter
Leave
....
Application defined EventMap
Default EventMap
Application specifig event processing
Standard event processing
Standard event processing
Standard event processing

Fig. 3 The structure of the Event Map.

3.3 Customization

In this section, we briefly describe customization facilities used in the Canae system.

To make a user interface be customizable, each editor needs to have KeyMaps and EventMaps. A KeyMap defines a mapping between a keyboard input sequence and a callback function, while an EventMap defines a mapping between a mouse input event and a callback function. Callback functions can be written in C or Canae-Lisp. Application can define a new KeyMap or EventMap, which defines application specific event—action binding, and push it onto the Canae's standard Key(Event)Map.

Actions defined in the new map override actions defined in the lower maps, if they have same event sequences. For example, if an application pushes its own EventMap which defines the actions to DoubleClick and MouseRelease, original definitions corresponding to these two event types will be overridden by them, while other definitions, such as MouseMotion, will remain (see Fig. 3).

To provide the default user interface to editor parts, Canae defines a standard KeyMap and EventMap. The standard EventMap provides basic direct manipulation user interface which allows the user to select, drag, and scale objects with a mouse. The standard KeyMap defines default key bindings of editing operations, such as Cut, Copy, and Paste. These maps are installed automatically at the creation of editors. If application programmers wish to alter the standard user interface, they can create an Event(Key)Map which defines a different user interface from the standard maps.

For example, consider an application that maintains module information and represents it by Graph media. An application programmer can define an EventMap which has the following definition:

```
On DoubleClick,
 1. select a node whose position corresponds to
    the mouse location,
 2. find a module that corresponds to the node,
 3. open a window, and display the information re-
    garding the module.
```

To implement step 2, there must be a relation between application data (modules) and objects in the Canae editor. This relation can be implemented by using *attributes* discussed in the next section.

3.4 Attributes

Canae allows applications to place their specific data for each object in the model. This mechanism is useful to relate objects in the model (e.g., nodes in the graph editor) to application data (e.g., module name and property). We call these data *attributes*.

An attribute is a tagged byte stream, and its usage is up to applications. Some typical usages of attributes are to store:

1. Canae-Lisp program
2. File name
3. Query command to database systems
4. Other application specific information.

Item 2 means that it is possible to implement a hypermedia system on top of Canae by using attributes as link data. It is also possible to implement a hot-link, a hypermedia link that looks for its destination by computing or querying, by using item 1 because we can store a navigation command written in Canae-Lisp as an attribute.

3.5 Media Mixing

Canae can combine one type of media data to other types of media. To implement this, we introduced a data structure called *fragment*. A fragment is a byte stream data which contains all information of the model. It is used in storing and sending (through a network) media data. It consists of a *shell* and *contents*. Contents represent the information of the model in byte stream format. They are composed of several blocks. The shell holds a medium type and the extent of media if it is displayed by the standard view. It is possible to get a MediaDescriptor (discussed in Section 3.2) from media type.

To explain how fragment is used in a media-mixing context, consider the case of pasting a diagram in the Text media. The diagram model is not copied directly to the text, because text, which has no knowledge about other media, does not handle diagram. Instead, a fragment, which is created by encoding the diagram model, is pasted to the text. The text media treats this fragment as some type of characters. Although text does not know the internal data structure of the diagram, by getting information from the shell part of the fragment, text can locate and display the diagram.

When editing an embedded fragment, a model is created by decoding the contents of the fragment. Adding a view and an EditorWidget to the newly created model, the user can edit the model. When the user finishes editing, Canae recreates a fragment by encoding the model. This fragment is stored in the parent model again.

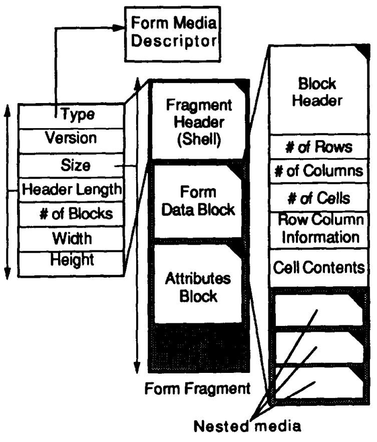


Fig. 4 Fragment data example.

In this way, by encapsulating media depending information into fragment, media-mixing is achieved without affecting each media's implementation. If new media type is added to Canae, this requires no modification to existing media. Note that fragment is also used as a file format of media data. Figure 4 depicts an example of fragment data format which corresponds to the table media.

4. Canae-Lisp

Canae-Lisp is a Lisp based interpreter language which aims to allow a programmer to build an application in the interactive environment. Programs written in Canae-Lisp are used in the following situations:

- callback functions set in EventMap or KeyMap.
- callback functions called from dialog parts.
- callback functions to handle back-end processes.
- callback functions embedded in the model as an attribute.

We selected Lisp as a customizing language for the following reasons:

- Advantages of Lisp as a customization language has been proven by the success of Emacs.
- Execution information, such as window layout, can be represented as a Lisp s-expression.
- Programmers familiar with the Lisp language can easily use Canae-Lisp.

Canae-Lisp can call C functions from interactive environment. This means that it is possible to use the X-libraries and the Canae libraries interactively. Canae-Lisp can also handle user interface objects (such as Model, View, Widgets, etc.) as primitive data types.

Figure 5 is an example Canae-Lisp program; It creates an EventMap which adds a label to a node or an arc of the Graph media when a user presses a mouse button. Note that a function beginning with a capital letter is a primitive function defined by Canae.

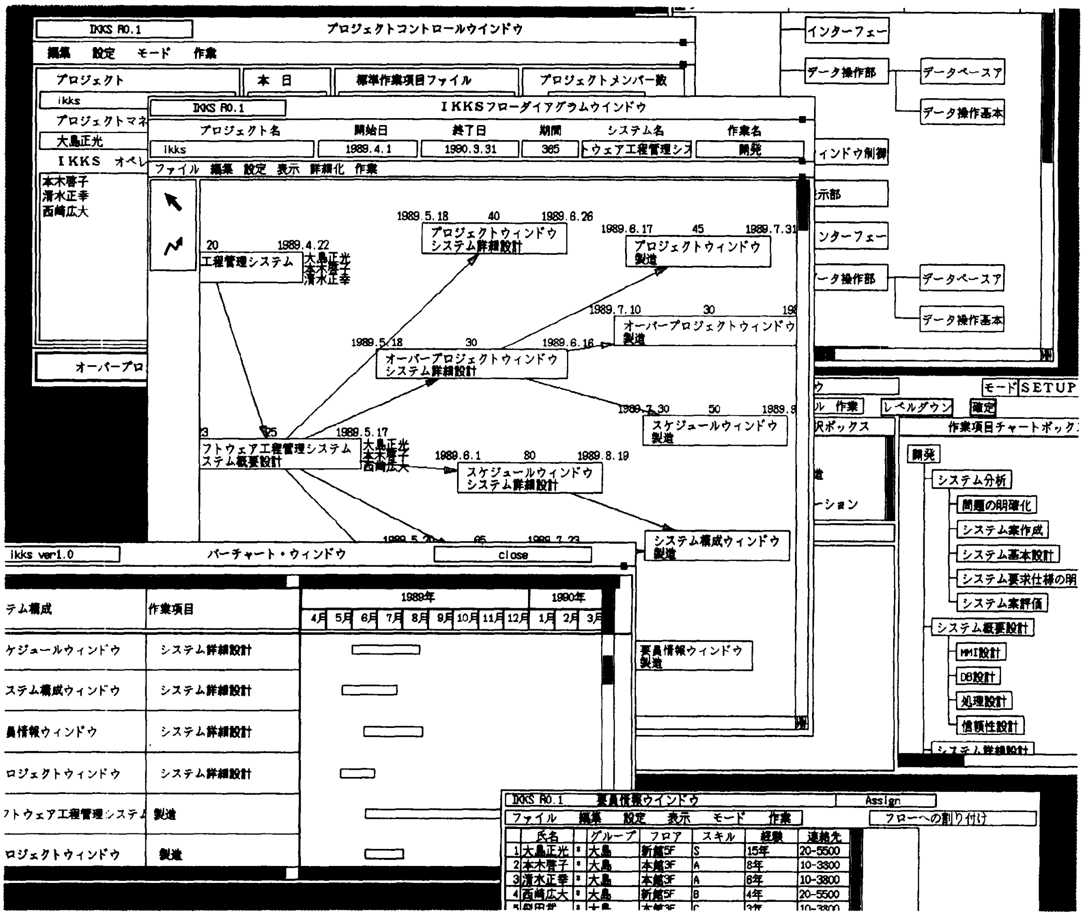
```
;;; Add a label to the node in the Graph editor
(defun edit-label (editor arg event-type x y event)
  (let* ((sx (CeDstXToSrc editor x))
         (sy (CeDstYToSrc editor y))
         (element (GeSearchElement editor sx sy))
         (label (GeCreateLabel
                  editor "label" element sx sy 100 50)))
    (GeEditLabel editor label nil)))

;;; Push an EventMap that adds a label text
;;; onto the Graph Editor.
(CxOverrideEventMap
 editor
 (CxCreateEventMap  CX_PRESS 'edit-label nil)
 NoEventMask)
```

Fig. 5. An example of Canae-Lisp program.

5. Evaluation

As of September 1989, the Canae system has been distributed to more than 30 divisions within NEC for building various applications. We have investigated three CASE systems developed with Canae to get quantitative data of Canae's performance as a user interface platform. The sizes of these applications are about ten thousand steps excluding the Canae library. They are written in C and build on top of the Canae library directly, i.e., back-end process approach and Canae-Lisp are not used.

Example-1 is a project management system. It can browse and manage the hierarchical structure of subprojects or subtasks, the schedules of subtasks, assignment of tasks to project members, and the hierarchical structure of the target system. Canae's graph editor parts are used to implement hierarchical structure charts of the project and the target system. Table editor is used for implementing the project member's table. Hi-



ikks ver1.0 バーチャート・ウィンドウ close

テム構成	作業項目	1989年 4月 5月 6月 7月 8月 9月 10月 11月 12月 1990年 1月 2月 3月
ケジュールウィンドウ	システム詳細設計	
ステム構成ウィンドウ	システム詳細設計	
員情報ウィンドウ	システム詳細設計	
ロジェクトウィンドウ	システム詳細設計	
フトウェア工程管理システム	製造	
ロジェクトウィンドウ	製造	

IKKS R0.1 要員情報ウインドウ Assign

ファイル 編集 設定 表示 モード 作業 フローへの割り付け

	氏名		グループ	フロア	スキル	経験	連絡先
1	大島正光	*	大島	新館5F	S	15年	20-5500
2	本木誉子	*	大島	本館3F	A	8年	10-3300
3	清水正季	*	大島	本館3F	A	6年	10-3800
4	西崎広大	*	大島	新館5F	B	4年	20-5500
5	[illegible]	*	[illegible]	本館3F	C	3年	10-3800

Fig. 6. A snapshot of the application developed with Canae.

Table 1. Amount of Canae's use in the real applications.

	Example 1	Example 2	Example 3
Canae version X version	V0R0 X11R2	V0R1 X11R2	V1R0 X11R2
(A) Application code	29.8%	18.9%	23.7%
(B) Canae basic library	2.6%	5.2%	8.2%
(C) Dialog parts	5.8%	7.7%	6.5%
(D) Japanese input library	5.4%	7.3%	5.7%
(E) Text editor	0.0%	3.2%	3.2%
(F) Graph + Diagram editor	11.0%	17.7%	21.6%
(G) Table editor	7.5%	0.0%	0.0%
(H) Hierarchy editor	0.0%	4.0%	0.0%
(I) Athena Widgets	10.4%	4.3%	6.3%
(J) X Toolkit	13.2%	15.1%	11.6%
(K) Xlib	14.3%	16.6%	13.3%
(L) $\frac{B+C+\cdots+H}{A+B+\cdots+H}$	52%	70%	66%
(M) $\frac{E+F+G+H}{B+C+\cdots+J}$	33%	38%	39%

erarchical editor is not used because it was not available at the development time. For showing schedules of subtasks, a horizontal bar chart is used. The bar chart is implemented by using table editor and diagram editor. Figure 6 depicts a snapshot of Example-1.

Example-2 is an upper level CASE tool. Canae's hierarchy editor is used to implement the hierarchical structure of functions and data. Graph editor parts are used to implement a chart representing relationships among functions, and a data from chart.

Example-3 is a middle level CASE tool. Graph editor is used to implement the process-flow chart and the relationship chart among programs and files.

Table 1 shows how Canae is used with the above three examples.

In Table 1, application code (A) shows the amount of source code developed originally for each example. (B) through (H) show the amount of source code of Canae, which is linked to the load module of each example. (I) through (K) show the amount of source code of the X window libraries linked to the load module of each example. The ratios are based on source code lines including comment lines. Library source code not actually linked are excluded. Since the diagram editor is always used with graph editor in these examples, they are counted together.

In Example-2, the ratio of (A) seems to be low because the application code itself is small. The ratios of application code in Example-1 through Example-3 are about 1.8, 1.0, and 1.6, respectively. The size the Canae library grows as its version increases; the ratios of total steps of (B)-(H) in Example-1 through Example-3 are about 1.0, 1.3, and 1.9, respectively.

The row (L) in Table 1 shows that double or more codes would be written to implement these examples, without Canae. These numbers also show the importance and the size of the user interface on workstation applications. Although the development cost may not be proportional to the number of source code lines, we can regard these percentages as showing the effectiveness of Canae.

We also believe that the costs of designing user interfaces are reduced, although we cannot give any quantitative data. Without Canae, an application programmer would have to design every user interface precisely. On the other hand, Canae defines the default behavior of user interfaces. A programmer can concentrate on designing the application specific user interface by leaving other common operations to Canae.

The row (M) shows the relative importance of editor parts in user interface. In case of these CASE tools, about one third of the code of user interface is editor parts.

6. Discussion

6.1 Comparison with Other UIMSs

The survey on UIMS written by Hartson et al. [13] reports that no UIMS exists which handles editing operations. From this point of view, Canae is different from other UIMSs. We consider that editing operations are closely related to the user interface, thus it is not a realistic approach to leave those operations to the application. For example, a user interface of selecting and dragging a pictorial element by mouse operation, is closely related to a representation of pictorial elements. Therefore, UIMS must have a knowledge about pictorial elements.

On the other hand, if the UIMS manages editing operations, how to relate data maintained by the UIMS and data maintained by the application? In Canae, an application can define an EventMap or KeyMap watching user operations and notify them to the application. However, this method is insufficient because a model, which is the data in the UIMS (Canae), could be modified without user interactions. For example, some script program might modify the contents of the model upon receiving an event from a back-end process. Event(Key)Map can not capture such modifications.

We are planning to solve this problem by introducing *hook* into Canae. A hook is an application defined function invoked on calling a primitive modification method of models. By using hooks, it is possible to notify any modifications onto the model to the application as well as the application which can inhibit the modification if it is needed.

6.2 Application's Domain

Canae assumes that all of the presentations used in the application can be implemented by combing six types of media. Of course, this assumption does not apply to all kinds of applications. For example, Canae might not be valuable when creating an application that needs to handle voice data. From this point of view, Canae is somewhat a domain specific UIMS rather than a general purpose UIMS. If an application is within the area that Canae anticipates, Canae is more valuable than other UMISs. Canae is designed to support CASE systems and Office System in mind. Most applications in these areas can be represented by combining six types of media that Canae supports. It is possible to implement user interfaces needed in those applications by using customization mechanisms provided by Canae. We consider this goal has been achieved.

On the other hand, consider to implement a desktop publishing (DTP) system with Canae. It might be difficult to develop a WYSIWYG DTP system with Canae, because the text editor of Canae does not support a fine text layout such as kerning or ligature. However, it is still possible to develop a DTP system which creates a structured text by using the hierarchical editor; defines a layout scheme by using the diagram editor, and shows page images by using the image editor.

One possible approach to extending the domain of Canae, is to provide a mechanism that allows applications to define new media types. However, it is uncertain if enough performance is gained when media is defined in Canae-Lisp. These issues require more study.

6.3 Consistent User Interfaces among Applications

It is desirable to make user interfaces consistent among several applications, because it reduces a user's effort to learn each application operation. Sakamura [15] claims that there are three approaches to promote standarization; they are (1) using metaphor, (2) providing intrinsics, and (3) creating standard applications. Canae is also helpful to standarize user interfaces among applications, because it provides higher level parts (editor parts) compared to other toolkits, and default user interfaces are determined by EventMaps and KeyMaps. Applications developed with Canae share integrated behavior, because they inherit the default user interface from Canae. This means that Canae provides intrinsics for editing operations.

7. Concluding Remarks

We have proposed a model to build a graphical user interface with editor parts, and have developed a system called Canae according to this model. We have evaluated three applications built on top of Canae, and proved that Canae reduces the user interface development efforts.

In Canae, application programmers must write a program in C or Canae- Lisp to define a user interface. We are developing an interactive tool on top of Canae, which enables application programmers to define user interfaces in WYSIWYG manner and generates source code skeletons. We are also planning to add an end-user programming facility to Canae, which allows end users (not application programmers) to customize and extend their environment without programming.

Acknowledgments

We deeply appreciate the help given by Shouhei Takeuchi and Chie Onuma, for their contributions on early discussions of Canae. We also appreciate the help of Nobuyuki Saji, who is a developer of CI-Lisp, which became the base of Canae-Lisp. We would also like to thank users of the Canae system for various suggestions.

References

1. Stallman, R. M. Emacs: The Extensible, Customizable, Self-documenting Display Editor, in *Proc. ACM SIGPLAN/SIGOA Conference of Text Manipulation* (1981).
2. Goodman, D. The Complete HyperCard Handbook, Bantam Books (1987).
3. Scheifler, R. W. and Gettys, J. The X window system, *ACM Trans. Graph.* 5, 3 (1986), 79-109.
4. McCormack, T. et al. X Toolkit Intrinsics—C Language X Interface, *X Window System*, 11, 3 (1988).
5. Weissman, T. et al. X Toolkit Widgets—C Language X Interface, *X Window Sytem*, 11, 3 (1988).
6. Shigo, O., Norifusa, M. et al. Software Production System for Communication and Control Software, *NEC Technical Journal*, 40, 1 (1987) (in Japanese),10-18.
7. Krasner, G. E. and Pops, S. T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, 1, 3 (1988).
8. Goldberg, A. and Robson, D. Smalltalk-80—The Language and its Implementation, Addison-Wesley, 1983.
9. Hazeyama, A. Harada, K. et al: A Software Development Control System IKKS(2)—Overview and Features—*Proc. 39th Annual Convention IPSJ* (1989) (in Japanese), 1411-1412.
10. Sugiyama, T. Yoshida, M. et al. The Japanese Interface Toolkit YUI based On Grammar Acquisiton from Japanese Example Sentences. *IPSJ SIGNL Notes*, No. 89-NL-73 (1989) (in Japanese).
11. Hirano, F. A User Interface for Multimedia Document Databases, *IPSJ SIGDB Notes*, No. 70-5 (1989) (in Japanese).
12. Watabe, K., Sakata, S. et al Distributed Multiparty Desktop Conferencing System: MERMAID, Preprints Work. Gr. for Office Systems. *Inst. Electronics, Inf. & Comm. Eng. Japan*, OS89-27 (1989) (in Japanese).
13. Hartson, H. R. and Hix, D.: Human-Computer Interface Development: Concepts and Systems for Its Management, *ACM Computing Surveys*, 21, 1 (1989), 5-92.
14. Young, R. et al.: Software Environment Architectures and User Interface Facilities, *IEEE Trans. Softw. Eng.*, 14, 6 (1988), 697-708.
15. Sakamura, K. The Design Approach of BTRON Human Interface, *Trans. Inst. Electronics, Inf. & Comm. Eng. Japan*, J70-D, 11 (1987) (in Japanese).