TECHNICAL REPORT 94-6e

BTRON2 Window System: A Window System Facilitating Cooperation among GUI Applications in Distributed Environments

Noboru Koshizuka

March 1994

DEPARTMENT OF INFORMATION SCIENCE
FACULTY OF SCIENCE, UNIVERSITY OF TOKYO

7-3-1 HONGO, BUNKYO-KU TOKYO, 113 JAPAN

TECHNICAL REPORT 94-6e

BTRON2 Window System: A Window System Facilitating Cooperation among GUI Applications in Distributed Environments

Noboru Koshizuka

March 1994

TECHNICAL REPORT 94-6e

TITLE

BTRON2 Window System: A Window System Facilitating Cooperation among GUI Applications in Distributed Environments

AUTHORS

Noboru Koshizuka

KEY WORDS AND PHRASES

TRON, BTRON, Window System, User Interface, Groupware, Scripting, Distributed Shared Memory

ABSTRACT

The goal of this research is to propose a software architecture of window systems that facilitate cooperative processing among graphical user interface (GUI) applications, and to implement a window system called BTRON2 window system based on the architecture.

Recently, cooperation of GUI applications with other external programs is of much interest because of several reasons. First, the cooperation is necessary for supporting distributed GUI applications in distributed environments, which include Groupware and distributed implementation of GUI applications. Second, cooperation among conventional GUI applications is effective to give them high inter-operability. Application cooperation and tool integration are the topics of this issue. Lastly, dynamic integration of GUI applications with other programs by users increases extensibility and customizability of the GUI applications. This issue includes script processing and assistive technology that dynamically extends or customizes user interfaces according to users' physical characteristics.

Currently, most conventional GUI platforms do not support the cooperation. Previous user interface researches have not clarified what kinds of data/control transfer mechanisms are suitable for the cooperation, nor which layer in the GUI software architecture should handle the mechanisms.

Thus, the first main contribution of this research is to propose a GUI software architecture that facilitates the cooperation among GUI applications. Our architecture exports interaction objects, and allows other programs to access them. For the functionality, it provides structured distributed shared memories, called *Window Real-Objects* (*WROs*) storing interaction objects. Operations to the WROs cause to generate event messages, which transfer control among the GUI applications. Every component shares WROs, via which it communicates with other components in cooperation. Further, this dissertation shows that the distributed shared interaction object architecture can offer more flexible functionality than conventional application cooperation schemes. We examine this issue on multi-user interfaces, script processing, application cooperation, assistive technologies, and distributed implementation of the GUI applications.

The second contribution of this dissertation is to show implementation techniques that assure short and stable response time of GUI software of the architecture implemented upon popular networked workstation environments. The techniques include an R-Tree index of the WRO, its full-replication coherence algorithm assuring constant response time independent of the number of sharing users, and multi-task implementation with a priority-based real-time scheduling scheme. To verify the effectiveness of these techniques, we have designed *BTRON2 window system* and implemented its prototype system upon the ITRON/FILE kernel. We have evaluated using our implementation. In the evaluation, average response time of user's operations with update operations in WROs is about 30 msec, and response-time of read operations is less than 1 msec.

REPORT DATE	WRITTEN LANGUAGE
March 1994	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
143	146

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

Doctor Dissertation, Submitted to The Graduate School of The University of Tokyo in Partial Fulfillment of Requirements of The Doctor of Science in Information Science

DISTRIBUTION STATEMENT

First Issue 0 copies

SUPPLEMENTARY NOTES

This report can be obtained only in the electronic form by anonymous ftp from `ftp.is.s.u-tokyo.ac.jp`

DEPARTMENT OF INFORMATION SCIENCE

Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

Abstract

The goal of this research is to propose a software architecture of window systems that facilitate cooperative processing among graphical user interface (GUI) applications, and to implement a window system called BTRON2 window system based on the architecture.

Recently, cooperation of GUI applications with other external programs is of much interest because of several reasons. First, the cooperation is necessary for supporting distributed GUI applications in distributed environments, which include Groupware and distributed implementation of GUI applications. Second, cooperation among conventional GUI applications is effective to give them high inter-operability. Application cooperation and tool integration are the topics of this issue. Lastly, dynamic integration of GUI applications with other programs by users increases extensibility and customizability of the GUI applications. This issue includes script processing and assistive technology that dynamically extends or customizes user interfaces according to users' physical characteristics.

Currently, most conventional GUI platforms do not support the cooperation. Previous user interface researches have not clarified what kinds of data/control transfer mechanisms are suitable for the cooperation, nor which layer in the GUI software architecture should handle the mechanisms.

Thus, the first main contribution of this research is to propose a GUI software architecture that facilitates the cooperation among GUI applications. Our architecture exports interaction objects, and allows other programs to access them. For the functionality, it provides structured distributed shared memories, called *Window Real-Objects* (*WROs*) storing interaction objects. Operations to the WROs cause to generate event messages, which transfer control among the GUI applications. Every component shares WROs, via which it communicates with other components in cooperation. Further, this dissertation shows that the distributed shared interaction object architecture can offer more flexible functionality than conventional application cooperation schemes. We examine this issue on multi-user interfaces, script processing, application cooperation, assistive technologies, and distributed implementation of the GUI applications.

The second contribution of this dissertation is to show implementation techniques that

assure short and stable response time of GUI software of the architecture implemented upon popular networked workstation environments. The techniques include an R-Tree index of the WRO, its full-replication coherence algorithm assuring constant response time independent of the number of sharing users, and multi-task implementation with a priority-based real-time scheduling scheme. To verify the effectiveness of these techniques, we have designed *BTRON2 window system* and implemented its prototype system upon the ITRON/FILE kernel. We have evaluated using our implementation. In the evaluation, average response time of user's operations with update operations in WROs is about 30 msec, and response-time of read operations is less than 1 msec.

Acknowledgment

This research is owing to cooperation and encouragement of many people. Firstly, I would like to express my highest gratitude to Associate Professor Ken Sakamura of University of Tokyo. He has been my supervisor in these six years. He has given me everything from the "ABC" of researching to his philosophy as a *computer architect*. For example, he has taught me how to carry on my research, how to write papers, how to make a presentation, and, this might be most important, how to be a good computer scientist.

I would like to acknowledge Associate Professor Masami Hagiya, Professor Takashi Masuda, Professor Haruhisa Ishida, Associate Professor Kei Hiraki, and Lecturer Yoshihisa Shinagawa for their fruitful discussions with me, invaluable comments, and adequate suggestions.

I would like to acknowledge Mr. Hiroaki Takada, who is the research assistant of Sakamura Laboratory. Especially, in the early stage of this research, he used to spend much time for discussing with me, and gave me a lot of invaluable suggestions and comments. Also, when I was preparing our papers, he always provided me very detailed and essential comments. Without his guidance, this work could not be accomplished.

Next, I'd like to express my sincere appreciation to Mrs. Yumiko Ohsuga and Miss Kaori Shiomi, who used to be/is the secretaries of Professor Sakamura. They have been caring my daily life in our laboratory, and met my selfish demands any time.

My special thanks also goes to my colleague in our laboratory. Eric E. Bowles always proofread my papers in detail. Without his proofreading, most of my papers could not receive good evaluation from foreign researchers. Also, I'd like to thank all of my co-workers who are/used to be members of the BTRON Group in our laboratory, Naoki Inoue, Norio Kimura, Michimasa Uematsu, Reiko Kondo, Tomoyuki Tanuma, and Hiroaki Maezawa. They mainly contributed to the implementation of the Enableware Specification of BTRON and the network system of PC 9340. Moreover, many discussions with them have made the foundation of my research. Also, I'd like to thank Aki Watanabe, Hirohisa Mori, Tadashi Takeuchi, Hisahiro Kobayashi, Yasuharu Sasaki, Kazunori Kawamura, and Tetsuya "tion" Yamane. In this year, I devoted myself only to my doctoral dissertation. So, I must be blamed for imposing much heavy non-research work to them. I'd like to express my appreciation to all of them.

I'd like to appreciate all the members of TRON Association, BTRON Technical Committee, and TRON Human-Machine Interface Research Group. Fruitful discussions with these members have formed my fundamental attitude to user interfaces and operating systems.

I bless my stars that I was surrounded by good friends, especially, wonderful folk of our Donkey Tennis Team. Through playing tennis with them, I always refreshed myself even when I was downhearted. Also, I'd like to thank Hironobu Gotoda. He is only the classmate of this department who has prepared the doctoral dissertation at the same time. I appreciate his frequent and timely encouragements for me.

Finally, I'd like to thank my parents, Shohtaro Koshizuka and Masako Koshizuka, for assisting me both mentally and economically all through this research. And, I'd like to thank my only brother, Seiichi Koshizuka. When I was a high school student, he guided me to the world of science and engineering. Also, he strongly recommended me to major in computer science many years ago. So, my attitude for thinking, investigating, and researching owes a lot to him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"I believe that users want the sense of their own accomplishment rather than to admire a magically smart, intelligent, or expert system. Users want to be empowered by technology to be able to apply their knowledge and experience to make judgments that lead to improved job performance and greater personal satisfaction. ... Therefore, I chose to emphasize the User Interface (UI) in place of Artificial Intelligence (AI)..."*
>
> [Shneiderman, 90][128]

1.1 Motivation

With the spread of low-cost but high-performance workstations or personal computers, the extent of computer users and applications becomes larger. In present, computers are used not only by experienced expert users but also by various classes of users, such as novice users, children, the aged, and the physically disabled. Furthermore, computers are used not only for scientific computations but also for supporting every aspect of human activity, such as communication, education, control, and office work. Accordingly, the significance of user interface is becoming larger because good user interface increases productivity in several human activities [37].

Since the latter half of the 1970's, graphical user interface (GUI) has revolutionized the way we interact with computers, and is widely accepted by workstations and personal computers. *Window systems* are the most popular and fundamental run-time systems that support the development and execution of GUIs. Their first generation is a group of window systems developed innovatively in Xerox PARC. They are developed as a part of a programming language environment such as Smalltalk [36] and D-Lisp. The second generation is just a class of window systems built upon commercial personal computers such as Star [130], Lisa, Macintosh [5], and BTRON1 [111]. Window systems of the third generation are network transparent window systems such as the X-Window [126], SUN NeWS [137], NeXT Step [145], and GMW [52].

As a lot of user interface researchers have pointed out, the most serious problem of GUIs is the difficulty of their development [84, 14, 87, 86, 92, 87]. Since the latter half of the 1980's, higher-level platforms such as GUI toolkits and user interface management systems (UIMSs) have been studied and developed actively in order to support GUI development [41, 85, 101, 44]. Many interactive applications with GUI (GUI applications) have been built with the help of these high-level platforms.

These tools mainly support the modularization by layering. They separate a GUI application into a user interface module and an application module, which transfer data and control by function calls and callback. These modules are tightly-coupled and statically linked into a single process. Thus, this modularization is transparent from end-users. The only module separation visible from end-users is the separation between a window server and its client.

Thus, this architecture prevents end-users to modify structure of GUI applications. For example, when using a word processor, a user may sometimes want to replace its built-in spell checker with more sophisticated one. However, the user cannot replace it because the spell checker is statically linked with the word processor.

This is a well-known problem of the GUI application as the *inflexibility* or the *low tailorbility* [131, 96, 142]. Smith and Susser have represented this architecture as a *monolithic* program [131]. Clément called similar architectures of software development environments as a *tightly-coupled* architecture [20].

GUI application developers are currently faced with limitations of the monolithic architecture in many aspects. Here, we point out two limitations. Firstly, although recent active researches have realized several advanced human-computer interaction (HCI) techniques, some of them cannot be implemented straightforwardly upon the conventional monolithic GUI architectures. This is because they essentially require cooperation among multiple programs. Multi-user interfaces, computer augmented environments, scripting, and assistive technologies are among them.

- *Multi-user interface* is a basic technology for real-time groupware [49] that enables multiple users to use the same application programs for collaboration. For example, it provides functionality to allow multiple users to edit the same document concurrently from different machines. The multi-user interface requires multiple user interface programs on different machines to cooperate with each other.

- *Computer Augmented Environment*[1] is a computerized environment that supports several aspects of human life [35]. User interface software in the environment must interface between human and several computerized intelligent objects connected with computer networks. Thus, an interactive application should be a distributed application that consists of multiple cooperative processes dispersed in a distributed environment.

- *Scripting* is an HCI technique that provides batch type interaction with interactive GUI applications. This technique is for supplementing a facet of desktop GUI environments that they are poorer than old-fashioned command language interfaces at repeating a fixed operation sequence in many times. For the scripting, script language interpreter processes must be able to drive multiple GUI application programs, , and, therefore, a cooperation mechanism between the interpreter and GUI applications is indispensable.

- *Assistive technologies* include some techniques which require GUI applications to cooperate with another external program. A popular example of these is an application-independent tool that reads up the information displayed on a GUI window with speech sound for visually disabled users [144, 143, 61, 89]. For its realization, the tool must be able to obtain the information on a screen by communicating with GUI applications.

Fundamentally, the monolithic GUI architecture gives no support for the cooperation necessary for these advanced techniques. Currently, the cooperation and communication necessary for each advanced HCI technique are supplemented by individual toolkit upon the architecture. However, these toolkits are insufficient because they are costumed only for specific techniques and, furthermore, give no abstraction or framework to integrate or to combine these advanced techniques. For example, groupware toolkits [34, 98, 25] support only groupware, and assistive technique toolkits [88, 89] support only some class of assistive techniques. In most cases, we cannot use these toolkits together to implement "*groupware with assistive techniques*."

Secondly, another problem caused by the monolithic architecture is the large size of each GUI application. Currently, a GUI application provides all functions necessary for some task. For example, a word processor provides text editing function, text formatting function, file load/save function, printing function, spell check function, and so on. Thus, both the size and the complexity of practical GUI software applications are becoming

[1]There are synonyms such as *ubiquitous computing*, *augmented reality*, *intelligent environment*, and *computerized environment*.

larger and larger. "*The functionality of applications written for personal computers is approaching the limits that current software technology can support*." This is the main cause of "*the coming software crisis*." [131]

Conventionally, user interface software allows to combine multiple layered component from different sources. For example, one can built a graphic editor upon Sun OS from Sun Microsystems, the X-Window system from MIT and the InterViews toolkit from Stanford University.

Nevertheless, this is insufficient. What is required is the combination among functionally decomposed components. For example, we want to build a word processor by combining a text editing component, a printing component, a formatting component, a spell checking component, and so on, each of which is developed by a different software vendor.

Through this examination, we have found that the key technology common with these problems is the cooperation facility among multiple GUI software components each of which provide a small piece of functionality. The facility includes the way to decompose GUI applications into multiple components, mechanism to communicate and synchronize among the components, and software architecture for the facilities.

1.2 Objectives

Objectives of this dissertation consists of three issues as follows:

1. The first objective is to propose an architecture of GUI applications that facilitates the cooperation among GUI applications.

2. The second objective is to build a window system based on the architecture. As its design issues, we will clarify its application programming interfaces. As its implementation issues, we will clarify efficient implementation techniques upon current popular computing environments.

3. The last objective is to apply this architecture to the advanced HCI techniques, and to compare it with other mechanisms for each technique.

1.3 Background of the TRON Project

This research is being carried out in the TRON (The Real-time Operating system Nucleus) Project. The ultimate goal of the TRON Project is to build special distributed environments, called *Highly Functionally Distributed Systems* (HFDSs) [123, 117, 114].

The HFDS is a heterogeneous distributed environment consisting of variety of computers: embedded micro computers equipped with ITRON (Industrial TRON) [113, 119], personal computers or workstations equipped with BTRON (Business TRON) [111, 118], and large data/communication server machines equipped with CTRON (Centeral/Communication TRON) [120]. These computers are connected one another and also with sensors and actuators. Its prototype systems are the TRON-concept Computerized House [112] and the TRON-concept Intelligent Building [117].

This research is a work on *BTRON2* (Business TRON ver. 2) [118, 115], which provides interfaces between human and the HFDS. Thus, the window system this dissertation presents is called *BTRON2 window system*.

Currently, BTRON has two versions: BTRON1 [111] and BTRON2 [118]. BTRON1 is the one for machines with rather restricted hardware resources, and has already been available commercially. BTRON2 is the one for machines with TRON Specification VLSI CPU [116] and other rich hardware resources. BTRON1 and BTRON2 are based on a common human-machine interface specification called *TRON Human-Machine Interface Specification* [111, 121, 68].

The TRON Project has also been emphasizing the importance of the new HCI techniques referred in the previous section by proposing their concrete concept or by building specifications of them. This is because they are indispensable for realization of the HFDS.

For example, the ubiquitous computing is the main concept of the HFDS, and is the ultimate goal of the TRON Project. Assistive technologies are necessary technologies for building HFDS, and we have been researching them actively (*EnableWare*) [109, 61, 60]. As the user's operation model in the HFDS, Sakamura proposes a "task delegation model" [110] [2], which includes aspects of groupware and scripting. The role of this research in the TRON Project is to clarify architectures and mechanisms to realize these ideas of HCI for the HFDS.

1.4 Contribution

The main contribution of this dissertation is to propose an architecture of GUI applications with distribute shared memories for the communication and cooperation model among GUI applications. The shared entities via the shared memory are static data of interaction objects, such as control parts, segmented graphical objects, file icons, and so on. In this architecture, a GUI application consists of multiple independent components sharing the

[2] In Japanese, it is called as "Omawasi".

shared memory.

Next, we will present design and implementation of the BTRON2 window system based on the architecture. It provides distributed shared memories that contain interaction objects, which is called *Window Real-Object* (WRO). A GUI application consists of tasks for GUI (*user-agent*) and tasks for semantic actions (*application core units*). The WRO is shared by the tasks and facilitates the cooperation among them. The WRO is just an "*interface bus*" among the tasks.

WROs are tailored especially for GUI applications, and provide several functions for GUI as follows:

1. asynchronous event messages each of which notifies an update in a WRO (*abstract events*)

2. facility supporting access patterns to interaction objects in GUI applications (*spatial addressing*)

3. attributes of stored data that support flexible data sharing (*lock* and *locality* attributes)

4. facility for sharing drawing pattern data among interaction objects

For its efficient implementation, we will propose implementation techniques. An *R-tree* index, a variation of *fully-replicated* coherence algorithms, and multitask implementation with real-time scheduling algorithm are among them.

Lastly, we will apply this architecture to some advanced HCI techniques. Especially, this dissertation will examine multi-user interface and scripting precisely. In implementing each technique, our architecture offers several novel features as follows:

- multi-user interface:

  The BTRON2 window system can support multi-user interface as flexible as conventional groupware toolkits. Additionally, its framework is independent of the application semantics.

- scripting aspect:

  In our architecture, application programmers need not embed codes for script processing interfaces in GUI applications. The user interface and the script processing interface are common.

1.5 Outline of This Dissertation

The rest of this dissertation is organized as follows. Chapter 2 classifies general software architecture of user interface. More precise description and comparison are made in each subsequent chapter that gives precise discussions of the issue.

Chapter 3 makes an overview of the GUI architecture proposed in this dissertation, which is the overall architecture of the BTRON2 window system.

Chapter 4, 5, and 6 precisely describe design and implementation issues of three main components. Chapter 4 describes Window Real-Objects, Chapter 5 describes User-Agents, and Chapter 6 describes Application Core Units.

In Chapter 7 and 8, we examine applicability of our architecture to two important advanced HCI techniques. Chapter 7 examines its multi-user interface aspects. Chapter 8 shows script processing facility in the architecture.

Chapter 9 describes precise implementation information of the prototype system of the BTRON2 window system, and reports results of its performance evaluations. Lastly, Chapter 10 concludes this dissertation, and gives the research direction of the future.

Chapter 2

Background

How to organize user interface software is a significant problem. This is because the amount of the program code of user interface is very large in many interactive application programs. For example, Borrows et. al. have reported in 1986 that, in developing an expert system, about 40% of program code of their system was taken up its user interface facility [84]. Myers and Rosson have reported in 1992 that about 40–60% of program code is used for the user interface [87]. From these reports, the implementation framework of GUI has the most great impact with the total software structure. With a little exaggeration, software architecture of interactive GUI applications is nearly equivalent to software architecture of GUI.

There have been several GUI platforms with various architectures. This chapter first presents our taxonomy of GUI software, then makes an extensible survey of software architectures of user interface and other concerned areas on the basis of the taxonomy. On the other hand, in the field of user interface software research, there are no strict terminologies to refer components or function in the architecture. Therefore, to avoid unnecessary confusion, we, next, define terminology of user interface architecture.

2.1 Architectural Dimensions of GUI Software

We can find three orthogonal dimensions of the way to separate GUI software into components. They are (1) *abstraction* dimension, (2) *data* dimension, and (3) *operation* dimension.

According to *abstraction* of functionality, we can divide GUI software into several layered components: e.g., device drivers, operating systems, window systems, toolkits, application bodies, and so on. Next, we can divide GUI software by *data* unit. In this case, each component manages only the data unit that it is applied to. Each component contains all operations for the data unit. The most popular data decomposition is the

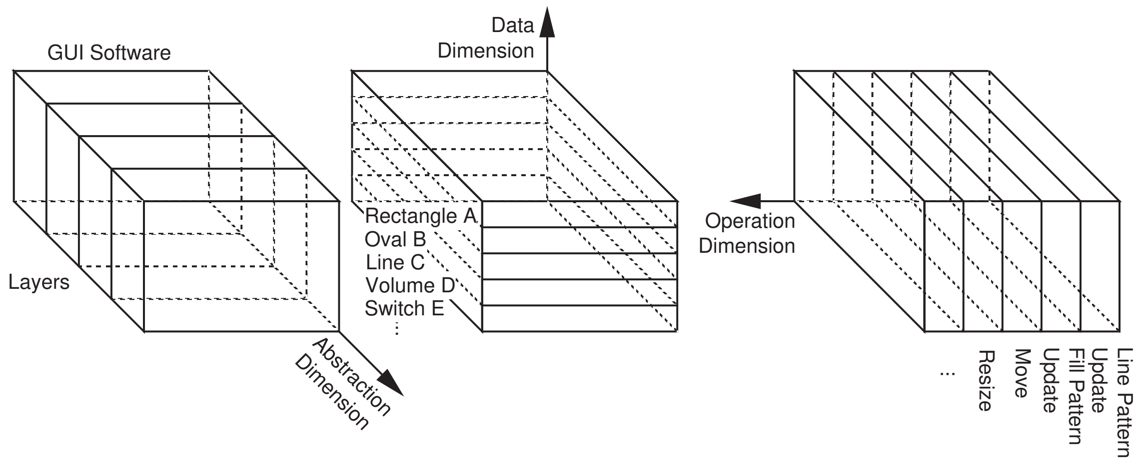


Figure 2.1: Architectural dimensions of GUI software

object-based/object-oriented GUI architecture. Lastly, we can divide a GUI software by *operation*. Each component provides an operation that can be applied commonly to all kinds of data unit. In this case, data is represented in a standard data format and shared by every component.

For example, we can decompose a graphic editor by the three dimensions as follows. By the abstraction dimension, we can decompose it into window system component, toolkit component, and editor application component. By the data dimension, we can decompose into components for rectangle segment **A**, oval segment **B**, and line segment **C**, etc. By the operation dimension, we can separate it into a line-pattern update component, a fill-pattern update component, a move component, a resize component, etc (Figure 2.1).

2.2 Modularization by Abstraction

Most user interface software architecture models and conventional GUI tools are based on modularization by abstraction. For example, window systems, toolkits, and UIMSs work as a layered component. This section describes three major layering models.

First, Seeheim model [100, 38] is the first and the most famous architecture model of user interface management systems (UIMSs). The model separates user interface software into three components: *presentation component*, *dialogue control component*, and *application interface model* (Figure 2.2).

In the model, *the presentation component* contains device-dependent details and specifics of displays as well as interaction style descriptions. *The dialogue control component* executes dialogue processing and sequencing. *The application interface*

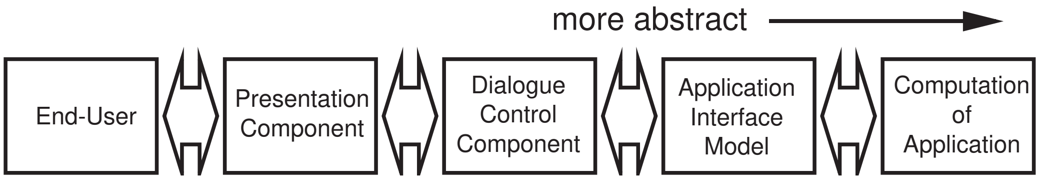


Figure 2.2: Seeheim model

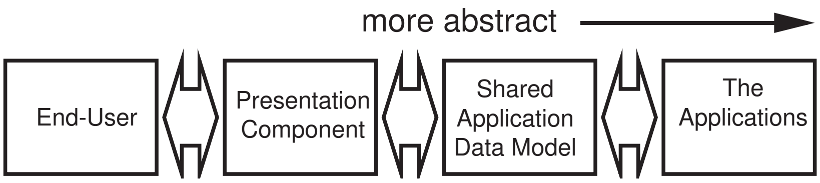


Figure 2.3: Hudson's user interface software model

model contains the application view of the interface and the interface view of the application.

Hudson has argued that a model without an explicit dialogue component is more appropriate for direct manipulation user interface software because syntax is being minimized in the direct manipulation [45]. Additionally, he has proposed making the *application interface model* of Seeheim model a set of active shared objects (Figure 2.3). On the basis of the model, a UIMS, called *HIGGENS*, has been developed.

Bass and Coutaz have also given a layer model of user interface software architectures [8]. Compared with the Seeheim model, it handles lower layers of user interface software more precisely. In the model, user interface software consists of five layers: *Device Drivers*, *Resource Managers*, *Interaction Objects*, *Dialogue Control*, and *Functional Core* (Figure 2.4).

In their model, *the device drivers* control physical devices. For example, they

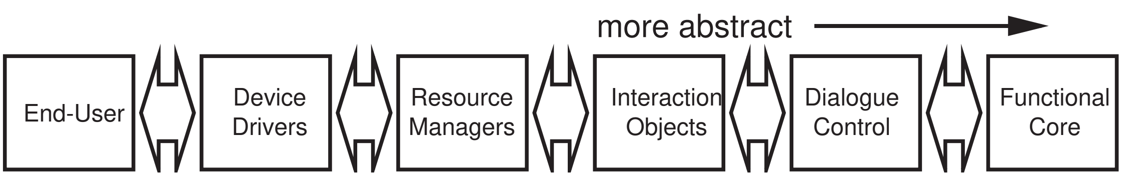


Figure 2.4: Architectural model of Bass and Coutaz

send pixel-level instructions to display the cursor in the desired position. *The resource managers* control the actual resources involved in the interactions. It manages the physical device's resources and provides the abstractions used for interaction. The most common resource manager is *window system*. *The interaction objects* are entities that the user can perceive and manipulate with physical interaction devices such as pointing devices or keyboards. *The dialogue control components* control the media used and the sequencing of user interactions. They also control the style of the interactions. For example, they control whether menus or command lines are used. *The functional core components* implement application-dependent domain knowledge. The functional core should, ideally, remain totally ignorant of the way that its data structures and functions are exposed to the user via interaction objects.

2.3 Modularization by Data or Operation

Currently, there are several user interface systems based on the modularization by data or operations. However, their architectures are not generalized as explicit software architecture models. This section makes a brief survey of concrete conventional user interface systems modularized by data or operations.

The object-based or object-oriented programming is a very general framework which is based on the modularization by data. The pioneer of special frameworks for user interface software in the object-oriented programming is the *Model-View-Control* (MVC) model [36]. It decomposes a GUI application program into a set of MVC object tuples, which are interactive objects with users. The model object implements semantic data. The view object implements output of the model data to users, and the control object implements its inputs from users. In other words, the view and control objects correspond to the interaction object layer, and the model object corresponds to the semantic layer. Mostly, the MVC tuple is not implemented using processes. This restricts users to combine MVC tuples dynamically. The conventional MVC modularization is only for developers, not for end-users.

Smith and Susser have developed the *Component Construction Kit* (CCK) framework, upon which end-users can compose a GUI application from several GUI application components such as paragraph component, video component, spreadsheet component, button component, and spelling checker component [131]. Each component has input ports and output ports through which application data are transferred. Each port corresponds with syntactic information and semantic information of the data. A user can dynamically connect an input port and an output port with the same information (Figure

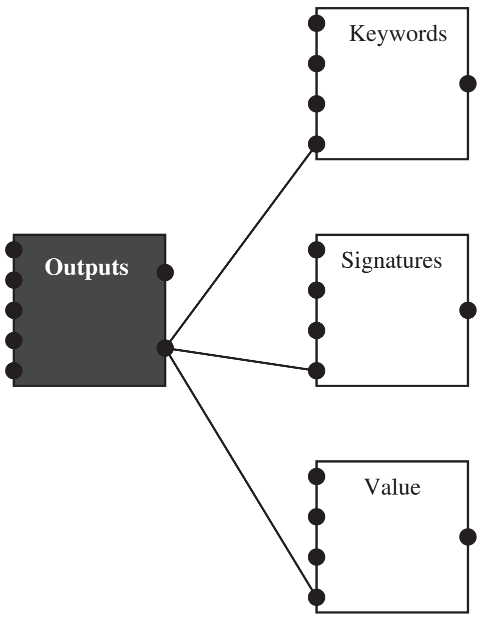


Figure 2.5: The Comopent Construction Kit framework

2.5).

Dewan et. al. have developed the *Suite* user interface software, which provides a cooperative framework similar to the CCK [22]. In Suite, each interactive application is programmed as a persistent object in a distributed environment. Additionally, users can compose these objects dynamically. Programmers can define remote procedures in each object, which other external objects can call them. The Suite framework is very general. When multiple independent objects developed by different vendors cooperate with each other, standard protocol and data format among them are necessary.

Macintosh system 7.0 provides a facility for the application cooperation, which is the *Apple Event Manager* [7]. It facilitates more abstract events than conventional window events, called *Apple Events*. The Apple Events are merged with the window events, and used as inter application communication messages. Thus, the functionality of the Apple Event Manager is just a typed asynchronous message passing. Application programmers can also implement the server-client model upon the facility. The main feature is that it defines standard event types every application should support. This standardization assures minimum compatibility among multiple application components from different sources. Each application can define its own Apple Events, which are used for more flexible and powerful application cooperation.

The concepts of the *tool integration* or *open CASE platforms* in the research are of

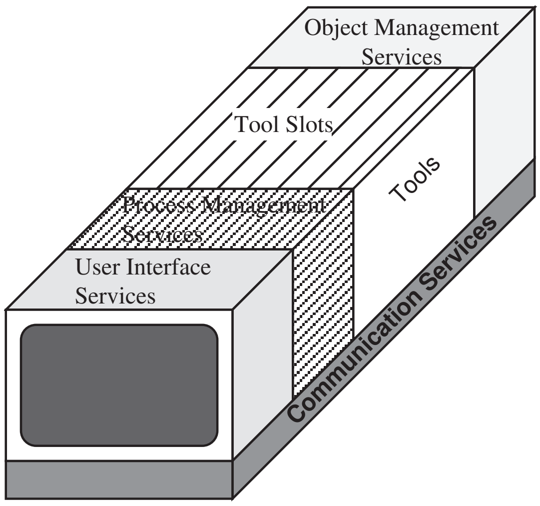


Figure 2.6: Architectural model of tool integration

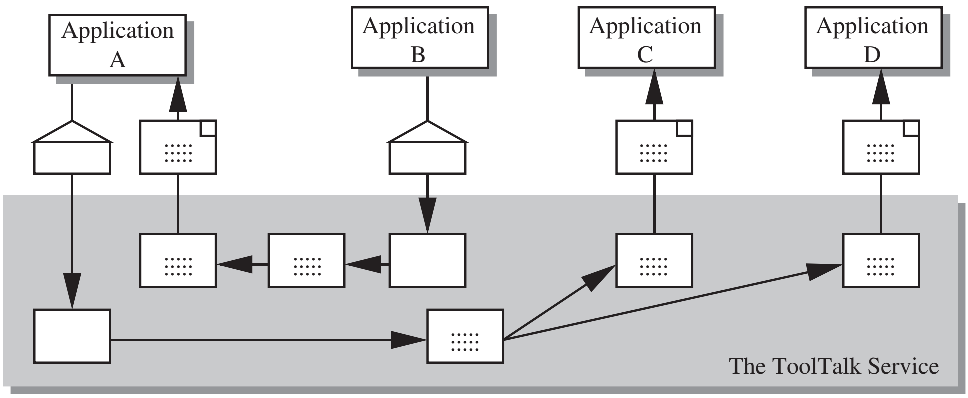


Figure 2.7: Architectural model of ToolTalk

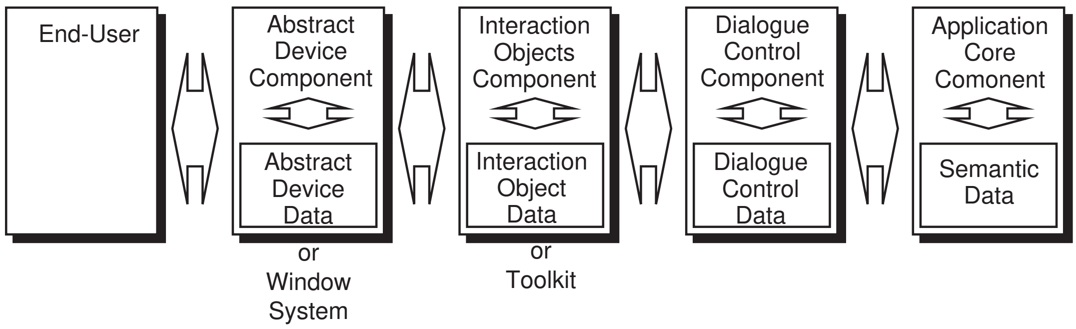


Figure 2.8: Architectural model of in this dissertation

Computer Aided Software Engineering (CASE) is similar to the cooperative architecture of user interface software (Figure 2.6) [2]. It addresses the cooperation of many tools, from many sources, in a uniform but open framework. Three key aspects of the tool integration are identified as *data* integration, *control* integration, and *user interface* integration. As for data integration, each tool shares common database of software products which are currently developed using the CASE tool. As for control integration, a standard message passing protocol among multiple CASE tools are defined. Some toolkits support this protocol: *BMS* [31] and *ToolTalk* (Figure 2.7) [138, 51] are among them. These tools support the implementation of the inter-tool communication, and are similar to Apple Events.

2.4 Layered Model of GUI Software Architecture in This Dissertation

This section illustrates our general GUI architecture model, which is referred frequently in this dissertation. On the basis of the model, we refer specific function of software with GUI or compare architecture of several GUI systems. In our model, an interactive software program is separated into four components by the abstraction level: *Abstract Device Component*, *Interaction Object Component*, *Dialogue Control Component*, and *Application Core Component*.

The *Abstract Device Component* provides device independent programming interface (abstract device) of windows. It is equivalent to the presentation component of the Seeheim Model, and to the Resource Managers of the Bass and Countaz's Model. In practical systems, the component corresponds to window servers such as the X-Server.

The *Interaction Object Component* is a collection of graphical objects used for the interaction among users and application programs. Here, interaction objects include control parts such as volumes, button switches, icons, menus, and graphic segments. The Interaction Object Component manages the data structure of a set of interaction objects, and their behavior. It is just the lower level of the Dialogue Control Component in the Seeheim Model and the Interaction Objects in the Bass and Countaz's Model. In practical system, the component corresponds to GUI toolkits such as the X-Toolkits, the InterViews toolkits, and the MacApp toolkits.

The *Dialogue Control Component* manages the relationship among interaction objects and controls operation sequences of user interfaces. For example, when a user presses an OK button on a window, a more detailed setting panel appears. This sequence is controlled by the Dialogue Control Component. It is equivalent with the higher level of the Dialogue Control Component in the Seeheim Model and the Dialogue Control in the Bass and Countaz's Model. In practical systems, the component is contained by UIMSs.

The *Application Core Component* is just the body of GUI applications. It corresponds to both the Application Interface Model and Computation of Application in the Seeheim Model, and to Functional Core in the Bass and Countaz's Model.

Next, we define a few terminologies used throughout in this dissertation. The term *GUI software* means all the components of the interactive software with GUI. A *GUI platform* (or simply a *platform*) means a runtime module shared by application programs with GUI. For example, window systems, toolkits, and UIMSs are among them. *GUI application* (or simply *application*, *application software*) means a part of GUI software except the GUI platforms.

Chapter 3

BTRON2 GUI Architecture

This chapter describes an architecture of GUI software on BTRON2 [58, 118, 59, 65].

3.1 Introduction

In this chapter, we propose a new GUI architecture that facilitates cooperation among GUI application components. There are two important issues for the cooperation. First, end-users can control the combination of the components. Second, the combination should be open so that components from different sources can cooperate with each other via standard cooperation facility. For example, an end-user can compose a word processor with a text editing component from **A** Corporation, a printing component from **B** Corporation, and a spell checking component from **C** Corporation.

This architecture increases availability of applications that provide a small piece of functionality: an application that provides only printing function becomes useful by combining with another editing component and text formatting component. Furthermore, this architecture makes GUI applications flexible so that end-users can extend, modify, and tailor them as they prefer.

The above cooperation requires a standard data/control transfer mechanism and a standard synchronization mechanism among the components. For the purpose, our architecture provides distributed shared memories. Data communicated via the memories are interaction objects such as segmented graphic data, control parts, and file icons. The TRON Application Databus (TAD) specification [111] defines the standard data format of the interaction object data.

Furthermore, the combination among the GUI components can realize advanced GUI techniques that monolithic GUI architecture cannot implement, such as multi-user interface, script processing, assistive technologies, and distributed GUI applications.

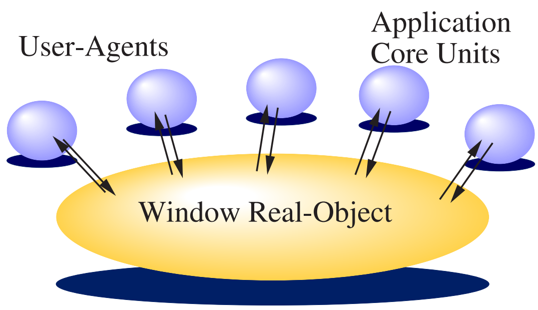


Figure 3.1: BTRON2 GUI architecture model

3.2 Overall Architecture

In our architecture, a GUI application consists of distributed shared memories called *Window Real-Objects* (WROs) and multiple concurrent executing units sharing the WROs. Further, these units are classified into *user-agent*s and *application core* units on the basis of their roles in the software (Figure 3.1).

A **Window Real-Object (WRO)** is a distributed shared memory (DSM) provided in the BTRON2 window system as an abstraction of a GUI window, which exports interaction object data to other processes. This facilitates cooperation among GUI applications, and consequently, as discussed below, supports distributed programming of GUI applications, multi-user interface, etc [65]. A window on a screen is a view of a WRO from a user, and as many WROs as windows or panels on a screen are created in a machine.

WROs provide a facility to share interaction objects and also to transfer control among the components across multiple machines. *Abstract event* is the control transfer mechanism of WROs. An abstract event is an asynchronous message from a WRO to components, each of which is activated when it receives the abstract event.

A **user-agent** provides human-computer interface common to GUI software. It corresponds with abstract device component and a part of interaction object component of our GUI software architecture. It visualizes interaction objects stored in a WRO, and conversely applies users' operations to the objects. WROs hold instance information of the interaction objects such as the number, the kinds, the location, and the attributes of interaction objects. And, each user-agent should be able to visualize and to operate the objects.

The abstract device component portion in the user-agent is called *window primitives*.

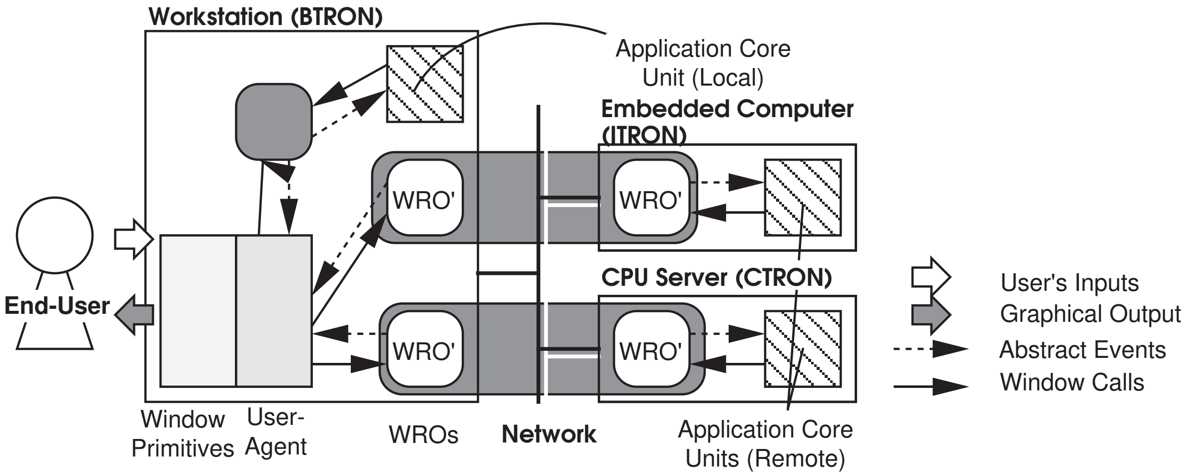


Figure 3.2: An example of BTRON2 GUI architecture

They simply draw objects requested by the user-agent and send it physical events notifying users' operations on input devices. The physical event is a device-dependent and low-level event notifying, for example, "a button on a pointing device is pressed" or "the 0x3b key is pressed."

An **application core unit** corresponds with the application core component of our GUI software architecture model. It realizes similar functions to the one of callback routines of conventional GUI toolkits. The application core unit interacts with users via WROs. It can output information to users by storing or updating interaction objects in the WROs. Conversely, it can get user's operations by retrieving data from the WROs or by receiving abstract events.

Here, we show an example of total GUI architecture. Usually, multiple GUI applications are running on a workstation or a personal computer with multiple windows. The whole GUI architecture of the machine is organized as Figure 3.2. In the example, the BTRON contains three WROs, each of which corresponds with a window. One WRO is managed by an application core unit executed on the local machine. Other two WROs are managed by remote application core units: one is running on an ITRON machine and another is on a CTRON machine.

In this dissertation, we refer to the WROs and the user-agent as *BTRON2 window system*. They are provided with the BTRON2 kernel as a part of its operating system. Application developers must program only the application core units.

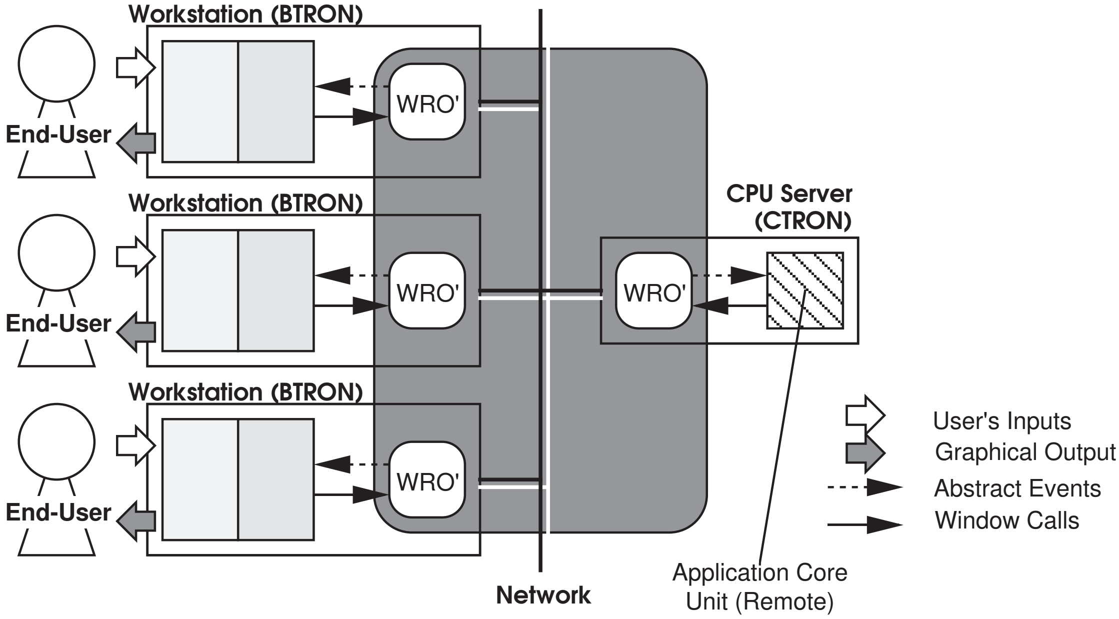


Figure 3.3: An example of multi-user interface architecture of BTRON2

3.3 Dynamic Architectural Configurations

Basically, the main feature of the BTRON2 GUI architecture is to export interaction objects to external programs. This architecture supports cooperation among GUI applications. This flexible cooperation facility can support very wide range of human-computer interaction styles. This section exemplifies the flexibility by the following four examples: (1) multi-user interface, (2) scripting in GUI environments, (3) auditory user interfaces for the blind (assistive technology), and (4) distributed implementation of GUI applications.

3.3.1 Multi-user Interface Configuration

Multi-user interface is a kind of user interface that allows multiple users to collaborate on computers in realtime [29]. Often, real-time groupware [48, 29] provides the multi-user interface, and a multi-user editor [29] is the typical example of such groupware.

Our architecture can adapt to multi-user interface easily. Multiple user-agents, each of which runs on a different machine, and an application core of the editor share a WRO. Figure 3.3 describes an example of the multi-user editor architecture in which three users are using the multi-user editor. When a user modifies the shared document, the user-agent updates the interaction object data in a WRO. Then, abstract events notify the update to all user-agents, which update screens. This update propagation protocol is described in

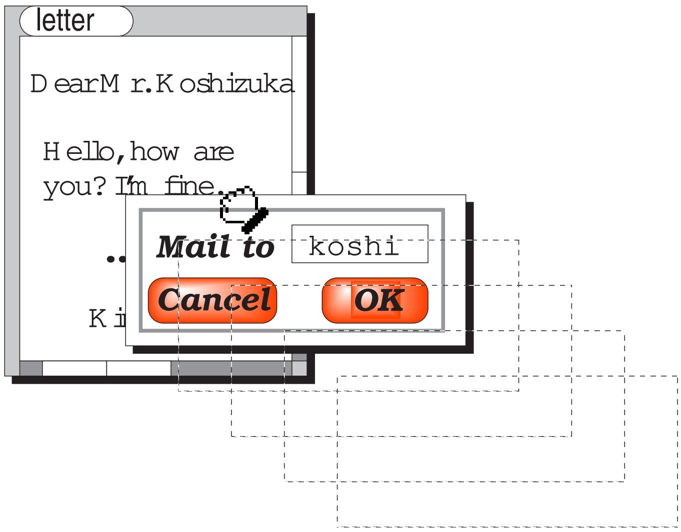


Figure 3.4: Post-it metaphor

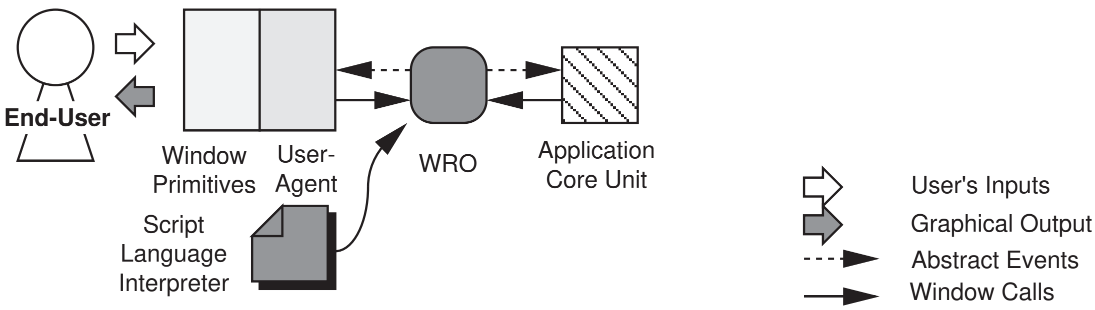


Figure 3.5: An example of batch-type user interface architecture of BTRON2

Chapter 4 in detail.

3.3.2 Scripting Configuration

When a user repeats the same operation sequence, script processing is more suitable than interactive processing. Sakamura has proposed "*Post-it metaphor*" [110] as a metaphor of script processing, which is suitable for GUI environments. In the metaphor, a target data of the script processing is considered as a document, and a script program is considered as a post-it on which instructions about the document are written. To invoke the script program, the user attaches the small post-it window on the document window, and pushes an OK button on it (Figure 3.5).

Our architecture can easily realize this technique as follows. When the post-it window

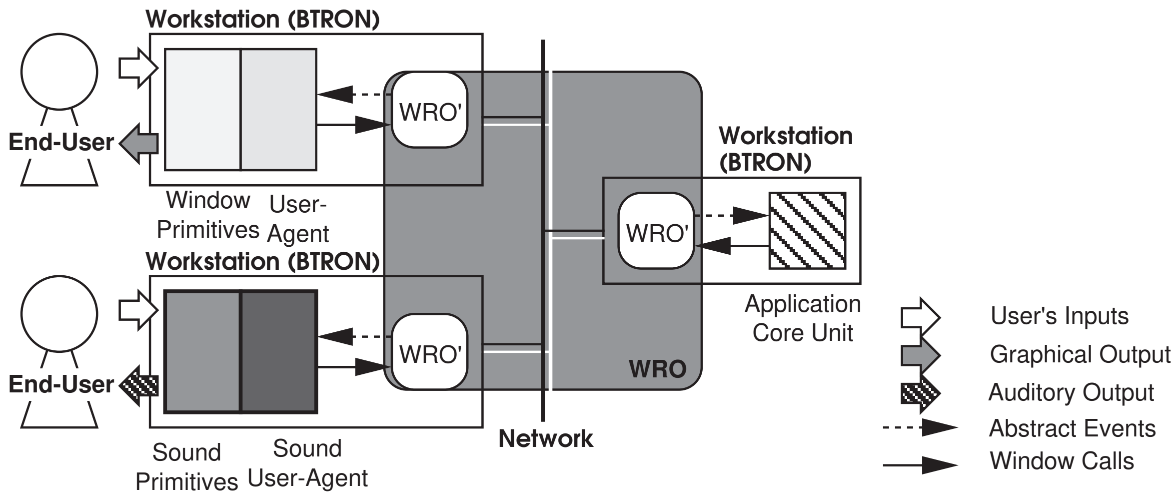


Figure 3.6: An example of auditory user interface architecture of BTRON2

is attached on a document window, a script program interpreter is launched and shares the WRO of the document window. Then, when the user push the execution button, the interpreter starts running (Figure 3.5).

3.3.3 Auditory Interface Configuration

One of the most important problems of GUI is that the visually disabled cannot use it. If GUI is used as the standard human-machine interface of computer systems, it must be able to adapt to the visually disabled [61].

The BTRON2 window system can adapt to visually disabled users by substituting the user-agent with *auditory user-agent*. Ordinary user-agents which provide GUIs are considered as *graphical* user-agents, which visualize the information in WROs. On the other hand, the auditory user-agent translates the graphical information of WROs into text information, and outputs it with speech sound.

The point is that the substitution is independent of application core units, and, therefore, all applications built on WROs can be operated by both graphical and auditory HMIs. Figure 3.6 is a sample organization of groupware components, via which a well-eyed user and a visually disabled user are collaborating with each other.

3.3.4 Configuration for Distributed GUI Application

In the BTRON2 GUI architecture, GUI applications can be decomposed into several independent execution units. Each unit can run in parallel and on a suitable machine for its execution. For example, file accessing units run on a file server machine, units

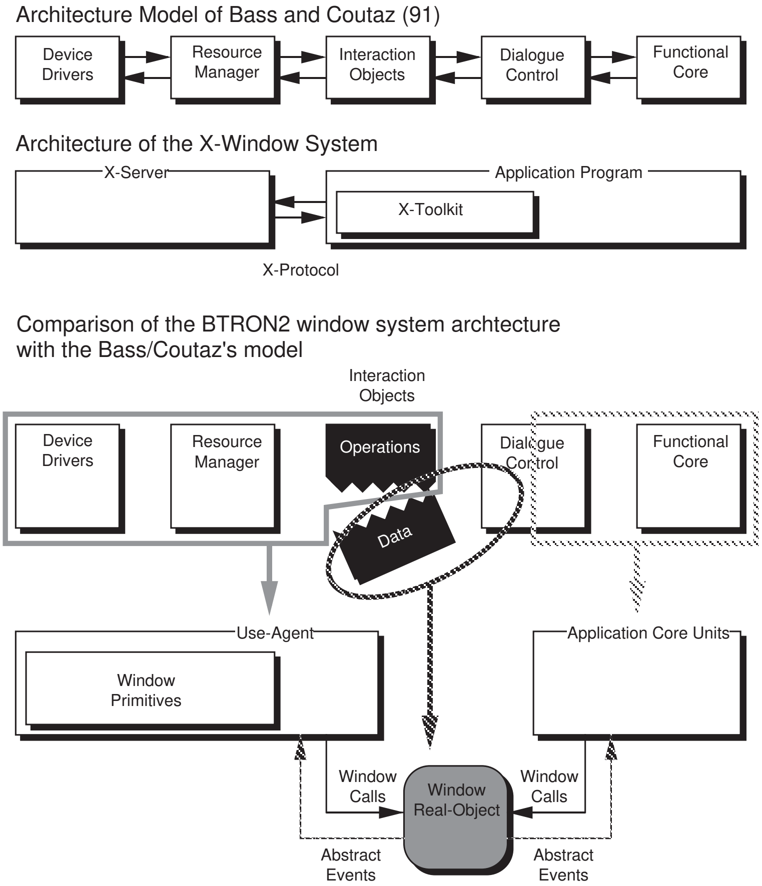


Figure 3.7: Comparison with general user interface software architecture model

programmed in Lisp run on a Lisp machine, and so on. This distributed processing can be effective for increasing performance of the GUI software.

3.4 Comparison with Other GUI Architecture Models

This section compares the software architecture of the BTRON2 window system with other user interface software architecture.

Seeheim Model [100, 38] is the first and the most popular user interface software architecture model. Comparing the BTRON2 window system and the Seeheim Model, the user-agent and the WROs correspond to the *presentation component*, and the application core units correspond to the *application interface model* and the *computation of application*. The *dialogue control* of the BTRON2 window system is handled by the application core units and the control transfer using abstract events among application core units and user-agents.

Next, we compare our architecture with the architecture model proposed by Bass and Coutaz [8]. The user-agent implements functions of the *device drivers* layer and the *resource manager* layer in their model. Functions of the *interaction objects* are implemented with both the user-agent and the WROs. The user-agent implements the code for actions of interaction objects, and the WROs manage their data structures. Function of the *dialogue control* layer is realized by the application core units and the control transfer among application core units and user-agents (Figure 3.7).

Our modularization policy is different from the one of the object-oriented model. The object-oriented model always considers the data structures and the operations in the same module, called an *object*. Our model separates the data structures and the operations into different components. This reflects our requirement for user interface. That is, "the view and the action of each interaction object should depend on end-user's characteristics." For example, value of a volume part should be displayed graphically to well-eyed users, and, to blind users, it should be displayed with sound.

Furthermore, in the multi-user interface configuration, multiple end-users with different characteristics should be able to share the same interaction objects. In this case, multiple different operation sets should be applied to the same interaction object. For example, when a blind user and a well-eyed user collaborate with groupware, the state of a volume part should be represented with graphics and sounds simultaneously. In this case, the user-agent of the well-eyed user applies an operation set with graphical output to the volume parts, and the user-agent of the blind user applies an operation set with auditory output.

3.5 Summary

This chapter has described the overall architecture of interactive software built upon the BTRON2 window system. It provides an abstraction of distributed shared memories called Window Real-Objects (WROs) as the programming interface of GUI windows. A WRO works as an interface between a GUI component and an application component. The novel feature of the architecture is that the WRO can work as an interface among application components and also among GUI components.

Next, this chapter has shown that the architecture can realize flexibly several advanced HCI techniques by changing the combination among GUI application components.

- A user can build *multi-user interfaces* by connecting multiple user-agents on different machines into a WRO.

- A user can build *distributed GUI applications* by connecting multiple application core units on different machines into a WRO.
- A user can make *script processing* by connecting script interpreter component into a WRO.
- A user can provide an *application-independent screen reading tool* by connecting a screen reading component into a WRO.

Chapter 4

Window Real-Objects

4.1 Introduction

A *Window Real-Object* (WRO) [1] is a structured distributed shared memory (SDSMs) that is an abstraction of a GUI window. It specifies the standard procedural interface for the communication among user-agents and application core units, which is common to a user-agent and an application core unit. The WRO represents the whole two-dimensional space that can be accessed by users through a window. Conversely, the window is the view through which users see a part of contents of the WRO.

There are advantages to use a shared memory abstraction in GUI software. First, it is intuitively natural because an end-user sees a window as a storage of graphical objects. Second, the partner of the shared memory communication is decided at run-time. The receiver of data put in a shared memory is the component that reads the data. The sender does not specify its receiver. This enables to change the connection among the components in execution time. Applying it to user interface components and GUI application components, we can realize several advanced HCI techniques as described in the previous chapter. Lastly, it provides high-modularity of GUI applications or GUI platforms. This allows programmers to divide a GUI application into multiple cooperative processes.

By allowing remote processes to access the shared memory through networks, we can obtain other features. We can distribute the cooperative components over distributed environments. This can be a basic mechanism of the multi-user interface and distributed implementation of GUI applications.

---

[1] BTRON is designed on the basis of an integrated resource model, called *Real-Object/Virtual Object Model*. In the model, every resource of BTRON is a Real-Object. For example, BTRON includes Process Real-Objects, Storage Real-Objects, Task Real-Objects, Semaphore Real-Objects, and so on. In BTRON2, a window is also handled as a resource of the Real-Object/Virtual Object Model. Thus, the resource representing a window is named Window *Real-Object*.

Conventionally, many distributed shared memory (DSM) systems are developed [16, 13, 19]. However, there is no DSM system whose application is GUI. The WRO is the first and the only one whose design is tailored for GUI. It provides functionality specially tailored for GUI such as abstract events, lock and local attributes of interaction objects, a context data sharing facility among stored objects, spatial addressing, etc. This GUI-oriented functionality is described in the next section.

Moreover, its implementation is also tailored for GUI. It adopts the implementation mechanisms that shorten and stabilize response time of the system. The implementation issues are shown in Section 4.3.

4.2 Programming Interface of Window Real-Objects

4.2.1 Interaction Objects

A WRO contains a collection of standardized parameterized data segments which represent graphical objects for interaction with users, called *interaction objects*. Each interaction object is stored in a WRO as a *passive* object with a set of parameterized data. That is, only external components such as application core units and user-agents can update its parameters by fixed procedure calls (*window calls*, see Section 4.2.2). The standard data format of the interaction objects is *TAD* [2] [111]. Any application must use this format in representing interaction objects. This standardization is necessary for the communication among unknown GUI application programs with each other. In present, five types of interaction objects are defined for WROs: (1) text objects, (2) graphic objects, (3) control parts [3], (4) pop-up/pull-down menus, and (5) virtual objects [4].

As an example of a graphic object in the TAD format, Figure 4.1 shows a data structure of a rectangle object. It contains parameterized data specifying frame information, drawing patterns, several representation attributes. Figure 4.2 represents another example of interaction objects. It is a data structure of an alternate text switch. It contains parameterized data representing both logical and physical information. The logical information is a switch value and a label name. The physical information is frame information and several display attributes.

[2] See Section 3.1.

[3] The BTRON HMI Specification [111] specifies eleven kinds of control parts: (1) text box, (2) secret text box, (3) number box, (4) serial box, (5) text alternate switch, (6) text momentary switch, (7) pictogram alternate switch, (8) pictogram momentary switch, (9) switch selector, (10) scroll selector, and (11) volume.

[4] A *virtual object* is a rectangular graphic object on a window representing a link to a real-object, which is a node of hypertext-type file system of BTRON. Users can open the real-object into a window by double clicking its pictogram on the upper-left corner.

```
typedef struct{
  UB    subid;    /* = 0 (RECT_ID)   */
  UB    attr;     /* attribute       */
  UH    l_attr;   /* line attributes */
  UH    l_pat;    /* line pattern id */
  UH    f_pat;    /* fill pattern id */
  UH    angle;    /* angle           */
  RECT  frame;    /* frame           */
}TGS_RECT;
```

Figure 4.1: Data structure of a rectangle object of WROs

```
typedef struct {
  W        type;   /* parts type                             */
  RECT     r;      /* frame information of the switch        */
  W        cv;     /* switch value (0: off, othewise: on)    */
  W        len;    /* string length of the following label */
  UB       name[]; /* label of the switch                    */
  PARTDISP atr;    /* attribute for displaying the switch    */
} TEXTSW;
```

Figure 4.2: Data structure of a text switch object of WROs

To facilitate the flexible control of data sharing among processes, several special attributes can be specified for each object. Among them, a *lock attribute* and a *locality attribute* are important.

The *Lock Attribute* indicates that objects can be updated only by the process that has locked them. Other processes can only read the data of the objects. This attribute is controlled by the `loc_obj()` and `unl_obj()` window calls (see Section 4.2.2). Usually, the lock attribute is set by a user-agent when it applies long-term operations to interaction objects. Other user-agents sharing the WRO inform their users of the lock by displaying the objects in *gray*. The users know that they are prohibited from operating on the objects. This technique is generally called *implicit locking* [91], which is effective to prevent conflicts in multi-user interfaces.

The *Locality Attribute* indicates that objects can be read and updated only by a particular process. The locality attribute can be used as a framework to support *user-interface independence* [25] of multi-user interfaces. For example, a semantic data "temperature" is shown to user **A** with an analog volume part and to user **B** with a digital numerical-box part. This is accomplished by storing the **A**'s local volume and the **B**'s local numerical box in the same WRO. Consistency among them can be managed by an application core process, and this involves easy programming with abstract events and window calls (see Section 7.3.4).

4.2.2 Procedural Interface

The WRO provides a simple procedural interface suitable for GUI programming. It mainly consists of seven integrated procedures called *window call*s. We can operate any kinds of interaction objects in WROs by issuing the window calls:

1. `set_obj()` stores interaction objects to a WRO.

2. `del_obj()` deletes interaction objects from a WRO.

3. `rea_obj()` reads interaction object data from a WRO.

4. `upd_obj()` modifies interaction object data. For example, it can move, resize, etc., them.

5. `ope_obj()` applies an operation which does not update parameterized data of interaction objects. For example, click operations to a momentary button switch part, menu selections, etc. This causes only to generate abstract events to notify the operations.

```
W ope_obj(UW wroid, UW search_mode, SKEY *key, UW operation, UW info, UH hint);
```

Figure 4.3: The C notation of the `ope_obj()` window call

```
union WindowObjectSearchKey{
  UW   id; /* when ID_SEARCH      */
  PNT  p;  /* when POINT_SEARCH   */
  RECT r;  /* when REGION_SEARCH  */
};

typedef union WindowObjectSearchKey SKEY;
```

Figure 4.4: The C notation of addressing key data type

6. `loc_obj()` sets a lock or locality attribute to interaction objects.

7. `unl_obj()` unsets the lock or locality attribute of interaction objects.

We have designed window calls so simple that various programming language can utilize them. Currently, our prototype system provides both C and Lisp interface of window calls. Appendix A and B show the C and Lisp notations of window calls.

4.2.3 Addressing

Addressing of a WRO is a mapping from a query key onto interaction object data in the WRO. A WRO supports the management of interaction objects by offering two classes of *spatial* addressing, which uses spatial information as a key to retrieve the objects. *Point Addressing* is a spatial addressing that maps from a point on a window to an interaction object just under the point. *Region Addressing* is an addressing that maps from a rectangular region in a window to interaction objects overlapped with or contained by the region. In the region addressing, programmers can specify to retrieve all objects overlapped with the region or fully contained by the region (*retrieval condition*).

Additionally, the WRO provides other two kinds of addressing. *Selection Addressing* maps to interaction objects in the selected state. Ordinary, a click operation by a user makes the interaction object in the selected state. *ID Addressing* maps directly an object ID to the object of the ID.

When the addressing retrieves multiple interaction objects, programmers can specify three kinds of *sorting order* of the retrieved objects: (1) Back to front order, (2) Front to back order, and (3) No sort.

In window calls, programmers specify addressing using two parameters. `search_mode` specifies the addressing class, retrieval condition, and sorting order. Each mode is represented by a flag bit of the parameter. The parameter `key` is a pointer to its addressing key, which is represented by the union of an integer for an identifier, a point data, and a rectangle data. Figure 4.3 is the C notation of the `ope_obj()` window call.

4.2.4 Examples of Spatial Addressing

Combination of the addressing, retrieval condition and sorting order specification can cover most access patterns to interaction objects.

Example 1: Picking Operation

When a PD click action occurs, a user-agent tries to find the interaction object just under the click point. The user-agent issues a window call with a point addressing in the front-to-back order.

Example 2: Region Selection

When an end-user drags his/her PD for the region selection, the user-agent issues a `rea_obj()` window call with the region addressing in the fully contained mode and in the no sort mode. Then, the user-agent draws selection frames of objects retrieved by the call.

Example 3: Redrawing

When a window region is revealed in front, a user-agent redraws the region. The user-agent issues a `rea_obj()` window call with the region addressing in the overlapped mode and in the back-to-front order, then draws all the retrieved interaction objects in the retrieved order.

Example 4: Object Deletion

When an end-user pushes a delete key, he/she can delete interaction objects in a selected state. To handle this, the user-agent issues a `del_obj()` window call with the selection addressing.

4.2.5 Abstract Events

```
typedef struct abstractevent{
    W  wro_id;      /* which WRO is operated            */
    UH wc_class;    /* which w-call is issued           */
    UH wc_subclass;/* which w-call is issued            */
    UW wo_class;    /* what type of object is operated */
    UW wo_id;       /* which object is operated         */
    ID tsk_id;      /* which task operates              */
    H  utime;       /* when it is operated              */
    UW ltime;       /* when it is operated              */
    UW info1;       /* additional information           */
    UW info2;       /* additional information           */
}A_EVT;
```

Figure 4.5: Data structure of abstract events

[Notation]

```
W get_aev(AEV *aev, TMO tmout);
```

[Parameters]

`aev`	abstract event
`tmout`	time out
return	Normal Completion (≥ 0)
	Error (< 0)

Figure 4.6: Function call for obtaining abstract events

A WRO provides a mechanism to transfer control among processes. Conventional GUI toolkits transfer control by invoking *callback* procedures specified by applications [82, 99]. However, the method reduces flexibility of connections among applications and toolkits. Without special frameworks, it is difficult to invoke callback routines written in Lisp from a toolkit written in C. Moreover, it is difficult to invoke callback routines on a remote machine.

To solve these problems, a WRO provides asynchronous event messages called abstract events. An abstract event is generated when a process updates interaction objects in a WRO, and is sent to processes that have expressed interests in the event. The processes express their interests of abstract events they want to receive by using event mask functions of WROs.

Application core units and user-agents can get abstract events by issuing the `get_aev()` window calls (Figure 4.6). The abstract event conveys information of the operated object identifier (ID), the operation kind, the operator's user ID, the object status before and after the operation, etc (Figure 4.5). When a process receives the abstract event, it executes any necessary calculations and returns feedback to users.

This mechanism increases the component independence among applications and WROs. It can invoke processes written in any languages and running on remote machines. Furthermore, it facilitates the invocation of multiple application core units in parallel, and it can dynamically change connections among them.

4.3 Implementation Issues of Window Real-Objects

This section describes implementation techniques that assure practically sufficient performance of WROs in familiar computing environment.

4.3.1 Performance Consideration

First, we should make clear our standpoint for performance requirements of WROs. Implementation of WROs should be evaluated by the usability of GUI built on them. For the usability, the most important performance factor is short and stable *response time*. In this dissertation, we define the *response time* as the time necessary for users' actions to be reflected on their own display.

Whereas short response time is very important for usability, response time cannot always be short practically. In this case, the problem is which response should be handled immediately. In our implementation strategy, the `rea_obj()` window call should be given a top priority.

This strategy is derived from our experiences as follows. The `rea_obj()` window call is mainly used in implementing rather time critical actions such as "picking" and "drawing" interaction objects. Other window calls are used in implementing feedback for users' *intentional* operations such as sliding a volume, resizing a graphic segment, and deleting a line. While the latter operations are used as commands, the former operations are not commands. In other words, the latter operations cause semantic action of GUI software. However, the former operations do not. We can naturally permit that it takes time to execute our commands. On the other hand, we, usually, expect feedback of non-command operations to be processed immediately.

Through this consideration, we have concluded that *the* `rea_obj()` *window call should be processed prior to other window calls* (**R1**). Thus, in our prototype implementation, a complete copy of data of a WRO is always put on a machine sharing the WRO.

Stable response time is also important. If response time varies according to situations, users cannot recognize immediately whether their operations are accepted by computers or not. Thus, we set three kinds of response stability requirements. First, *the response time should be independent of the number of sharing users of a window* (**R2**). Second, *the response time should be independent of the number of interaction objects in a window* (**R3**). Of course, it is impossible to realize complete independence. The performance degradation according to the increase of the number of objects should be as little as possible. Lastly, *the response time should be independent of the position on a window* (**R4**).

Another important factor is that implementation algorithm causing misoperations of users should not be adopted even if the algorithm is efficient. For example, there are coherence algorithms of distributed shared memory systems that uses cancellation or rollback of operations such as time stamp based algorithms. Thus, we do not adopt such algorithms to WROs.

4.3.2 Module Organization

A WRO consists of three kinds of components: (1) *WRO Data Structure*, (2) *WRO Manager*, and (3) *Distribution Server*. Figure 4.7 illustrates the relationship among them. *WRO Data Structures* contain data body such as interaction objects and index data structures. A BTRON2 machine contains as many WRO Data Structures as WROs. When the window is shared by multiple machines, a complete copy of its corresponding WRO Data Structure is always put on each sharing machine. *WRO Manager* manages accesses to WROs. It processes window calls and generates abstract events. *Distribution*

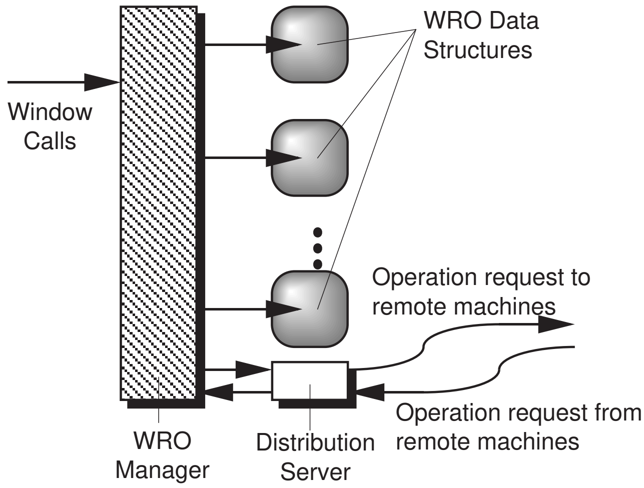


Figure 4.7: Module structure of WROs

Server manages the coherence among copies of WROs. The WRO Manager and the Distribution Server reside one for each BTRON2 machine.

4.3.3 R-tree Index

When interaction objects are stored in a system-provided SDSM, the physical data structure to manage them is fixed, and, therefore, each application cannot tune it. This problem becomes serious especially when the SDSM contains a large number of interaction objects, which is the case in applications such as geographic systems and VLSI CAD systems. To overcome this problem, we apply R-trees [39] to increase the performance of spatial addressing in WROs.

An R-treeis a balanced tree similar to a B-tree with indexed records in its leaf nodes containing pointers to the data object. A non-leaf node contains pointers to its child nodes. An R-tree is structured on the basis of the region each node covers. The non-leaf node represents the smallest rectangle that covers all rectangles of its children (Figure 4.8). There are several efficient data structures for spatial addressing [125]. Among them, the R-tree is efficient when objects are frequently inserted and deleted.

The R-tree method has two relevant advantages. First, it reduces the spatial search time to roughly $O(log_m N)$ [5] [39], while a simple list structure costs O(N). This is a positive factor for the requirements **R1** and **R3**. Second, the difference between the best and worst case of the spatial search becomes small. This is because an R-tree is a

[5] N is the number of interaction objects in a WRO, and m is the minimum number of entries in an R-tree note.

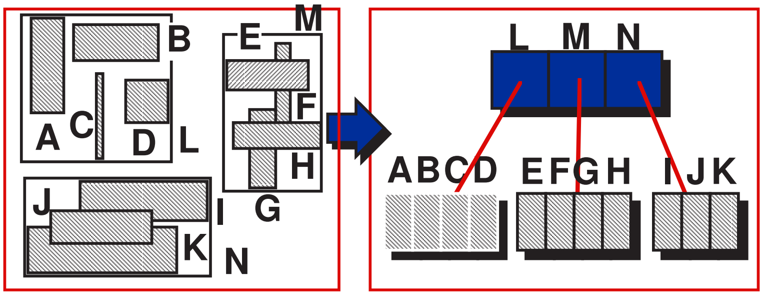


Figure 4.8: R-tree

height-balanced tree. This feature assures stable response time of users' operations. This is effective to accomplish the requirement **R4**.

In our prototype implementation of the current version, the minimum and maximum numbers of entries in a non-root node are 5 and 10. The ones of the root node are 6 and 12. In Chapter 9, we will show the performance evaluation of the total BTRON2 window system, and examine the effectiveness of the R-tree.

4.3.4 Implementation of Window Calls

Window calls can be classified into two classes: the `rea_obj()` window call and other window calls. The former causes no side effect, but the latter causes the side effect. The WRO Manager handles these two classes of window calls differently.

Implementation of the `rea_obj()` Window Call

The WRO Manager can process the `rea_obj()` window calls effectively because it requires no message and is processed on the local machine. The `rea_obj()` window call is processed as follows (See also Figure 4.9):

1. A user-agent on a local machine issues `rea_obj()`.
2. WRO Manager locks the whole WRO Data Structure in a *reader* mode. The reader mode prohibits write accesses to the WRO Data Structure, but allows concurrent read accesses.
3. The WRO Manager retrieves the data that match with a specified addressing key.
4. The WRO Manager unlocks the reader mode.
5. The user-agent receives the value of the read data.

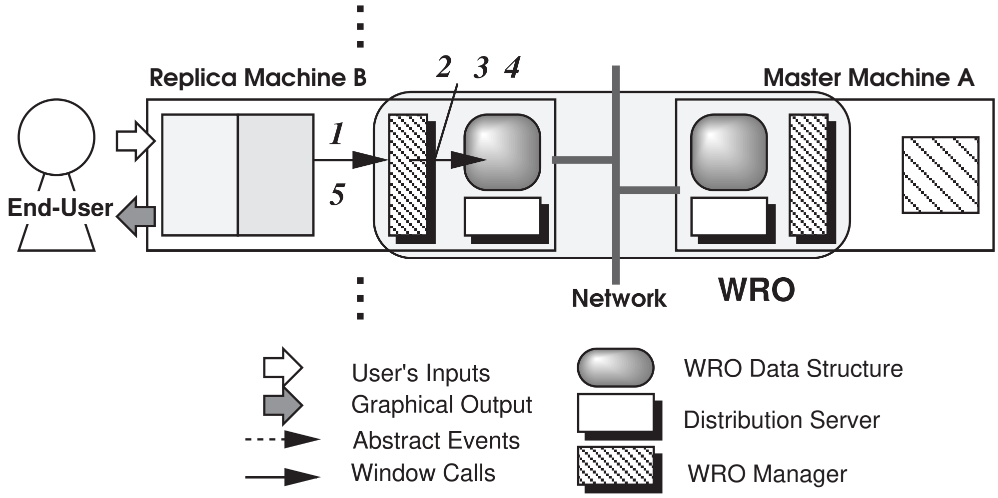


Figure 4.9: Processing algorithm of the `rea_obj()` window call

Implementation of Window Calls with Updates

Other window calls update the data structure in a WRO. These calls may cause inconsistency among their copies without proper coherence algorithms. A WRO adopts a full-replication algorithm, which is mainly implemented by a distribution server in each WRO copy. The algorithm works as follows (see also Figure 4.10).

1. An `upd_obj()` call is issued by a user-agent on machine **C**.

2. The `upd_obj()` call sends a request message to a distribution server on master machine **A**. (The `upd_obj()` call returns asynchronously at this time.)

3. The distribution server on **A** returns the message, in the first, to a distribution server on the original machine **C** (3-1), then to other replica machines (3-2).

4. Every distribution server on **A**, **B**, **C**, and **D** applies the `upd_obj()` call to its local copy of the WRO. Further, the local execution of the call is divided into more detailed steps as follows:

   (a) The WRO Manager locks the whole WRO Data Structure in the *write* mode.

   (b) The WRO Manager updates the data in the WRO Data Structure.

   (c) The WRO Manager unlocks it.

5. An abstract event notifying the update is sent to each user-agent.

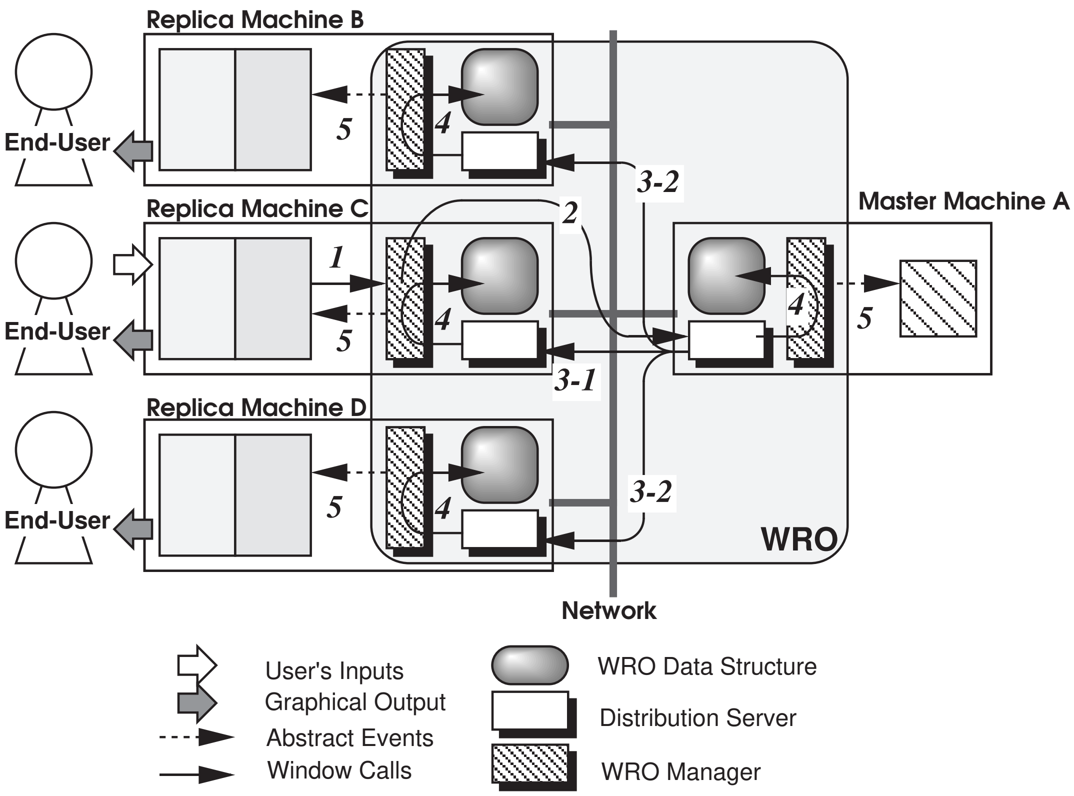


Figure 4.10: Processing algorithm of the `upd_obj()` window call

This algorithm requires only N messages to update copies of the N machines. We can reduce the number of messages if its underlying network system supports multicast or broadcast protocol. However, the current version of our prototype system does not use these functions yet.

The notable technique in this algorithm is that, at the stage 3, the distribution server firstly sends a message to the originally-requesting machine. This technique assures constant response time regardless of the number of machines that have copies (**R2**).

4.4 Related Work

This section shows conventional work related to WROs, and compares functionality or implementation of them.

4.4.1 Distributed Data Sharing Facility

User Interface Software with Data Sharing Facility

Many user interface systems provide data sharing facility or storage-based programming interfaces. Firstly, there are UIMSs whose application interface is organized as a set of shared data. *Higgens* (Human Interface Graphical GENeration System) [46, 47] is a UIMS tailored for the direct manipulation technique. Its main architectural feature is that the application interface of Higgens is a set of shared active objects. *Serpent* (Software Engineering Rapid Prototype ENvironmenT) [9, 10, 8] is also a UIMS that provides an application interface of shared data.

Both of them provide application interfaces of a set of semantic data, and generate user interface modules including the semantic feedback from high level descriptions. The consistency between the physical view of a display and the semantic data is automatically maintained by the UIMSs. For the maintenance, Higgens uses an attribute grammar framework and its incremental evaluation techniques. On the other hand, Serpent uses a rule-based description technique that is implemented using OPS 83.

The main difference between WROs and them is that they provide a database of *semantic data*, while a WRO provides a database of *interaction objects*. Furthermore, the data is not distributed over networks. They do not address applying the framework for new other HCI techniques such as multi-user interface, script processing, application cooperation, and distributed implementation of GUI applications.

Real-time groupware systems provide *distributed* data sharing functionality in distributed computing environment. Such systems are *LIZA* [34], *Rendezvous* [98, 97], *Multi-user Suite* [23, 24, 25], *Teidan* [103], and so on. However, they support to share *semantic* data, whose abstraction level is higher than the stored data in WROs.

There are a few systems that facilitate interaction object sharing. First, *VGTS* (Virtual Graphic Terminal Service) [70, 71] is a pioneer of a GUI platform in distributed environments. It is an interactive graphic system on the V distributed operating system. The main similarity between VGTS and a WRO is its programming interface that is a storage of graphical objects. VGTS takes the server-client architecture, and its server holds the graphical objects. It is innovative that it supports server-client distribution of GUI software and state-based API. However, it does not fully utilize functionality of distributed environments. For example, the graphical objects are shared only between one server and one client. It does not support multiple servers to share the same object set nor multiple clients to share it simultaneously. Thus, it cannot facilitate the distributed implementation of GUI application programs, multi-user interfaces, nor cooperation among GUI applications.

Among UIMSs, *WORKSPACE* [93] uses the shared memory framework for concurrent programming, and it resembles our architecture. Using WORKSPACE, a GUI application can be implemented with multiple light weight processes (LWPs) sharing a single addressing space. However, WORKSPACE does not allow the LWPs to be dispersed over a distributed system because it cannot share the data space across multiple machines.

3D graphics or 3D window systems like *PHIGS* and *PEX* (PHIGS Extension of the X-window) [104, 140, 27] offer similar functionality to our WRO. They provide not only drawing primitives of 3D graphics, but also 3D graphical objects databases and rendering engines. If this concept is applied to the 2D graphics of usual GUI, the architecture and its behavior are very similar to the ones of the BTRON2 window system. In this case, the database corresponds with WROs, and the rendering engine corresponds with a user-agent.

Distributed Shared Memories

Among the many DSM systems, some structured DSMs are similar to our WROs. First, *tuple space* [16, 3, 33] is a pioneer of distributed shared memories and used for applications of parallel computation. The semantics of the tuple space is a shared memory whose data element is a *tuple*, which is a kind of variable length record. The tuples are addressed with an associative (content) addressing. Their access methods are based on pattern matching of each field of the tuples. Access methods are restricted to only `in()`, `out()`, `rd()`, and `eval()`. The tuple space is so popular that it is used for implementation of GUI platforms as referred below.

Agora [13, 30] is a DSM system in heterogeneous distributed environments for parallel application software consisting of modules coded with different programming languages. Agora provides a powerful addressing system like our WRO, however, its addressing is not tailored for GUI.

Problem-Oriented Shared Memory [18, 19] can specialize the address space and data space of its shared data for each application. For example, in case of name server application, the address space is organized in a tree structure reflecting its naming structure. [18] and [19] provide several specialization examples. Nevertheless, they include no user interface software example.

GUI Systems with DSMs

PIX (Parallel Interactive eXecutive) [73] is a complete rewrite of NeWS and is the first [6] window systems adopted explicitly distributed shared memory technology. It implements dictionaries of the PostScript interpreter in the NeWS server using tuple spaces of Linda [16, 3, 33]. However, the usage of the tuple spaces is very limited because it is used only to provide secure access primitives to the dictionaries. Another difference between PIX and our BTRON2 window system is that PIX uses a general purpose distributed shared memory. On the contrary, we use a specially tailored memory for GUI.

T. Masui has advocated to build multi-thread user interface software with general purpose DSM systems to use results of studies of parallel and distributed processing [80]. He has build a multi-thread GUI toolkits, in which threads communicate and synchronize with each other through a tuple space of Linda [79]. Also, he has advanced this idea, and built a real-time groupware using the tuple space [81].

Their objective to adopt DSMs resembles ours: increasing module independence, multi language implementation, flexible control-flow, cooperation with other systems, and so on [81]. In their system, the tuple space is an interface bus among application modules and user interface modules. However, their approach is different from ours. First, they store the tuple space semantic data or physical input data. Their papers include no example that the tuple space contains interaction object data. Second, they use general purpose DSMs, but we use special DSMs tailored for GUI. If the functionality of WROs is implemented using tuple space, there might be performance problems and space utilization problems. For example, can it process spatial addressing efficiently? Can it efficiently store large data structures like drawing pattern data and mask pattern data?

[6] As far as the author knows

4.4.2 Spatial Addressing

Although GUI platforms mainly handle graphical objects, there are few systems which utilize the result of computer geometry such as data structure like R-tree or other data structures suitable for graphical data storage [125].

The only relatd system we have found is the database in the *PROBE* project [95]. It has been building an object-oriented database that strongly supports spatial data. Its applications are VLSI CAD, geographical applications, and so on. It provides query primitives suitable for spatial data such as spatial join. Also, it provides an index data structure which supports spatial queries to be processed efficiently. It uses a 2D gridding index and implements spatial queries using Z-value of the grid data structure.

4.4.3 Distributed Data Sharing Algorithms

Next, we compare distributed data sharing algorithms. Stumm and Zhou have classified algorithms implementing distributed shared memory into three classes: *central-server algorithm*, *migration algorithm*, and *full-replication algorithm* [136]. The distributed data sharing algorithm of WROs is a variation of the full-replication algorithm. Among distributed shared memory systems, the Agora implementation upon loosely-coupled environments adopts the full-replication algorithm [13].

In groupware toolkits, *Teidan* [103] adopts a similar full-replication algorithm to the WRO. Update requests to shared semantic objects are forwarded to a Teidan server, which broadcast the request to all the copied objects (Figure 4.11). A difference between the WRO and Teidan is that the Teidan server aborts operation requests with old timestamps. Our algorithm does not abort operation requests. Distribution server of the master WRO only serializes operation requests in their arrival order.

This difference affects the view of critical conflicts of users' parallel operations. Using our implementation, a user might see that another user's operation interrupts his/her operation. Using Teidan, users' operations are not interrupted by another user's operations, but might be aborted.

Another difference is its implementation. Teidan implements a semantic object with an *object*. It requests a Teidan server update operations and also handles requests broadcasted by the Teidan server. Both processings cannot run concurrently. When Teidan servers and clients communicate synchronously using the RPC primitives, etc., deadlock occurs soon [103]. Thus, they use only asynchronous message passing.

In our implementation, update operation requests are sent by processes issuing window calls, and broadcasted requests are handled by the distribution server. Each

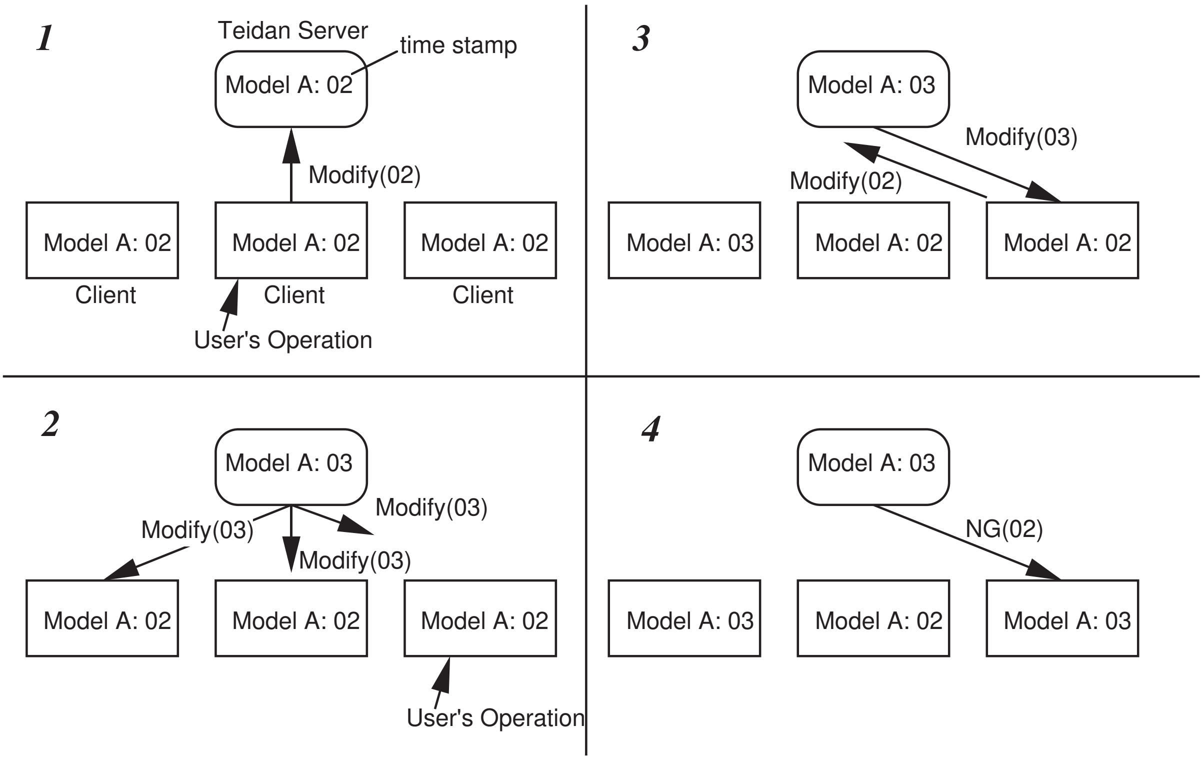


Figure 4.11: Distributed data sharing algorithm of Teidan

processing is handled by different process concurrently. Thus, the deadlock problem like Teidan does not occur in the WRO. In our implementation, some window calls which return information are implemented with synchronous communication, and others are implemented with asynchronous communication primitives to shorten response-time.

A real-time groupware "*GROVE*" [28] uses novel coherence algorithm, called *distributed operational transformation* algorithm. It allows every operation to be handled immediately on the local machine and then forwarded the operation to other remote machines. The remote machines may transform the operation if concurrent operations are handled in an invalid order. This algorithm is very attractive because GUI software can handle update operations immediately on the local machine without obtaining locks. This characteristic is effective for increasing responsiveness of the GUI software. However, this algorithm requires $N^2_{operation}$ transformation rules [7]. GROVE requires only simple transformation rules because it handles only text data and simple operations of the data. On the other hand, WROs manages various interaction objects and operations. Thus, building its transformation rules is unrealistically hard task.

4.5 Summary

This chapter has presented both programming interface and implementation techniques of Window Real-Objects (WROs). The programming interface is specially tailored for storing and retrieving interaction objects. The specializations include

1. lock attribute and locality attribute,
2. spatial addressing, and
3. abstract events.

Additionally, for the ease of programming, we have made its data element, window calls, and addressing all *orthogonal*. This means that programmers can apply *any* window calls to *any* interaction objects with *any* addressing.

The WRO is implemented using R-tree and a variation of full-replication algorithm. These techniques are effective for both shortening and stabilizing response time of users' operations.

[7] $N_{operation}$ is the number of operations to the shared data.

Chapter 5

User-Agents

5.1 Introduction

This chapter presents the detailed description of a user-agent. Here, we focus our discussion on its implementation issues, especially its multitasking issues. The user-agent is implemented with multiple concurrent tasks with priority-based scheduling. Conventionally, user interface researchers have pointed out many problems caused by the single task implementation of GUI. In this chapter, we point out another problem occurring in multi-user interfaces. We show that the multitask implementation of the BTRON2 window system can solve these problems.

5.2 Functionality of User-Agent

The main functionality of a user-agent is to interface among a user and Window Real-Objects (WROs). In BTRON2, the user-agent is an agent of a user that acts as the user requests in a computer. It handles a sequence of inputs from the user, and applies the input onto adequate WROs. For example, when a user moves a knob of a volume part on a window, the user-agent receives a sequence of input events and updates the value of the corresponding volume part data stored in the WRO by issuing window calls.

Conversely, it visualizes the data in the WRO on a screen. It can immediately know updates occurred in WROs by receiving abstract events from them.

Specification of the human-computer dialogue provided by the user-agent is based on the BTRON HMI Specification [111], and contains several extensions on multi-user interface, gestures, and global input modes.

Multi-user Interface Support: A user-agent provides multi-user interface functionality. The current version of our user-agent supports the following three kinds of multi-user interface specification: (1) implicit locking by *graying* to avoid collisions, (2) remote pointers, and (3) complete/continuous coupling. These multi-user interface issues are precisely described in Chapter 7.

Gesture Support: A user-agent contains an engine for gesture recognition. In our current version, it can recognize some of the gestures with one stroke: *circle*, *check*, *hat*, *greater than*, *less than*, etc.

Global input mode: A user-agent manages input mode of all windows on a screen. The input mode is called *global input mode*, and applied to all window commonly. They include text input mode, rectangle input mode, operation mode, gesture, and son forth. For example, while the global input mode is the text input mode, a user can input text on any windows on a screen. Of course, when the target window prohibits text inputting, text input operations to the window fails. The user can change the global input mode by selecting a mode button on the system input pallet (Figure 5.1). In conventional GUI environments, each application or each window defines the input mode. Thus, window switching causes the input mode switching.

We believe the global input mode is more natural than the conventional one. This is because, on the real desktop, the change of paper does not cause the change of its writing mode. We can change our writing mode only when we change our stationery in our hand. When we take a black pencil, we are in a writing mode. When we change the pen with an eraser, we enter in a deletion mode. However, we can never change our input mode by changing paper.

Furthermore, we are planning more advanced HCI technique. In the technique, a user use multiple pens in operating a computer. For example, they include a pen for data input, a pen for data editing, a pen to issue commands to computers and so on. The user can change the input mode of the computer by changing his/her pen. Also, in this case, the input mode is global. Our research group has named this scheme as “stationery-ware” scheme [107].

5.3 Module Organization of User-Agents

5.3.1 The Need for Multitask Implementation

As discussed in the previous chapter, short response time is one of the most important factors for the usability of user interface software systems. However, every feedback cannot be always processed in a short period. There are many kinds of long-term feedback

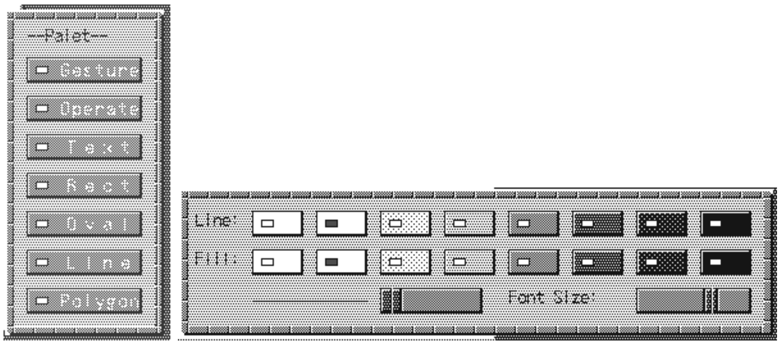


Figure 5.1: System input pallet

in interactive application programs. For example, redrawing windows, printing, and spell checking are the representative ones. However, users want the programs to accept their inputs immediately even during the long-term feedback. This problem is generally known as the *command cancellation problem*, *preemption* in user interface software, etc [26]. Conventional systems deal with this problem by ad hoc schemes such as polling periodically user's inputs during the feedback processing.

Generally, this is a typical problem in real-time systems. That is, important feedback tasks are served prior to unimportant feedback tasks. For example, when multiple requests of feedback processing occur simultaneously, window systems should serve for the request that a user feels important, first. To obtain the flexible feedback ordering, it is very popular method to divide a user-agent into multiple concurrent tasks and to apply them a real-time scheduling algorithm. Our main contribution here is to show, experimentally, the task separation strategy and the priority assignment strategy effective for increasing responsiveness.

5.3.2 Multitasking Strategy

Our multitasking strategy is to divide a user-agent into tasks by *operation* [66]. A user-agent is broken down into many concurrent tasks such as Redrawing Tasks (RTs), a Physical-Events Handling Task (PHT), and Abstract Event Handling Task (AHT) (See also Section 5.3.3 and Figure 5.2).

This multitasking strategy allows multiple feedbacks to be processed concurrently. For example, even while RT is redrawing a large number of graphical objects, PHT can process user's PD operations concurrently. Users need not wait until the redraw terminates.

On the other hand, there is another strategy. It is to divide by *objects*. This strategy

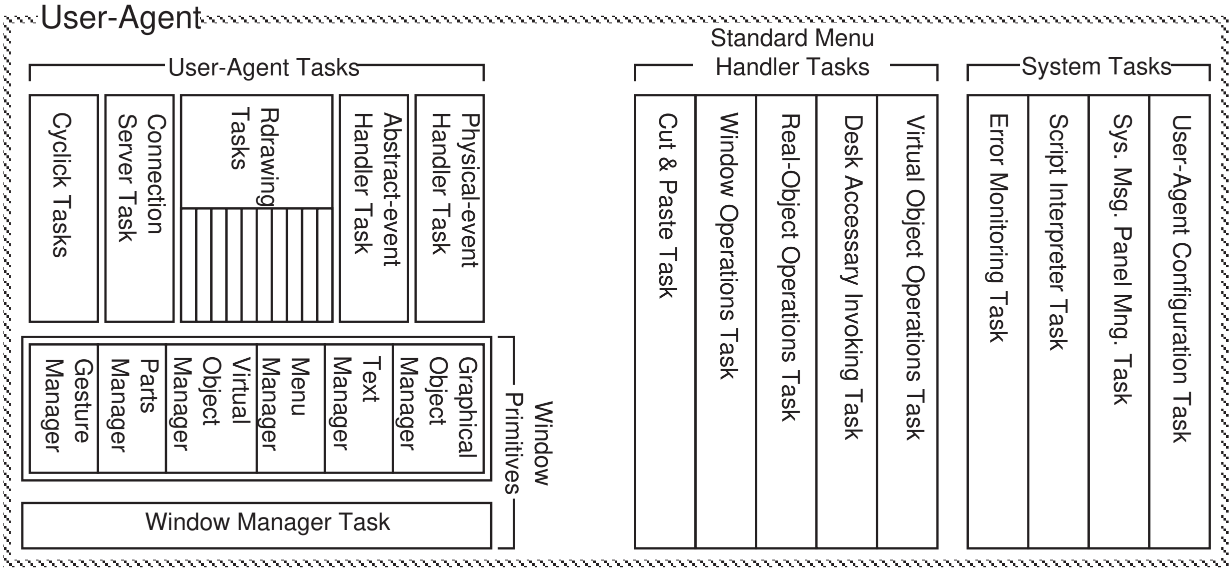


Figure 5.2: Module structure of user-agent

is popular in the object-oriented GUI. It will divide a user-agent into tasks each of which implements an interaction object. For example, a user-agent consists of tasks implementing volume parts, switch parts, virtual objects, and so on. Each task provides all operations for the object such as several kinds of feedback to PD-operations and redrawing. This strategy enables only to control which object is processed first. However, what we want to control is not which object is processed first. We want to control just which operation is processed first. Consequently, the *division-by-object* strategy is not effective for enhancing responsiveness.

5.3.3 Module Structure of User-Agent

A user-agent consists from three kinds of tasks, *user-agent tasks*, *standard menu handler tasks*, and *system tasks*. Figure 5.2 illustrates the relationship among these tasks.

User-Agent Tasks

User-agent tasks provide main functionality of a user-agent, which handles user's inputs, abstract events from WROs, and so on.

- *Window Manager Task (WMT)*: WMT provides abstract devices of *windows* which accept drawing operations from other user-agent tasks. Conversely, the windows generate physical events and pass them into Physical-event handling Task (PHT, see below).

- *Physical-event Handler Task (PHT)*: PHT accepts user's inputs in the form of physical events such as clicks, drags, and key-inputs from the WMT. It issues window calls according to these user's inputs.

- *Abstract-event Handler Task (AHT)*: AHT is a collection of abstract event handlers. It receives abstract events from WROs, and reflects the change of the WROs on a screen by issuing drawing primitives provided by the WMT.

- *Redrawing Tasks (RTs)*: RT processes long-term drawing. Usually, other user-agent tasks invoke the RTs. Short-time drawing such as drawing only one object is processed directly by AHTs. When a WRO is created, an RT is also created together. Thus, a user-agent contains as many RTs as WROs.

- *Cyclic Tasks (CTs)*: CTs wake up periodically, and process several routine works such as flickering feedback of selection frames, blinking feedback of text cursors, and remote pointer drawing. Users can control the frequency of the flickering frames by simply controlling the period to wake up the CTs.

- *Connection Server Task (CST)*: New interactive application programs firstly communicate with CST, which handles initialization processing of a user-agent and a WRO.

Standard Menu Handling Tasks

A Standard Menu Handler Task is a kind of AHT. It receives abstract events that notify the selection of standard menu-items such as desk accessory invoking, virtual object operations, real-object operations, window operations, and cut and paste operations. The user-agent contains multiple Standard Menu Handling Tasks each of which handles the selections of each menu item.

System Tasks

System tasks are application core units provided in standard by the BTRON2 window system. They include the following tasks.

- *User-agent Configuration Tasks (UCT)*: UCT implements system input pallets to control the Global Input Mode.

- *System message panel Managing Task (SMT)*: SMT manages outputs into the System Message Panel [111] displayed below of the BTRON screen. Additionally, it periodically updates the timer displayed at the right side of the System Message Panel.
- *Script Interpreter Task (SIT)*: SIT is an interpreter task that executes script programs. Currently, SIT is a scheme interpreter. It will be replaced by TACL [1] [110, 57] Manager Task in the near future.
- *Error Monitor Task (EMT)*: EMT manages error outputs. It displays system errors on a system message panel or a debug console.

5.4 Behavior of User-Agents

This section gives typical examples of internal actions of user-agents.

5.4.1 Handling an Instant User's Action

First, we describe internal actions of user-agents handling an instant user's action such as gesture, click, and button press operations. Figure 5.3 shows an example of its gesture handling. This is the most simple action sequence of the user-agent. In this case, the user-agent traces the PD actions and recognize gestures from the traced data. Then, the user-agent issues an `ope_obj()` window call to apply the gesture operation. In this case, addressing is the point addressing whose key is just the hot point of the gesture.

5.4.2 Handling a Long-Term User's Action

The second case is to handle a long-term user's action such as resizing/moving interaction objects by dragging and menu selection. In this case, interaction objects are continuously operated in a few seconds. Thus, to avoid conflicts with other users' operations, the user-agent locks the object during the operation. Then, the user-agent reads the object data and returns temporary feedback on a screen. As soon as the operation terminates, the user-agent updates the value of the objects then unlocks the object. This sequence is just like a transaction operation. Figure 5.4 shows the interaction between the user-agent and a WRO.

[1] TACL is an abbreviation of TRON Application Control-flow Language.

```
Event    ev;      /* window event                     */
ID       win_id;  /* identifier of hot window         */
ID       wro_id;  /* identifier of WRO of the hot window */
            :
while((type = wget_evt(&ev)) == PD_PRESS) {
           trace_pen_action...
}
gesture_id = recogniza_gesture(traced_pen_action_data);
swirch(gesture_id) {
case GESTURE_1:
       :
case GESTURE_M:
        ope_obj(wro_id, POINT, (S_KEY*) &hotpoint, ....);
             :
        break;
             :
case GESTURE_N:
          :
```

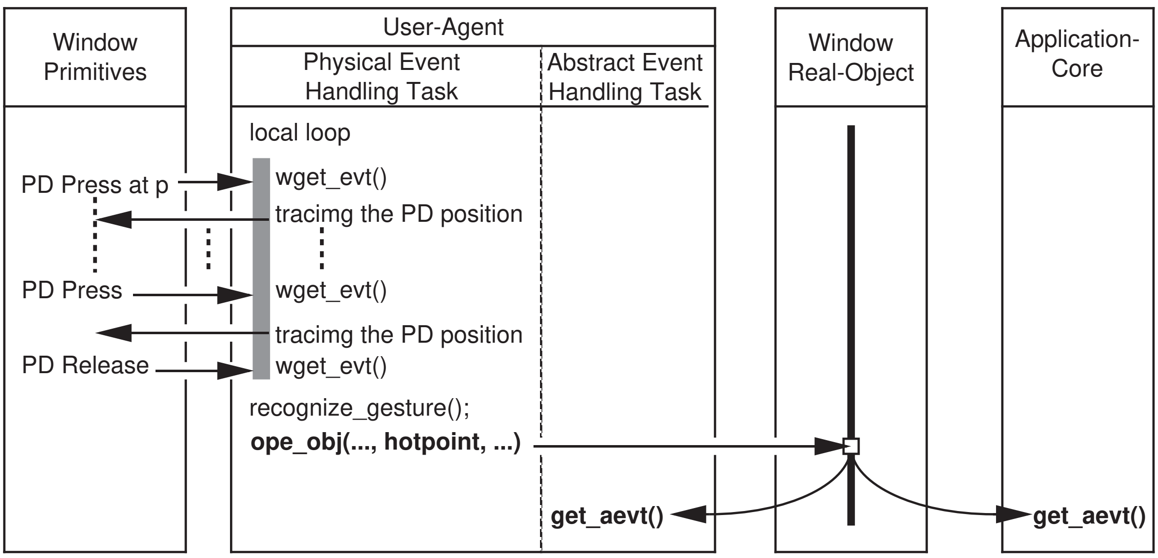


Figure 5.3: An example of action sequence of a user-agent in handling an instant user's action

```
Event    ev;          /* physical event                          */
ID       obj_id;      /* identifier of an interaction object,    */
ID       win_id;      /* identifier of hot window                */
ID       wro_id;      /* identifier of WRO of the hot window     */

switch(type = wget_evt(&ev)) {
                      :
case PD_PRESS:
                      :
    loc_obj(wro_id, ID, (S_KEY*) &obj_id, ....);                 (1)
    if((obj_id = rea_obj(wro_id, POINT, (S_KEY) ev.p,.., &object,..))>= 0){
         while((type = wget_evt(&ev)) != PD_RELEASE) {
              draw_drag_frame(win_id,... )
             if(coupling_mode == INCREMENTAL){                   (2)
                    upd_obj(wro_id, ID, (S_KEY*) &obj_id, ....);
              }
         }
        upd_obj(wro_id, ID, (S_KEY*) &obj_id, ....);             (3)
        unl_obj(wro_id, ID, (S_KEY*) &obj_id, ....);             (4)
     }else{ /* else (There is no object under the specified point.) */
         if(...) clearing selection frame...
      }
     break;
}
```

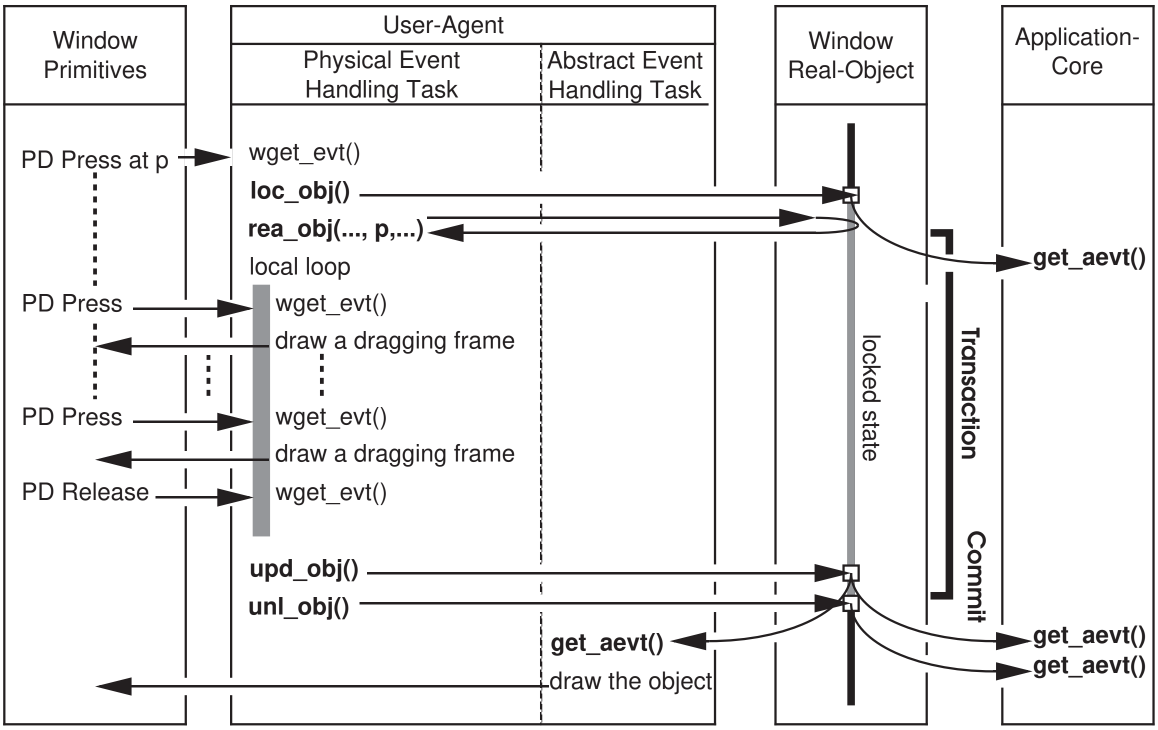


Figure 5.4: An example of action sequence of a user-agent in handling a long-term user's action

5.4.3 Object Registration

Object registration is a kind of long-term operation that register a new graphical object in a window. On the contrary with the previous example, the object registration operation requires no object locking. After terminating the user's drag operation, the user-agent issues a `set_obj()` window call to register a graphical object into a WRO. Figure 5.5 traces the action sequence of the object registration.

5.4.4 Redrawing

When redrawing a region on a window, a user-agent first retrieves the whole interaction object data overlapped with the region with the region addressing. Then, it draws obtained interaction objects one by one. Figure 5.6 is a sample code for implementing the redraw operation.

5.5 Scheduling Strategy

To make our multitasking approach more efficient, we have adopted priority-based scheduling among tasks. Each task is given a priority according to its "importance." The task can be suspended when another task with a higher priority starts executing. The suspended task resumes after the higher-priority task terminates.

The *user-agent scheduler* handles this scheduling. It schedules the tasks, in the first stage, on the basis of their priorities. Only the task with the highest priority can run. If there are multiple tasks of the same priority, as its second stage, it schedules them by the *round-robin* scheme in 50 msec.

In the BTRON2 window system, the task with the highest priority is the Distribution Server task of WROs. This means that the consistency among WRO replicas is considered as most important. PHT and WMT of the user-agent are the next priority. Other tasks are given the same lowest priority.

By this examination, we can confirm that the multitask implementation with priority-based scheduling can control feedback order. Our feedback order is as follows: (1) coherent algorithm among WRO replicas, (2) physical event handling, (3) redrawing, and 4) semantic actions.

This approach requires frequent inter-task communications and task switches. Our approach cannot be effective without operating systems with fast inter-task communication/synchronization mechanisms and fast task switches. Operating systems of the TRON architecture, especially ITRON, can meet the requirement.

```
Event    ev;       /* window event                      */
ID       win_id;   /* identifier of hot window          */
ID       wro_id;   /* identifier of WRO of the hot window */

switch(type = wget_evt(&ev)) {
      :
case PD_PRESS:
            :
        while((type = wget_evt(&ev)) != PD_RELEASE) {
                draw_drag_frame(win_id, )
        }
        set_obj(wro_id, ID, (S_KEY) obj_id, ....);
        break;
```

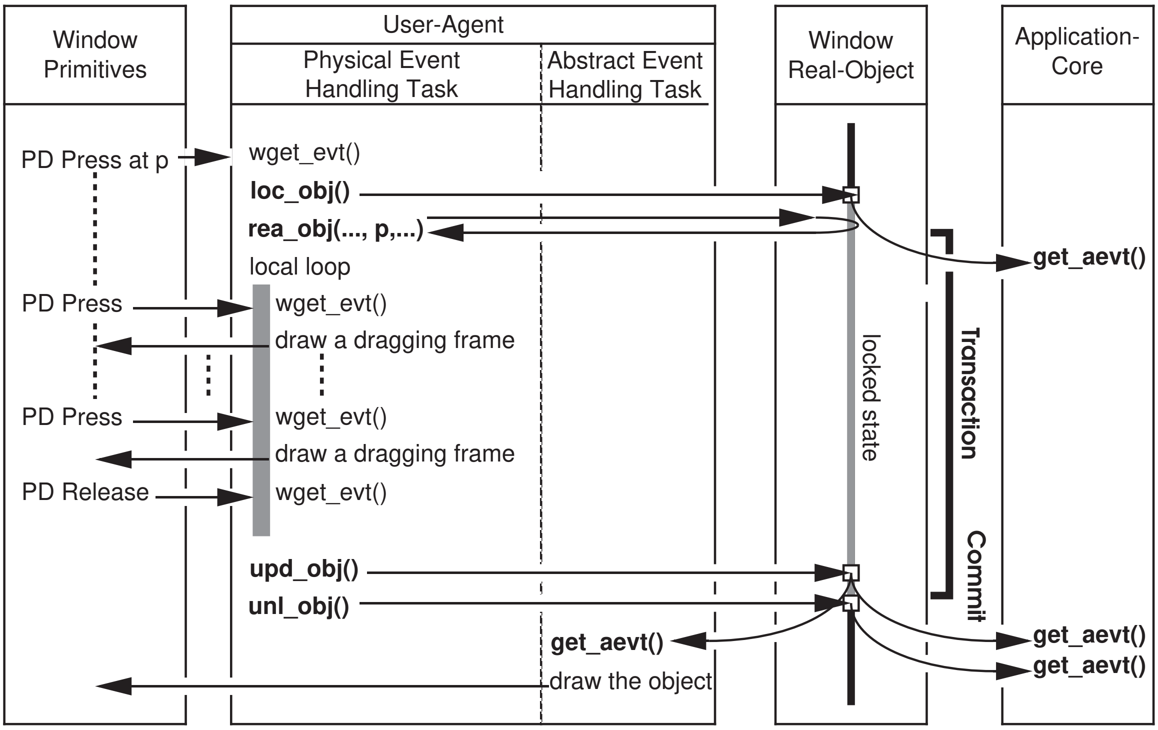


Figure 5.5: An example of action sequence of a user-agent in handling object registrations

```
redraw_region(RECT r)
{
                 :
    obj_num = rea_obj(wro_id, REGION_SEARCH|BOTTOM2TOP, (SKEY *)(&r),obj,...);
                 :
     for(i = 0 ; i < obj_num ; i++) {
                 :
          switch(objType(obj->type)) {
          case GSOBJ:
            drawGraphicObject(wro_id, my_win, (GOBJ_GENERAL *) obj);
              break;
          case PARTS:
              drawParts(wro_id, my_win, (PARTS_GENERAL *) obj);
              break;
          case VOBJ:
            drawVirtualObject(wro_id, my_win, (VOBJ_GENERAL *) obj);
              break;
          }
          obj = getNextTadSegment(obj);
     }
}
```

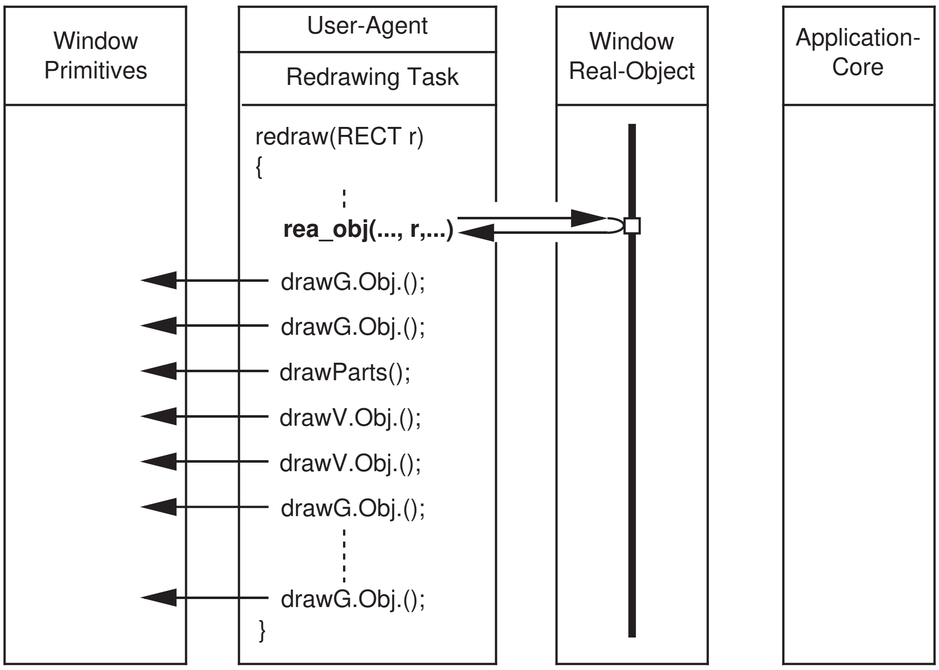


Figure 5.6: An example of action sequence of a user-agent in redrawing

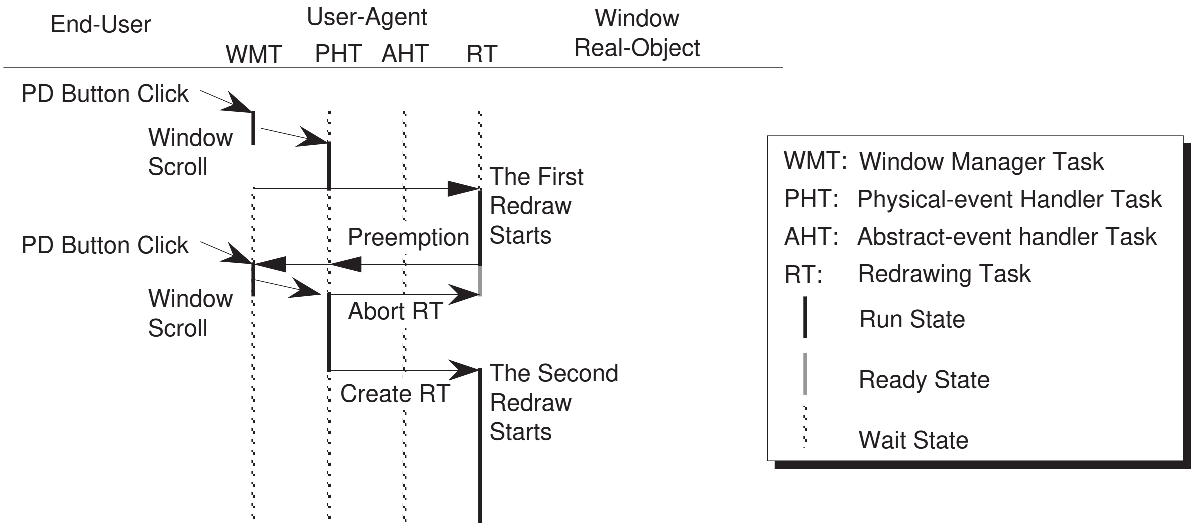


Figure 5.7: Handling the user interface abortion problem

5.6 Solution to Single Task Implementation Problems

This section shows three problems occurring when GUI software is implemented with a single task. The first two problems are popular problems that many researchers have described [26, 69, 86], and the last problem is newly introduced by us.

5.6.1 The User Interface Abortion Problem

[Problem]

Soon after a user makes a jump scroll of a window, the user immediately makes a jump scroll again. If the redraw processing caused by the first scroll has not finished yet, it is efficient to abort the first redrawing and, again, to redraw them after processing the second scroll operation. Nevertheless, most conventional interactive application programs start the second redraw after completing the first redrawing event if the first redraw takes very long time.

[Solution by Multitasking]

In the BTRON2 window system, this can be implemented by giving high-priority to a Physical-event Handler Task, and it aborts the Redrawing Task, then it launches another Redrawing Task again after finishing its processing.

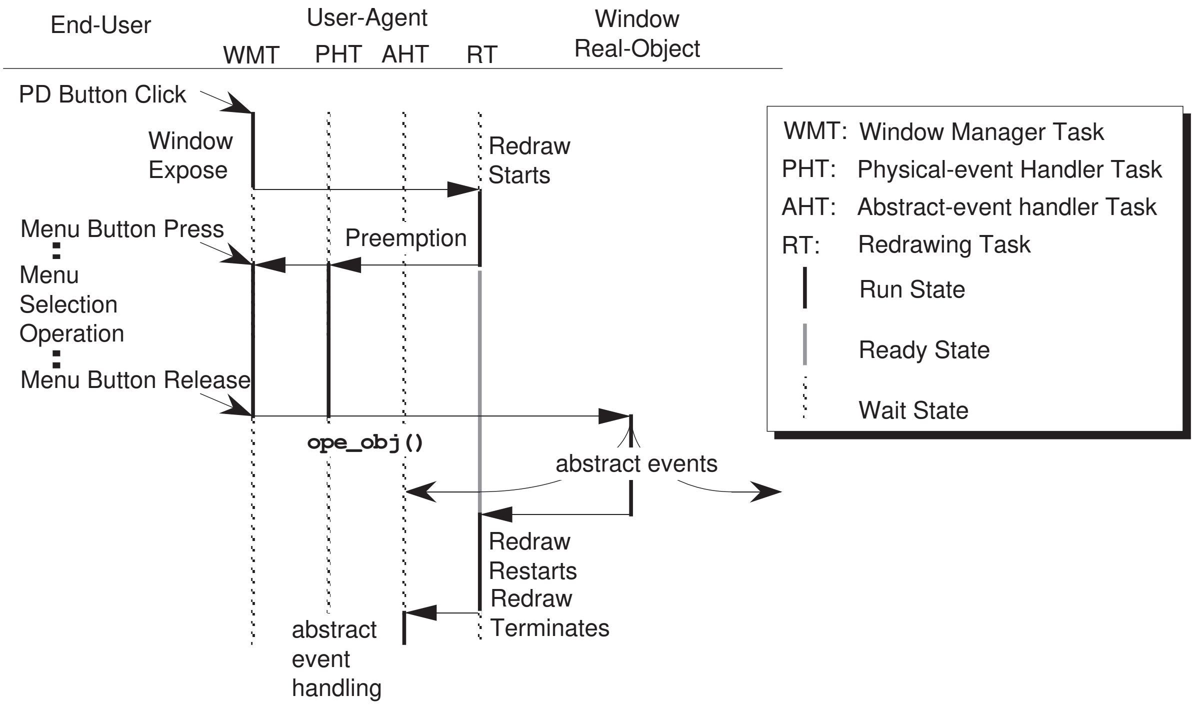


Figure 5.8: Handling a long term drawing problem

5.6.2 The Long Term Drawing Problems

[Problem]

Even while a window is redrawn, a user may want his/her operation to be processed immediately (e.g., menu operation etc.).

[Solution by Multitasking]

This problem is simply solved in the BTRON2 window system. In the user-agent, the task to process the long-term drawing (Redrawing Task) and the task to handle users' operations (Physical-event Handler Task) are different. When the drawing is requested, a Redrawing Task is waken up and starts the redraw processing. Then, the user performs another operation on the window, and the Physical-event Handler Task restarts. Because the Physical-event Handler Task is given a higher priority than the Redrawing Task, the Redrawing Task is suspended and the Physical-event Handler Task is scheduled.

5.6.3 The Shared Window Locking Problem in Multi-user Interface

[Problem]

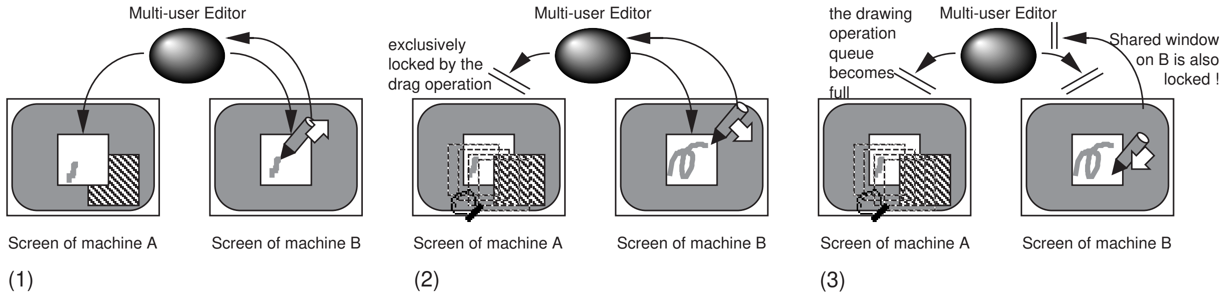


Figure 5.9: The shared window locking problem

For example, consider the case in which a window of a multi-user editor is shared from machine **A** and **B** (Figure 5.9-(1)). When an **A**'s user is dragging to move a window, a **B**'s user starts drawing a segment in the shared window. During the drag operation by the **A**'s user, the window system on **A** serves exclusively to the drag, and cannot reflect the update of the shared window on **A**'s screen (Figure 5.9-(2)). When the drawing operation queue of the multi-user editor on **A** becomes full, the window system stops its processing (Figure 5.9-(3)). This means that operation on a window may lock behavior of other windows of another machine.

[Solution by Multitasking]

Multitask implementation can easily resolve this problem. In our implementation, Window Manager Task handles dragging for a window move, and Abstract-event Handler Task and Redrawing Task handle drawing. Since these tasks can run concurrently, the window drag operation does not lock the window system exclusively. While the user drags, a user-agent can reflect the update of the shared window.

5.7 Related Work

Previously, many researchers [41, 26, 40, 69, 86] have advocated the need for concurrency in user interface software. The concurrency can be classified into two classes: fine grained concurrency and coarse grain concurrency. The fine grained concurrency is a concurrency to handle asynchronous or parallel input of users in non-modal direct manipulation user interfaces. For example, the user's operations to drag a knob of a volume part and to type-in a string in a text box part are essentially concurrent processes. To handle these actions elegantly, user interface software often adopts concurrent programming paradigms.

Conventionally, many GUI platforms have supported the fine grained concurrency. Sassafras UIMS [43] and Switchboard of Adagio [141] are the pioneering systems of the concurrency. As for window systems, SUN NeWS [137] adopts the fine grained concurrency. GMW [52] adopts the concurrent object-oriented mechanism as a more general framework for the concurrency.

The coarse grained concurrency is concurrency among long-term feedback processing and other processings. Even during the long-term feedback, users want computers to accept their inputs. To deal with this requirements, coarser concurrency than the fine grained concurrency is effective. In this technique, user's actions are handled by another concurrent task than the long-term feedback task. One of the coarse grained concurrency is often referred as *command cancellation problem* [69].

Unlike the fine grained concurrency, there are only few contributions of this concurrency. Literatures of the systems with the fine grained concurrency [52, 43, 141] do not address how to solve the coarse grained concurrency problem. The only available examples are performed by sophisticated application programs: for example, background repagenation of Microsoft Word and successive scroll handling of GNU Emacs [132, 133] are among them.

5.8 Future Work

In user-agents, multiple tasks concurrently draw graphics on a window. Our prototype system allows only one task to draw on a window at the same time. This mutual exclusion is implemented using a semaphore of the ITRON kernel. The most frequent case of the concurrent drawing occurs between Cyclic Task (CT) drawing flickering frames and other tasks.

For example, there is a window with several flickering frame of selected objects. Before a Redarwing Task (RT) draws graphical objects on the window, it must wait for the CT to finish its drawing. This pre-processing before drawing degrades the response time for user's operations.

However, if the drawn region has no overlap with the frames, the RT and the CT can draw concurrently. Generally speaking, the required technique is the efficient technique of concurrent accesses to continuous multi-dimensional spaces. We are planning to implement the *spatial lock* technique [64]. Our spatial technique classified drawing operations into two classes: *reversible drawing* [2] and *non-reversible drawing*. Two

[2]Concretely, drawing in the XOR mode is the reversible drawing. Flickering frames and blinking objects are the representative ones.

reversible drawing operations can be handled concurrently even if they are overlapping. In other case, any two drawing operations cannot be processed concurrently if they are overlapping.

5.9 Summary

This chapter has described the multi-tasking implementation of user-agents. Our multi-tasking strategy is to decompose a user-agent by operations. Consequently, the user-agent consists of Window Manager Task, Physical-event Handler Task, Abstract-event Handler Task, Redrawing Tasks, Cyclic Tasks, Connection Server Task, and so on.

Next, in this chapter, we have shown that our multi-tasking approach can solve several important usability problems of single threaded implementation such as the user interface abortion problem, the long term redrawing problem, and the shared window locking problem.

Multi-task implementation tends to degrade total system performance because of task switching costs and scheduling costs. To solve this problem, we have adopted a real-time scheduling mechanism based on priorities of each task.

Chapter 6

Application Core Units

An application core unit handles the semantic computation of a GUI application. This chapter describes application programming issues upon the BTRON2 window system.

6.1 Introduction

Conventionally, most GUI applications are built with a monolithic architecture in which a single program provides all necessary functions for a particular task. For example, a word processor provides text editing function, text formatting function, file load/save function, printing function, spell check function, and so on. Thus, both the size and the complexity of practical GUI software become very large. As Smith and Susser of Apple Computer Inc., have stated, "*the functionality of applications written for personal computers is approaching the limits that current software technology can support*" [131].

The BTRON2 window system supports composing a GUI application with multiple components each of which provides a small piece of functionality. For example, an end-user can compose a word processor with independent components from different vendors, such as a text editing component, a text formatting component, and a spell checking component.

Furthermore, this facility offers other novel features of the GUI application construction as follows:

1. Allowing multi-language implementation:

   Application core units can be programmed with several programming languages. This is because application programming interface of WROs is assumed only popular abstractions that most programming languages provide: function calls and asynchronous messages.

2. Allowing distributed implementation of GUI applications:

Application core units can be dispersed into multiple machines, and executed in parallel and on suitable machines for their execution. For example, in a word processor, a spell checking unit runs on a file server machine, a printing unit runs on a printer server machine, and an editing unit runs on a workstation machine. Of course, all of them can run in parallel on each machine.

Supporting the implementation of these units, it provides higher-level of programming interface with following features:

1. Abstract programming interface as GUI toolkit:

   Abstraction level of the window call/abstract event programming interface is just the level of interaction object component, which is similar to the one of conventional GUI toolkits but includes novel functionality. Therefore, application programmers can skip the programming of redrawing windows, interaction object actions, the cut&paste and drag&drop operations, simple gestures, minimum multi-user interface functionality, and so on.

2. Server-like programming of GUI applications:

   Each application core unit consists of several abstract event handlers. Each abstract event handler does not use annoyed event loop nesting because each abstract event notifies logically meaningful operations of users. Thus, the programming of the abstract event handler is rather the server-like style than the event-loop style.

3. Integrated support of advanced HCI techniques:

   The BTRON2 window system supports multiple advanced HCI techniques such as multi-user interface, gestures, and script processing in "post-it metaphor" in a uniform framework. Application programmers need not make efforts to combine multiple toolkits.

In the rest of this chapter, we describe an overview of the general application programming style upon the BTRON2 window system. Furthermore, we show the novel features of the application programming framework: application decomposition, multi-language implementation and distributed implementation.

6.2 GUI Programming on the BTRON2 Window System

An application program upon the BTRON window system is a collection of abstract event handlers. Application programs can be classified into *master applications* and *servant* *applications*. The former creates and deletes WROs, and the latter only connects WROs created by other master applications.

Generally, there are the following six steps in the master application programming:

1. Initializing
2. Creating a WRO
3. Storing interaction objects to the WROs
4. Handling abstract events from the WRO
5. Deleting the WRO
6. Terminating

Similarly, there are the following five steps in the servant application programming:

1. Initializing
2. Connecting with a WRO
3. Handling abstract events from the WRO
4. Disconnecting from the WRO
5. Terminating

Figure 6.1 and 6.2 show a sample code of the master application programming.

In both cases, the main portion is the step *handling abstract events*. As described in the figures, the step *handling abstract events* is programmed as follows: (1) getting an abstract event, (2) branching according to the type of event, (3) computing application-dependent actions, and (4) reflecting the result of the computation by issuing window calls. In the step, this programming style is rather the server programming of the server-client model than the event loop because it has no event loop nesting.

6.3 Application Decomposition

The BTRON2 window system supports to decompose a GUI application into multiple components with a small piece of functionality. We call the concept of this decomposition *application decomposition*.

As a result of the application decomposition, this architecture represents high modularity, which provides the following three novel features: First, a user can replace even

▷ Step 1 (Initializing)

```
void main(argc, argv)
int   argc;
char  **argv;
{
        :
  ini_sys(mbfid, src);
        :
```

▷ Step 2 (Creating a WRO)

```
        :
  wroid = set_wro(src, 550, 150, &hint);
        :
```

▷ Step 3 (Storing interaction objects to the WRO)

```
        :
  volLw.type = PARTS | VL_PARTS | P_DISP | P_HALIGN;
  SetRECT(volLw.r, 150, 10, 530, 40);
  volLw.clo = 10;
  volLw.chi = 30;
  volLw.lo = 0;
  volLw.hi = 100;
  vol1_ID = set_obj(wroid,TOP2BOTTOM,PARTS,(UB *)&volLw,sizeof(VOLUME),0);
        :
  sw.type = PARTS | AS_PARTS | P_DISP;
  SetRECT(sw.r, 150, 90, 230, 110);
  sw.cv = 0;
  sw.len = 4;
  strcpy(sw.name, "AUTO");
  swAuto_ID = set_obj(wroid, TOP2BOTTOM,PARTS,(UB *)&sw,sizeof(TEXTSW),0);
        :
```

Figure 6.1: Summary of a sample master application programming (Part 1)

▷ **Step 4 (Handling abstract events from the WRO)**

```
          :
  gme_aev(wroid);
          :
  while(FOREVER){
    aev_type = get_aev(&aev, TMO_FEVR);

    switch(aev_type){
    case UPD_OBJ:
      if(aev.wo_id == vol1_ID) {
        sematic_action_vol1(....);    /* Semantic actions */
      }else if(aev.wo_id == vol2_ID){
        sematic_action_vol2(....);    /* Semantic actions */
      }
      break;
    case OPE_OBJ:
      if(aev.wo_id == swAuto_ID) {
        sematic_action_Auto(....);    /* Semantic actions */
      }else if(aev.wo_id == swManual_ID){
        sematic_action_Manual(....);  /* Semantic actions */
      }else if(aev.wo_id == swQuit_ID){
       goto QUITTING;              /* Quitting this application */
      }
      break;
    }
  }
```

▷ **Step 5 (Deleting the WRO)**

```
QUITTING:
  del_wro(wroid);
```

▷ **Step 6 (Terminating)**

```
  dgm_aev(wroid) ;
  qui_sys();
}
```

Figure 6.2: Summary of a sample master application programming (Part 2)

one unit of an application with another unit from another vendor even if it is provided object code form. Second, each unit can be used by another application. Finally, a new unit can be added dynamically. Of course, unit replacement and unit sharing are also executed dynamically when the application is running. For example, an end-user can replace a spell checker in a GUI application with another one. The spell checker can be used from both a word processor and a graphic editor.

For the application decomposition, the key technology is to support these components to cooperate *like a single GUI application program*. If each component behaves like conventional single function applications, the composed application does not provide good usability. For example, consider a small application program with only spell checking function. Making a document, a user must repeat the operations to close and open application programs many times: to close a word processor and to open a spell checker, then to close the spell checker and to open again the word processor. This boring operation sequence annoys most users and degrades the performance of their work. In implementing cooperative application core units, a key technique to make the composite of multiple components behave as if an integrated application is that they can share the same window. In the BTRON2 window system, this technique is easily implemented by sharing the same WRO at the same time.

6.3.1 Application Decomposition Strategy

For the application decomposition, we must separate an application into multiple components that require no communication with each other except through a WRO. This separation assures that communication through a WRO is sufficient for their cooperation. This separation is possible in most GUI applications because each of them consists of several independent modules. For example, in a word processor, a spell checking module handler and a printing handler usually require no communication except the shared document data.

For example, our sample graphic editor is implemented with cooperative application core units as follows. Each program implements semantic actions invoked by menu selection or button pressing. For example, such semantic actions are file accessing, spell checking, printing, font selecting, line width selecting, editing mode selecting, etc. (Figure 6.3). Each action is implemented by an individual application core units. This application decomposition is based on the *division-by-operation* strategy.

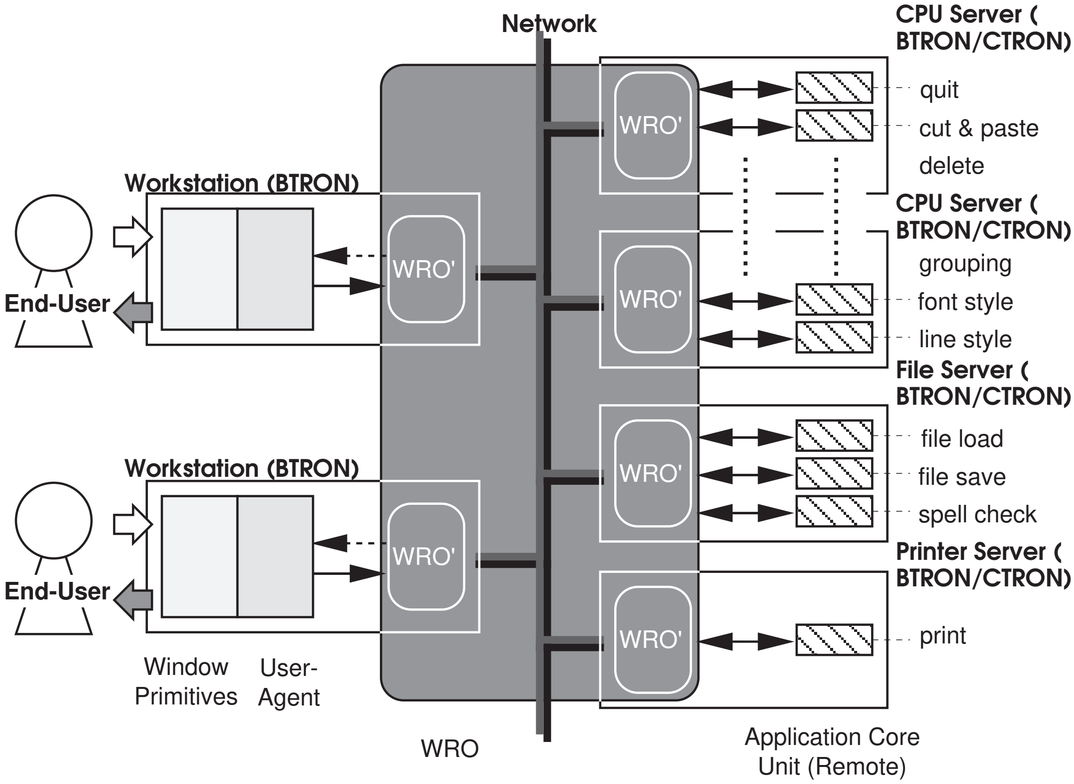


Figure 6.3: The distributed architecture of multi-user graphic editor (specialist parallelism)

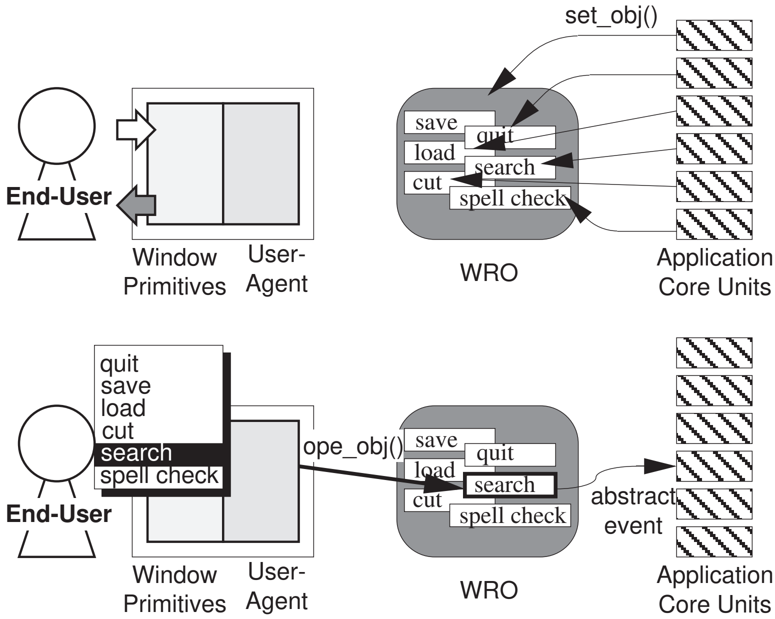


Figure 6.4: A mechanism to control multiple application core units via a window

6.3.2 Control Transfer among Decomposed Processes

Application core units can obtain their control by receiving abstract events from WROs. To receive necessary abstract events, each application core unit should customize its GUI software environment. There are two kinds of customization as follows.

One method is to be invoked by abstract events directly from the WRO shared by all collaborating units. To be invoked by this method, it stores, in advance at its initialization time, its own interaction object such as a menu item object in a WRO.

In the previously referred example, each application core unit stores its own menu item object into a WRO. A file save unit stores a menu item labeled as “Save,” a string search unit stores “Find,” etc. Each unit is waiting for abstract events notifying select operations of its menu item. When users select the menu item, the WRO generates an abstract event notifying the selection, and sends it to the application core unit (Figure 6.4).

Another method is to receive abstract events from its own command window. For the purpose, it must create another WRO, which contains several control parts specifying parameters required for the application core unit. They are, for example, text/numerical box parts, a button switch for the invocation of the unit, and so forth.

For example, a string search unit creates a WRO for a command window, and stores a text box for the search string input and a switch to start the search. This WRO is displayed

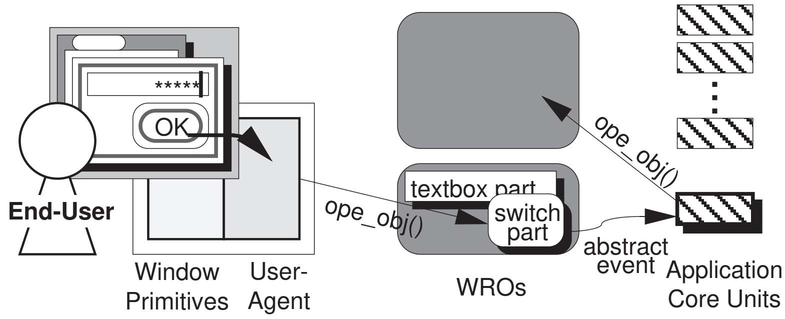


Figure 6.5: A mechanism to control multiple application core units from another window

as a small command window, and users can start the string search by clicking the button switch on the window (Figure 6.5). As well as a user-agent, an application core unit can invoke another unit by issuing window calls to operate the menu item that the latter unit has set in advance.

The abstract event mechanism can work as a control transfer mechanism not only between a user and an application core unit, but also among multiple application core units. When one application core unit wants to invoke another application core unit, it issues the `ope_obj()` window call as user-agents do.

6.4 Multi-language Implementation of GUI Applications

Upon the BTRON2 window system, programmers can not only decompose a GUI application into multiple small components but also write each component with a suitable programming language for its function and characteristics. This is because the window call/abstract event programming interface requires only most basic abstraction mechanism: function calls and asynchronous message passing. Currently, our prototype system supports C, scheme, and lisp, and furthermore, these units written in difference languages can cooperate with each other.

For example, in a graphic editor application, a file accessing component is programmed in C and an inferring component is programmed in lisp or prolog. A component required high execution performance is made by compiled codes, and a component required user's modification or extension is made by interpreting codes. Figure 6.6 and 6.7 are sample menu handler program codes of the graphic editor that changes the line width of selected objects into one point.

```
#include "BGSHELL.h"
void main(int argc, char **argv)
{
  ABSTRACT_EVENT ev;           /* Abstract event */
  unsigned int   l_atr = 0x01; /* Line width     */
  ERROR          err;

  ini_sys(MBF_ID, MACHINE_ID); /* Initialization             */
  gme_aev(ACCESSWROID);         /* Abstract event requesting */

  while((err = get_aev(&ev, TMO_FEVR)) >=0)/* Obtaining abstract events*/
    if((ev.wo_class    == MENU_OBJ)&&  /* Menu operation           */
       (ev.wo_id      == MENUOBJID)&& /* Object id == Menu        */
       (ev.wc_class    == OPE_WO) &&   /* operate                  */
       (ev.wc_subclass == CLICK)&&     /* operation == click       */
       (ev.info1      == 1PT_ITEM))   /* selected menu item == 1pt */

       /* updating the line width of interaction objects */
       /* in selection state                             */
      upd_obj(aev.wro_id, SELECTION_SEARCH, NULLKEY, L_ATTR, &l_atr);

  qui_sys();
  exit(0);
}
```

Figure 6.6: Sample C code to change the line width into one point

```
(define (make-latr-1pt)
  (do (init_sys MBF_ID MACHINE_ID)
      (gme_aev ACCESSWROID)
      (repeat
         (let* ((ev          (Get_AEVT TMO_FEVR))
                (woclass     (get_wo_class    ev) MENU_OBJ)
                (woid        (get_wo_id       ev) MENUOBJID)
                (wcclass     (get_wc_class    ev) OPE_WO)
                (wcsubclass  (get_wc_subclass ev) CLICK)
                (info        (get_info1       ev) 1PT_ITEM))
            (cond ((and (= woclass    MENU_OBJ)
                        (= woid       MENUOBJID)
                        (= wcclass    OPE_WO)
                        (= wcsubclass CLICK_GESTURE)
                        (= info       LINE_1PT_ITEM))
                   (upd_obj (get_wro_id ev)
                            '(SELECTION_SEARCH NULLKEY)
                            'L_ATTR
                            info)))))))
```

Figure 6.7: Sample Scheme code to change the line width into one point

6.5 Distributed Implementation of GUI Applications

Upon the BTRON2 window system, the decomposed application core units can be dispersed into distributed environment [65]. They can run in parallel and on machines suitable for their efficient execution. This distributed processing provides two kinds of benefit.

First, it can provide a framework for the implementation of distributed applications in nature. For example, consider a software application for controlling air condition of rooms manually built in HFDS. It consists of controller units running on embedded computers, user interface units running on workstations, and units for hard computations running on CPU servers. For such applications, distributed implementation frameworks of GUI applications are necessary.

Second, distributed processing is one of the most popular approaches to obtain high computational power. This approach will be necessary because GUI is applied to applications requiring large processing power, for example CAD, mathematical processing, and high-quality DTP. Even in present, some of them cannot achieve sufficient performance without using very high-performance workstations.

For example, a CAD system can be implemented efficiently by decomposing into multiple parallel units such as interactive editing units, simulation units, file accessing units, and printing units. We can increase its execution performance by allowing these units to run in parallel and on machines suitable for their execution. The simulation units run on machines with high-performance FPUs, the file accessing units run on a file server machine, and so on.

Conventionally, most GUI applications, even constructed upon distributed environments, do not benefit from distributed processing. This is because they are tightly connected with window systems or GUI toolkits, and, therefore, they cannot be decomposed and dispersed in the distributed environment.

Conventionally, the framework available for their distributed processing is the server-client model in window systems [126, 137]. However, it supports only coarse-grained distribution. On the other hand, the BTRON2 window system supports the fine grained distributed implementation of GUI applications.

The main feature of our distributed processing framework is that its architecture can flexibly facilitate several classes of distributed implementations. To prove the flexibility, this section shows that our architecture can support all the three conceptual classes of parallel programming classified by Carriero and Gelernter [15]. They are (1) *result*, (2) *specialist*, and (3) *agenda* parallelism. Application developers can select the class of parallelism adequate for each application.

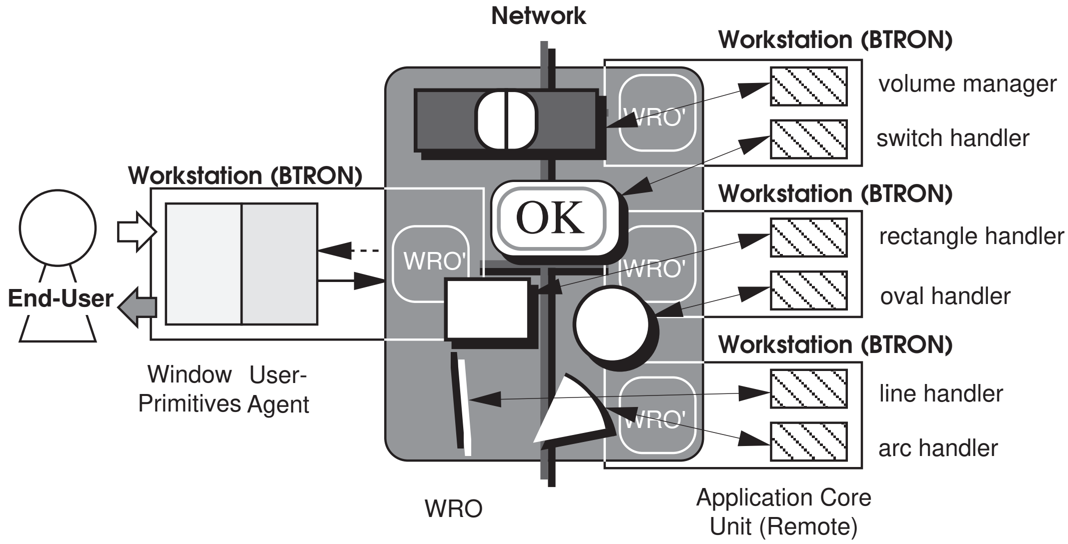


Figure 6.8: Result parallelism in GUI software

6.5.1 Result Parallelism

Result parallelism divides results of computation into many separate components, and proceeds to build all components in parallel. We can see result parallelism in object-based GUI systems. For example, a set of segmented graphical objects (the result) is divided into individual segments, and each segment is managed by an individual task/process. In our BTRON2 environment, this parallelism is adopted by application core processes each of which is invoked by abstract events notifying operations on a specific object (Figure 6.8).

6.5.2 Specialist Parallelism

Specialist parallelism is represented by a combination of special purpose parallel processing units. In GUI applications, this parallelism is applicable to menu command processing units. For example, in our graphic editor, a GUI application can be divided into a set of parallel units each of which provides only a specific function such as file save unit, printing unit, and spell-checking unit (Figure 6.3). In the BTRON2 environment, this parallelism is adopted by application core processes each of which handles abstract events indicating a specific command from users.

6.5.3 Agenda Parallelism

Agenda parallelism assembles a team of generalists that work in parallel, and assigns

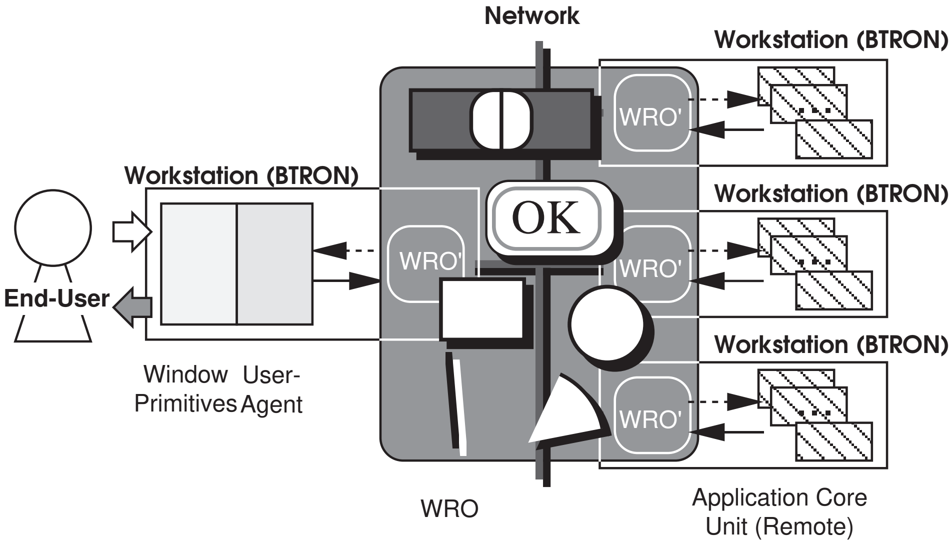


Figure 6.9: Agenda parallelism in GUI software

tasks to them. This parallelism can be used as a dynamic load balancing mechanism among multiple machines. We envision agenda parallelism in the BTRON2 window system as follows. A set of application core processes of a multi-user graphic editor is fully replicated at every participant's machine. Each task requested by a user's operation is assigned to any one of these replicas (Figure 6.9). For example, a task is assigned to the process on the machine with the lightest load. Another policy is that it is assigned to the process invoked first.

This architecture might resemble fully-replicated architecture [72, 25] in groupware systems. However it behaves in a completely different manner. The fully-replicated architecture executes a task multiple times, once for each replica. The agenda parallelism distributes each task to any one of these replicas.

6.6 A Sample Application: A Graphic Editor

This section describes an example of the decomposed and distributed implementation of GUI applications. The example is a graphic editor, whose appearance is shown in Figure 6.10.

In our architecture, the editor consists of loosely-coupled parallel cooperative processes sharing the same WRO. The user interface common to all applications, including input processing from PDs and feedback to the input, is implemented by user-agents. Furthermore, when multiple user-agents running on different machines share the same WRO, the editor works as a multi-user editor.

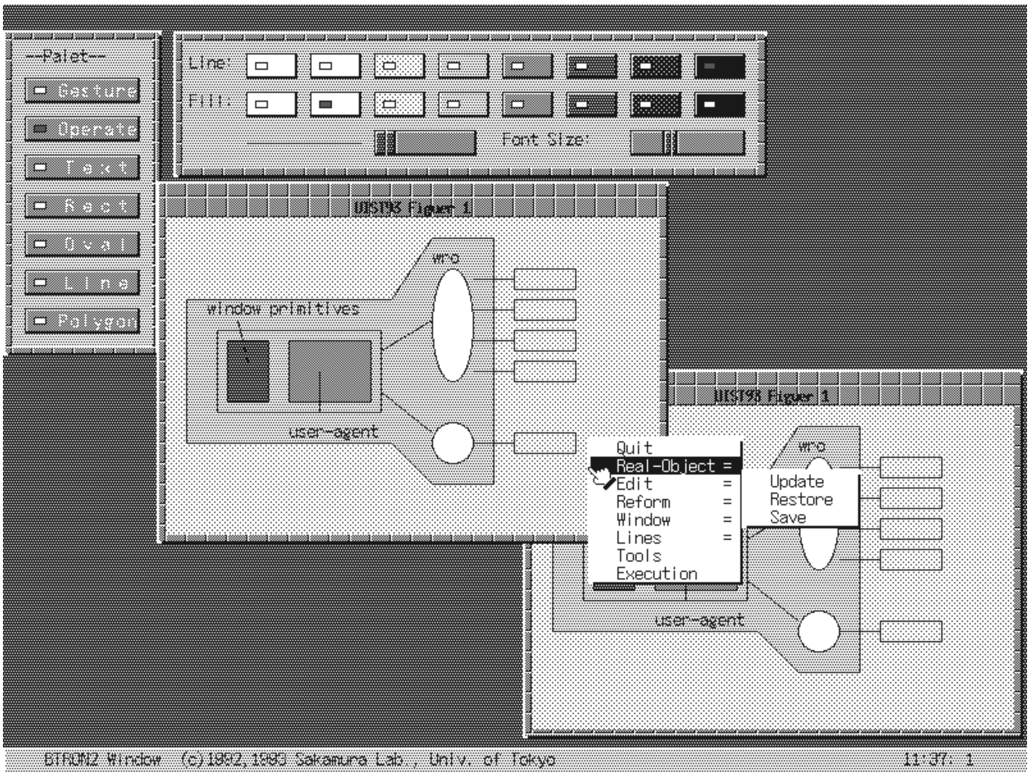


Figure 6.10: A graphic editor on the BTRON2 window system

Semantic actions invoked by users' operations such as menu selection and button pressing are implemented by individual multiple application core units. For example, such semantic actions are file saving, spell checking, printing, font selecting, line width selecting, editing mode selecting, etc. Each of these processes runs on the machine suitable for its execution. For example, user-agents run on machines with screens, file save processes and spell checking processes on a file server, and printing processes on a printer server machine (Figure 6.3).

With the benefit of its architectural extensiblity, this editor can work as a core component of GUI applications. For example, by connecting multiple user-agents with a WRO, the editor can work as a multi-user editor. Moreover, by connecting script programs with the WRO, the editor can work as a simple presentation tool with animation functionality. These examples of extension are used in Chapter 7 and 8.

6.7 Related Work

Recently, the cooperation of tools from different sources is widely recognized as a central issue. As described in Chapter 2, there are several systems supporting the cooperation. These systems can be classified into three groups: The first group is based on the concept of *tool integration*: a framework to combine multiple CASE tools. ToolTalk [138], Apple Event Manager of Macintosh [7], BMS [31], and the Esprit GIPE 2 project [20] are among them. Second, there are frameworks to support cooperation among GUI applications such as Component Construction Kit (CCK) [131] and Suite [22]. The third class is the one of dynamic document systems such as ADV framework [21]. These systems share several features with our application decomposition, but their objectives and required mechanisms are different from ours.

First, we examine tool integration systems. Its main objective is to provide *inter-operability* among multiple software development tools. The most popular example is the cooperation between a program editor and a symbolic debugger; Break points of debugged programs can be specified on the text editor. When the debugger catches an error of the program, the editor turns the color of the error position in the source code.

Thus, the granularity of cooperating units is just the conventional GUI application, such as editors, compilers, visual debuggers, and so on. In our application decomposition, the granularity is just a part of a component of the GUI applications, which are as large as callback procedures of conventional GUI toolkits.

Facility for the tool integration is, in most case, provided by toolkits that support rich message passing among application programs. Each application cooperates with other applications by exchange its semantic information using the tools. On the other hand, the BTRON2 window system supports user interfaces of application software, but provides no framework for the semantics of the application software. Thus, the BTRON2 window system does not support the exchange of semantic information among applications. It only supports the exchange of interaction object data among them.

The application decomposition supports another user interface facility. It is a facility for application core units to act on the same window. This facility is necessary for the collection of the units to behave as if a single application program. The BTRON2 window system provides this facility by making WROs shared memory based resources.

Conventionally, the toolkit named Suite [22] facilitates the distributed implementation of GUI applications. Suite provides an object layer supporting distributed, shared, and persistent objects. Each object can define remote procedures called from other objects. Using the remote procedures, Suite objects cooperate with each other. However, Suite differs from the framework of the BTRON2 window system in many points. First, it supports parallel programming based on only an object-oriented framework, which is a kind of result parallelism. Second, its assumed user interface style is a form-based user interface, not a direct manipulation GUI as the BTRON2 window system.

Dynamic multimedia document system requires the application cooperation [21]. In such systems, each component manages data of a specific media: e.g., a component to manage text data, segmented graphic data, animation data, video data, and so on. A dynamic document is composed from several media components. As the BTRON2 window system, this system provides a facility for multiple components to interface with an end-user through a single window. However, it only divides a window into several regions, and each component manages one of these regions. Also in this case, the application decomposition strategy is based on the region on a screen. It cannot assign multiple components to the same region on a window, but our system can.

6.8 Summary

This chapter has made an overview of GUI application programming upon the BTRON2 window system, and describes its three novel issues.

- Application decomposition
- Multi-language implementation
- Distributed implementation

Fundamentally, these features are caused by the high modularity in the parallel or distributed programming using distributed shared memories.

Chapter 7

Multi-user Interface in the BTRON2 Window System

This chapter describes the realization of functionality of real-time groupware upon the BTRON2 window system.

7.1 Introduction

Recently, there is a great increase of the requirement for systems to support collaboration among multiple users dispersed in separated places using workstations or personal computers connected with computer networks. Among such systems, real-time groupware [49] such as a multi-user editor or a distributed shared control panel is expected much.

However, the most serious problem of the real-time groupware is the difficulty of its implementation. Especially, the development of its user interface portion (*multi-user interface*) requires complicated mechanisms. For example, a multi-user editor requires a concurrent access control mechanism to support concurrent editing by multiple users. For the good responsiveness, data replication techniques and, further, consistency maintenance techniques among the replicated data are necessary.

These mechanisms are supported by *groupware toolkits* [34, 25, 103], which realize the shared window or the shared screen by sharing semantic data of the application software. For example, when we share a "graph" on a shared window, groupware toolkits share the original "numerical" data of the "graph" across multiple machines. In this dissertation, we call this data sharing method *shared semantics scheme*.

The advantage of the shared semantics scheme is its presentation flexibility: the toolkits can show the same data in a different appearance for each user. On the other hand, this approach has a defect that they force application programmers to use their built-in framework or a specific programming language.

The BTRON2 window system keeps the consistency among multiple windows on different machines by sharing interaction object data using Window Real-Objects (WROs). We call this technique as *shared interaction object scheme* [67]. In this scheme, the abstraction level of shared data is lower than the one of the shared semantics scheme. The shared interaction object scheme does not limit the framework and the programming language that programmers use. For example, in building real-time groupware upon the BTRON2 window system, we can use multiple programming languages such as C and Lisp (Scheme). Also, the semantic data structure of the application can be implemented using either the `struct` data structure of C or the list structure of Lisp. The rest of this chapter shows that the shared interaction object scheme can realize the most of the multi-user functionality the shared semantic scheme can.

7.2 Multi-user Extension of the BTRON HMI Specification

In adopting multi-user interface functionality in the BTRON2 window system, we have, firstly, extended the BTRON Human-Machine Interface Specification [56].

This extension contains three issues as follows:

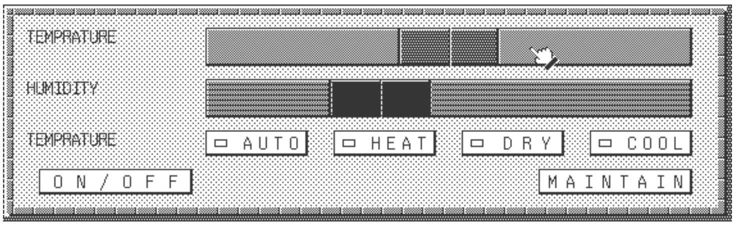
1. *Locking State* is newly added to states of interaction objects. This state specifies that interaction object of the state is being operated by another user. This technique was introduced by Newman et. al. [91]

2. Two kinds of *coupling* can be defined for interaction objects shared by multiple users.

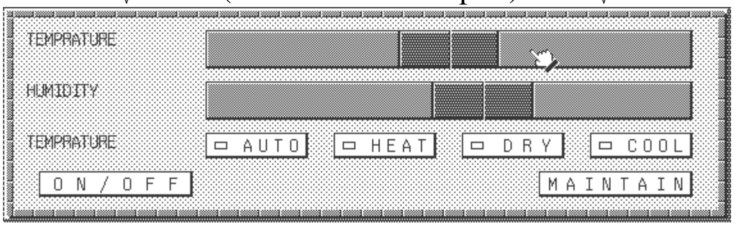
3. Remote pointers are available upon the BTRON2 window system.

7.2.1 Locking State

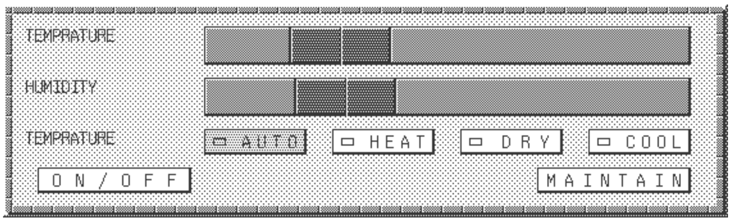
A locking state is a new state of an interaction object that indicates that another user is now operating the object. The BTRON window system shows the state by turning the color of the object into *gray* (Figure 7.1). Users cannot operate the object that another user is operating. The graying works as a sign showing inability to operate the object. A WRO refuses any operations by users to grayed objects.



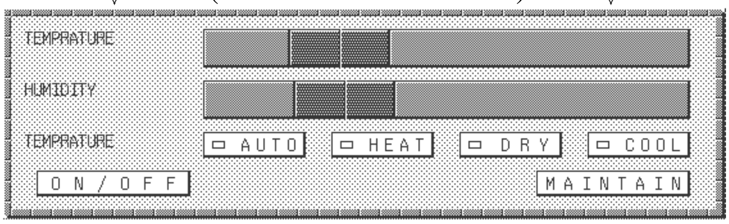
⇓ (A locked volume part) ⇓



(Unlocked)



⇓ (A locked “auto” button) ⇓



(Unlocked)

Figure 7.1: Locking state

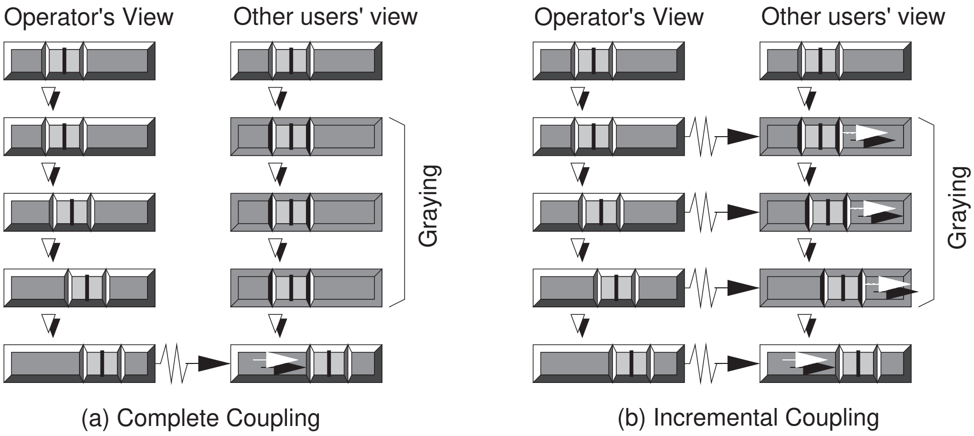


Figure 7.2: Coupling

7.2.2 Coupling

Multi-user interface keeps the consistency among interaction objects displayed on different windows, in some level. This functionality is called *coupling*. There are several kinds of timing in propagating an update of an object to other objects. Consider an example that multiple users are sharing a volume part. When a user scrolls the volume smoothly, the volume on every user's display is always consistent (*incremental coupling*) or only at releasing PD button (*complete coupling*) (Figure 7.2).

Which coupling is the better depends on applications, performance of target machines/networks, and so on. Thus, we provide both the incremental and complete couplings in the BTRON Human-Machine Interface Specification.

Dewan and Choudhary have discussed the coupling of the multi-user interface precisely [23]. Their groupware toolkit "Suite" provides three classes of coupling: *incremental*, *complete*, and *transmit*. The first two are the same as the ones of the BTRON2 window system. The last one is a kind of coupling that propagates operations of interaction objects when a user explicitly demands the propagation by pushing a transmit switch, etc.

The BTRON2 window system can realize this by combining interaction objects with local attribute, an application core unit managing the consistency among these objects, and a "transmit" button switch. When a user operates an interaction object, the user-agent updates only the interaction object that is local to the user. Next, when the user pushes the transmit button switch, the application core unit receives an abstract event and updates all the coupled interaction objects.

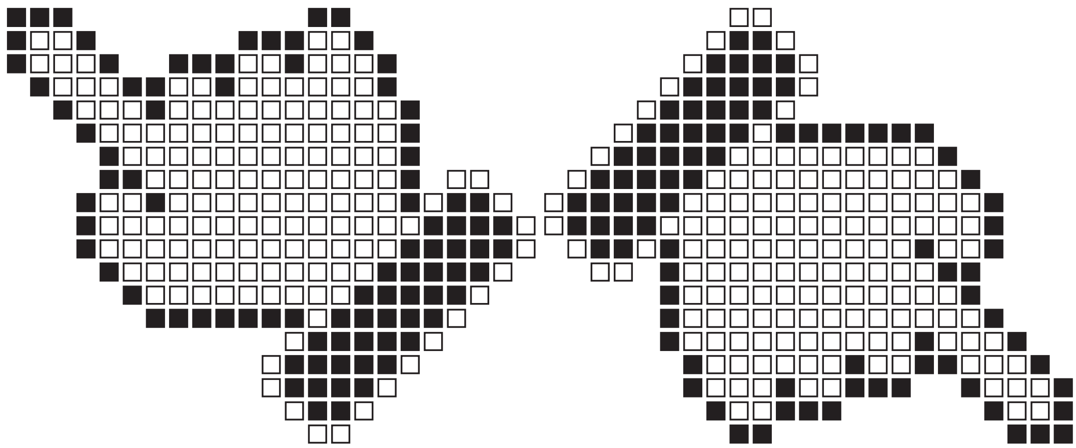


Figure 7.3: Normal pointer and remote pointer (pointing)

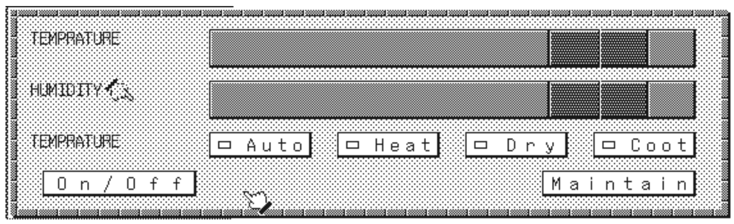


Figure 7.4: Remote pointer on a shared panel

7.2.3 Remote Pointers

To show actions of other users, the BTRON2 window system displays other users' PD pointers on shared windows. The pointers are called *remote pointers*. The system helps a user to distinguish his/her own pointer and others by tuning other users' pointers in 180 degree (Figure 7.3 and 7.4).

7.3 Realization of Multi-user Interface Functionality

This section describes how the BTRON2 window system handles multi-user interface techniques described in the previous section.

7.3.1 Conflict Handling

The BTRON2 window system resolves *conflicts* in multi-user interfaces in two ways. The first case is the conflict with an *instant* operation, such as clicking a button switch,

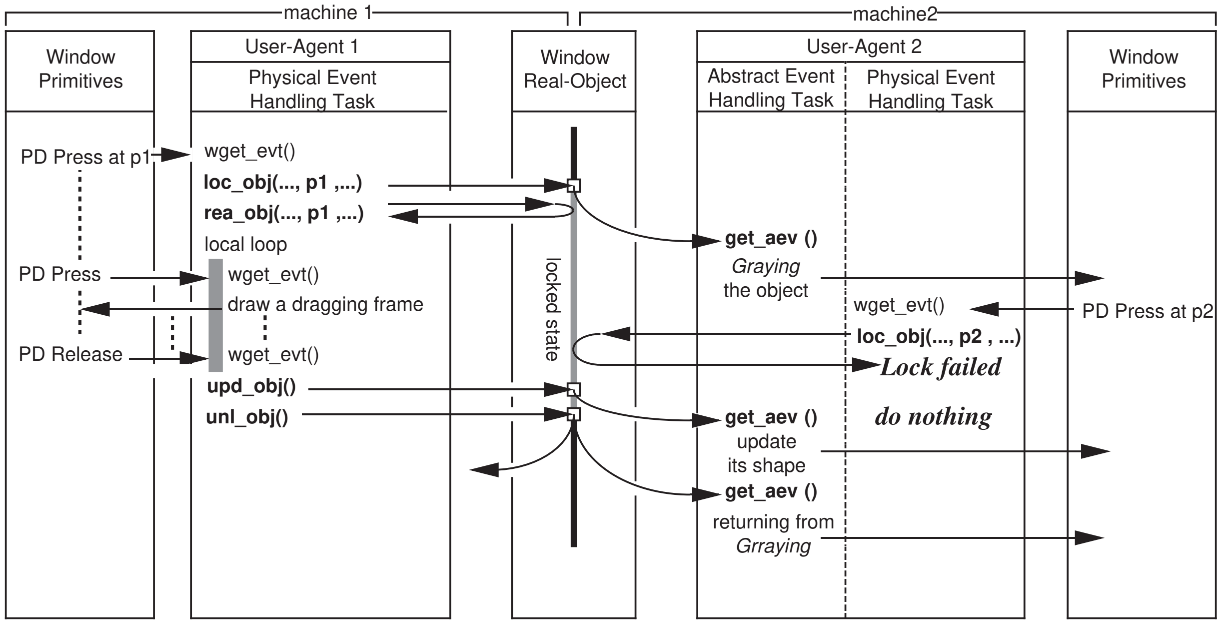


Figure 7.5: An example of conflict resolving protocol

pushing a key, selecting a graphical object, etc. The instant operation is implemented by one window call. Thus, this type of conflicts are easily resolved by the window call serialization facility of the distribution server of a master WRO. For example, a conflict between two click operations on a button switch is resolved by serializing `ope_obj()` window calls implementing the clicking operations.

The second case is the conflicts with a long-term operation such as dragging a volume knob, moving an interaction object, and reshaping an object. This type of conflicts are resolved by the transactive protocol between a user-agent and a WRO. Before the long-term operation, the user-agent always locks the operated objects by issuing the `loc_obj()` call. After the operation, it unlocks them by issuing the `unl_obj()` call (Section 5.4.2).

Operations by window calls to locked objects are always refused by WROs. Thus, during the long-term operation, any conflicted operations are all refused by WROs. For example, as Figure 7.5 illustrates, when two long-term operations conflict, the `loc_obj()` window call of the latter fails. Then, the user-agent of the latter finds that the object is locked, and it immediately aborts the long-term operations.

By the implicit locking facility, the locked objects are displayed in *gray*. When users' operations are refused because of conflicts, they know that they cannot operate the grayed object.

```
    :
switch(etype = wget_evt()){  /* main window event loop */
    :
case PD_PRESS:
    if(loc_obj() == E_OK) {
        while((etype = wget_evt()) != PD_RELEASE){
            draw_knob_frame();
        }
        upd_obj();
        unl_obj();
    }
    :
}
```

Figure 7.6: Program code summary implementing complete coupling (precise descriptions of parameters are omitted)

7.3.2 Coupling

The BTRON HMI Specification contains two kinds of coupling: complete coupling and incremental coupling. We can implement both coupling upon the BTRON2 window system.

Complete coupling

A user-agent can implement the complete coupling by issuing an `upd_obj()` or `ope_obj()` window call only when a user finishes one logical operation. For example, while a user drags a knob of a volume part, the user-agent only draws a frame of the volume knob as feedback of the dragging. It issues an `upd_obj()` call just when it receives a PD release event from a pointing device (Figure 7.6).

Incremental coupling

A user-agent can implement the incremental coupling by issuing periodically `upd_obj()` or `ope_obj()` window calls during user's operation. In the volume dragging example, the user-agent continues both drawing the volume knob at the PD position and issuing `upd_obj()` window calls until it receives a PD release event (Figure 7.6). In our implementation of user-agent, Physical-event Handler Task issues the `upd_obj()` call, Abstract-event Handler Task receives abstract events of the update and draw the volume knob.

```
switch(etype = wget_evt()){  /* main window event loop */
    :
case PD_PRESS:
    rea_obj();
    :
    if(loc_obj() == E_OK) {
        while((etype = wget_evt()) != PD_RELEASE){
            upd_obj();
                :
        }
        unl_obj();
    }
    :
}
```

Figure 7.7: Program code summary implementing incremental coupling (precise descriptions of parameters are omitted)

7.3.3 Remote Pointer

To support user-agents to implement remote pointers, a WRO provides a facility to store the information of remote pointers. Each user-agent sharing the WRO periodically stores its PD cursor position to it. This periodical storing is handled by the cyclic task of the user-agent. The period is defined by each user-agent according to the performance of collaborating machines. When only high performance machines share the WRO, each user-agent stores its PD cursor position in a very short period for smooth move. On the contrary, when there is a slow machine in the environment, the cycle should be as slow as possible. The multitask implementation of the user-agent makes such performance adjustment very easy.

On the other hand, the user-agent periodically retrieves the information of all remote pointers, and displays them on window. This visualization is also handled by the cyclic task, and the visualization cycle time is also easily adjusted according to the performance of their environment.

7.3.4 User Interface Independence

User interface independence is also one of the most important issues for multi-user interface. The user interface independence provides *different views* of the same semantic data. Here, the term *different view* contains several classes, and we classify it into three classes: *geographical difference*, *media difference*, and *object difference*.

Geographical difference is the difference of several physical attributes of windows, such as magnification, position on a screen, and scroll positions. The BTRON2 window system supports the geographical difference transparently from GUI applications. Since each participant's machine sets these attributes, each user can set these attributes as he/she likes. Hence, each participant can set his/her own setting of these attributes. Generally, this independence can be supported easily by every multi-user interface architecture that replicates its abstract device component and the interaction object component on each participant machine.

Media difference is the second class of the user interface independence. For example, a value of a volume part is presented in the graphic media for a user, and in the speech sound media for another user. For example, the value of the volume part is read as "Value of the volume part **A** is just seventy-three." We can also implement this easily by replacing the user-agent with a special *auditory user-agent*. This replacement can give speech user interfaces to GUI applications independently of the applications.

This media difference has another application. Users can change the design of interaction objects as they prefer by tailoring their own user-agents. For example, user **A** access a real-time groupware with the interaction object set in the Motif design guidelines[94], and user **B** access the same real-time groupware with the one of the OPEN LOOK design guidelines[139].

Object difference is the difference of interaction objects themselves: for example, a user sees room temperature via numerical digits, and another user sees it via a volume part. The BTRON2 window system does not realize this difference transparently. It only provides a primitive that helps application programmers to implement the object difference. This primitive is the *locality attribute*. Interaction objects with the local attribute, called *local object*, can be seen by only one user.

We consider this primitive is sufficient for the support of the object difference because of the following two reasons. First, practically, the situation in which the different view of interaction objects is used is rare. Second, with this primitive and the window call/abstract event programming interface, programmers can easily [1] implement the consistent different interaction objects. From these reasons, the BTRON2 window system need not support the object difference by limiting the framework or language for the description of semantic data structure of application program.

7.4 Related Work

Multi-user interfaces of conventional real-time groupware systems are implemented using either *shared semantics*, *shared window*, or *fully-replicated architecture*. Additionally, there are systems that provide both semantic sharing facilities and shared window

[1] An obvious implementation is that there are a pair of different objects with local attribute in a WRO. An application core units can maintain the consistensy easily. When the unit receives an update event from one of the pair, it updates the other pair consistently.

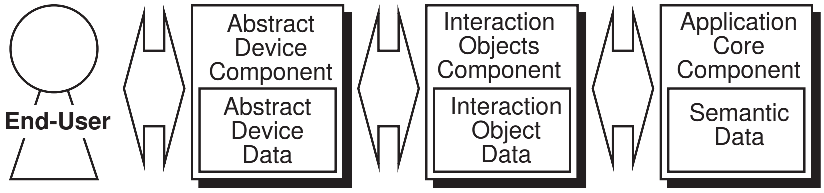


Figure 7.8: Simplified GUI software architecture model

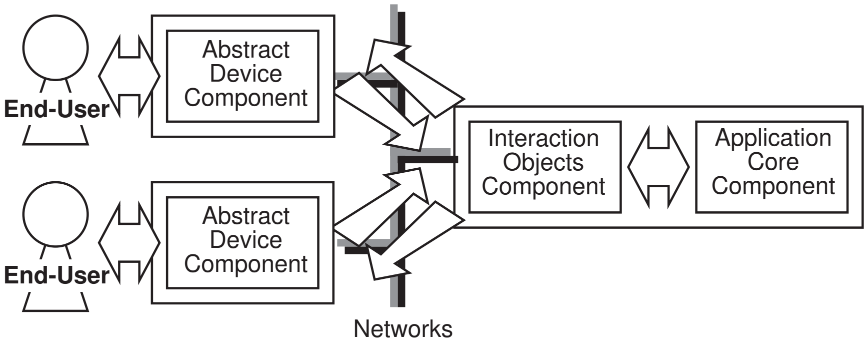


Figure 7.9: Shared window scheme

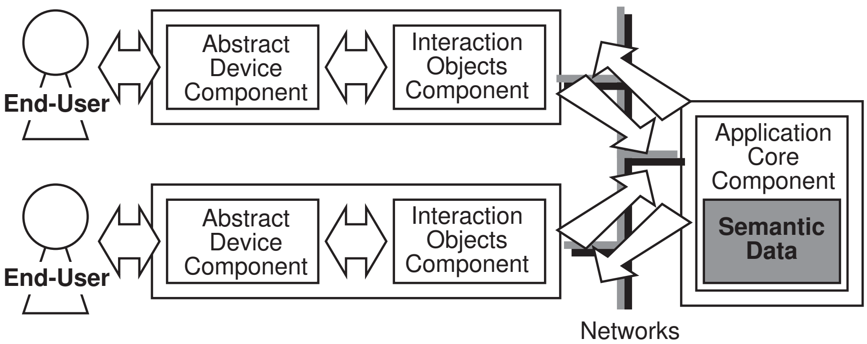


Figure 7.10: Shared semantics scheme

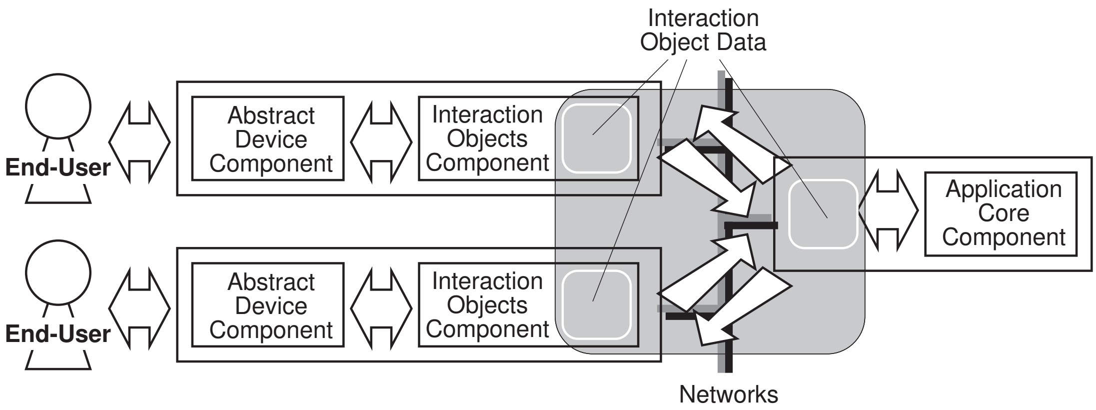


Figure 7.11: Shared interaction object scheme

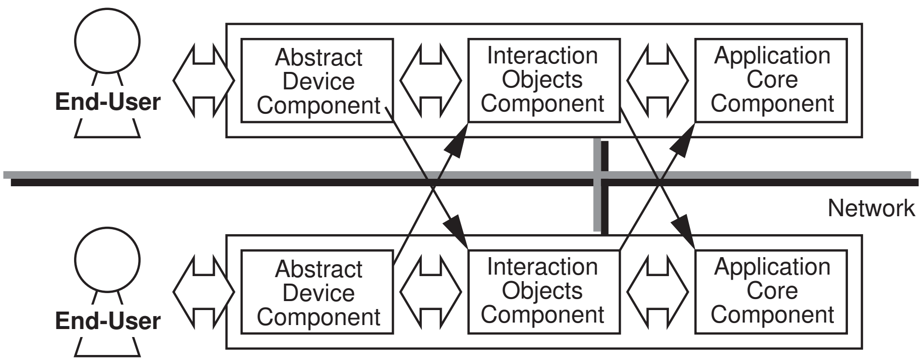


Figure 7.12: Fully-replicated architecture

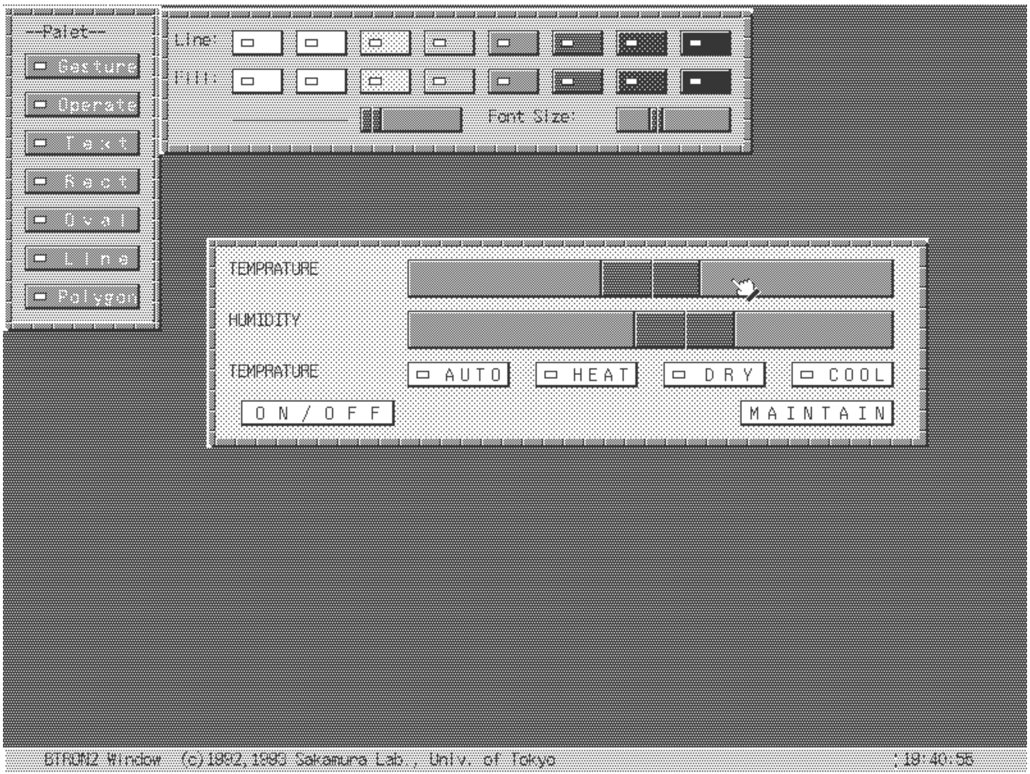


Figure 7.13: A sample control panel for an air conditioner

facilities [1, 90]. Data sharing mechanisms of these schemes differ from the one of the BTRON2 window system (*shared interaction object scheme*).

The shared semantics scheme is adopted by many conventional groupware support systems such as *LIZA* [34], *RENDEZVOUS* [98, 97], *Suite* [24, 23, 25], and *TEIDAN* [103]. This scheme realizes multi-user interfaces by sharing semantics data among multiple workstations across networks. However, sharing techniques are different among these: *Teidan* adopts distributed sharing and other systems adopt centralized sharing.

In the simplified [2] GUI software architecture model (Figure 7.8), the data sharing is implemented in the application core component (Figure 7.10). For example, the panel illustrated in Figure 7.13 is a shared control panel of air conditioner in an office room. To keep consistency of the value of a volume part for room temperature setting among different machines, the shared semantics scheme shares the variable representing the room temperature setting. On the other hand, the shared interaction object scheme shares the volume part.

The advantage of the shared semantics scheme is that it can support wide range of user interface independence transparently from applications. As discussed in the previous section, the shared interaction object scheme can also realize this feature by adopting locality attribute to interaction objects.

The defect of the shared semantics scheme is that the platform systems limit the way of implementation of the application core component of groupware to manage the semantics data. Even the application core component of groupware must be implemented using the framework given by the toolkits. Application programmers cannot select programming language, implementation framework, module organization, etc.

For example, *TEIDAN* provides an extended *Model-View-Controller* framework. Programmers of TEIDAN must implement the application on component of groupware as Models of the MVC.

LIZA also provides a framework called *kernel objects* based on concurrent object-oriented framework. LIZA provides a special *kernel-image* [3] *interface* to support the implementation of real-time groupware.

The framework of *Suite* does not limit the programming style of application core components so strong. Suite maintains the relationship among interaction objects and semantic data using active variables. In case of Suite, interaction object data take the form of active variables. The application core component of the Suite must provide Suite callbacks that handle events from active variables. Suite supports multi-user interface

[2]The dialogue control component is omitted.

[3]The LIZA framework for the interaction objects component is *image object*, and the interface among image objects and kernel objects is the kernel-image interface.

		Shared Window Scheme	Shared Interaction Object Scheme	Shared Semantics Scheme
WYSIWIS Windows		○	○	○
Remote Pointers		×	○	○
Implicit Locking		×	○	○
Flexible Coupling		×	○	○
UI Independence	Geographical Independence	○	○	○
	Media Independence	×	○	○
	Object Independence	×	×	○

Table 7.1: Functional comparison among multi-user interface implementation schemes

functionality in the Suite Callbacks and Suite Calls, which are the access methods to active variables of the interaction objects.

Upon *Rendezvous*, programmers must specify the application core component with the object framework of MEI UIMS. On the other hand, it provides very strong mechanism for maintaining the consistency among centralized semantic data and replicated interaction object data. It provides a *constraint* framework for declarative specification and a *rule* framework for procedural specification.

Our shared interaction object scheme realizes the data sharing in less abstract level (Figure 7.11). Thus, the data sharing mechanism is independent of the framework or the programming language of the application core portion of the application.

Next, another scheme is the shared window scheme. In this scheme, a GUI toolkit and an application core portion of the software locate on one workstation. The interaction object component broadcasts drawing operations to multiple abstract device components, and conversely, it collects users' inputs from all the abstract device components (Figure 7.9). *XTV* [17] and *shX* [4] are representative examples of conventional systems of this scheme.

The shared window scheme realizes the data sharing in less abstract level than our shared interaction object level. Thus, it is more independent from the application core layer of groupware. However, it is impossible to control user interface independence flexibly. It cannot realize even the geographical difference such as different scroll position, the magnification, or the position of shared windows. It can realize only the WYSIWIS[4]-type of shared windows (Table 7.1). Therefore, as [135] reports, it is impossible to build practical real-time groupware using only the shared window scheme.

Lastly, *fully-replicated architecture* is also a classical scheme for building real-time groupware [72]. *DistEdit* [55] and *GROVE* [28] are groupware tools of the fully-replicated architecture. As Dewan and Choudhary have discussed [25], the fully-replicated architecture "*meets the requirements of flexible placement, concurrent scheduling, and error independence*." "*However, it executes semantic operations multiple times, once for each copy of the application.*" In our opinion, the essence of the replicated architecture is that it is one of the easiest ways to modify a single user application program into a multi-user application program.

7.5 Summary

This chapter has described multi-user interface issues of the BTRON2 window system. Its main feature is that the consistency among multiple shared windows is maintained by the interaction object component of GUI software. We have called this scheme as *shared interaction object scheme*.

The data sharing facilities are less abstract than most groupware toolkits of shared semantics scheme. With the benefit of the less abstractness, the data sharing mechanism is independent of the application core component of the GUI software. Application programmers can build a GUI application using suitable programming languages and programming frameworks for its implementation.

As well as the shared semantics scheme, this chapter has shown that the shared interaction object scheme can handle important multi-user interface functionality such as conflict resolving, flexible coupling, remote pointers, and several classes of user interface independence.

[4]What You See Is What **I** See

Chapter 8

Script Processing in the BTRON2 Window System

8.1 Introduction

Direct manipulation [127] is an HCI technique that represents internal data of computers with graphical objects and to allow users to manipulate the objects directly with pointing devices. *Desktop metaphor* is a popular metaphor of the direct manipulation that displays internal data of computers with graphical objects imitating the desktop of real offices. Mostly, the desktop metaphor or the direct manipulation can offer more human-friendly user interfaces than old-fashioned command languages. However, the command language interface is sometimes suitable.

The first case is to repeat a fixed operation sequence in many times. For example, `csh` of Unix can issue instructions to print all documents stored in a directory with a simple program as follows:

```
% foreach doc (*.dvi)
? dvi2ps $doc | lpr
? end
```

On the other hand, to execute the equivalent operations in the direct manipulation environments, a user must repeat an annoying operation sequence as follows:

1. double-clicking a document file icon to open the file with a word processor
2. selecting its menu item “print” to print the document
3. selecting the menu item “open files” to open the next document
4. selecting the menu item “print” again

5. repeating the above step 3 and 4

Moreover, the command language user interface offers another advantage. It can deal with new kinds of tasks by combining conventional programs. For example, a task to collect statistics of words appearing in a document is dealt by the combination of four kinds of well-known programs as follows.

```
% delatex sample.tex | tr '[A-Z]' '[a-z]' | sort | uniq -c | sort
```

On the contrary, in the desktop environments, a new program for the statistics must be made.

From these considerations, we can notice that the desktop environments are inferior to the command language user interface environments in the following two points: (1) specifying a repeated fixed operation sequence, and (2) dealing with a new task by combining conventional programs

Thus, in several systems such as the X-Window System and the NeXT Step, a command language interface and a direct manipulation GUI are coexisting. However, this easy solution contains two problems. First, two individual programs, which are an interactive program and a batch program, are required for each function. Second, users are required to understand two machine models: one is the desktop model and another is more physical model of computer for batch processing.

In this chapter, we propose a batch processing metaphor called "*post-it (fusen) metaphor*", which matches naturally with the BTRON desktop metaphor. The post-it metaphor is a batch processing model applied to document-like objects on windows or panels. Applications of the post-it metaphor are (1) batch processing, (2) animation, (3) automatic demonstrations, (4) extending/customizing mechanisms of GUI applications, and so on. The rest of this chapter shows the detail of the post-it metaphor and its implementation upon the BTRON2 window system.

8.2 Post-It Metaphor

In the real world, post-it [1] is a small piece of paper attached temporarily on books or documents, and used to record instructions. For example, when one's boss is away from his office, he/she puts a document attached with a post-it that contains a message that "please read this document, and if OK, please make your sign at the bottom of this document."

[1]"Fusen" in Japanese

Desktop metaphor gives internal data of computers a view of real office desktop. For example, the data stored in a file is visualized as a document. Thus, a sequence of instruction (e.g., a script program) applied to internal data of computes is naturally considered as the post-it on a document. Here, post-it metaphor is defined as a batch processing model in desktop GUI environments in which scripts are applied to interaction objects on a window to control application programs.

Next, we show the summary of an operation sequence of batch processing in the post-it metaphor. For the purpose, here, we use an example to print all document files in a directory.

1. A user describes a script program to print all document files in a directory, and registers it as a "post-it" program.
2. A post-it is represented with a small window on which the script program name and switches to invoke the program are displayed.
3. The user clicks a directory icon, and opens the directory including the document files to a window.
4. Using a pointing device, the user selects all document files in the window.
5. The user drags and drops the post-it window upon the window.
6. The user pushes the "OK" button on the post-it window, then the script program is invoked.
7. The script program opens each document file with its standard application software (a word processor here).
8. Then, it selects the menu item "print", and the document is printed out.
9. The script program repeats the step 7 and 8 for each document file.

The script program does not have to know the way to print each document file by itself. It only invokes the document-printing routine in the word processor by operating WROs, and, indirectly, lets the routines print the document. Other than the above example, the post-it metaphor can be applied to *animation*, *automatic demonstration* of interactive software, and so forth.

8.3 Scripting Mechanism of Script Processing

For scripting, the following two issues are important. The first issue is how a script program operates or controls GUI applications. The second issue is what kind of features the script language should provide.

The BTRON2 GUI environment realizes scripting using the cooperation mechanism among decomposed GUI application programs. Scripting is the cooperation among a script language interpreter program and interactive GUI application programs. The script language interpreter neither communicates nor invokes application programs directly. It communicates and invokes indirectly other application programs by operating interaction objects in WROs using window calls. The operations cause to generate abstract events, which transfer controls to the application programs.

The method with which a script language interpreter invokes GUI applications is the same as the one with which a user-agent invokes GUI applications according to users' operations. In this mechanism, interactive processing and script processing are integratedly implemented by one mechanism.

For its script programming language, we have adopted *scheme* [102] because there are available portable scheme interpreters in free. Current our research is mainly devoted to the cooperation mechanism between the script interpreter and interactive application programs, and to the interface to describe and invoke the script programs. Thus, our main motivation to adopt the scheme interpreter is its modest execution performance and portability on BTRON2. Also, it satisfies requirements for command language [74]. In the future, we are planning to build a script language suitable for the post-it metaphor, called *TACL* (TRON Application Control-flow Language) [110, 57].

Next, we explain the script processing mechanism more precisely using an example referred in the previous section, which prints all the document files in a directory.

1. The script program issues an `ope_obj()` window call equivalent to double-clicking a file icon (the pictogram of a file virtual object), then, an abstract event notifying the operation is generated and sent to a data land editor (Figure 8.1-1).

2. The data-land editor [2] opens the operated document file with its standard application program. In this case, the program is a word processor. The word processor generates a WRO, and sets document data read from the document file and some other interaction objects including a pop-up menu (Figure 8.1-2).

3. Then the script language interpreter issues an `ope_obj()` window call to select

[2] "Data-land editor" is the name of a directory editor of BTRON, which is similar to Finder of Macintosh.

the "print" item of the popup menu. Then, an abstract event to notify the selection is generated and sent to the word processor (Figure 8.1-3).

4. The word processor reads the document data from the WRO, translates it into printing format, and sends it to a printing device (Figure 8.1-4).

8.4 A Scripting Example: Algorithm Animation

Animation is an application of the post-it metaphor since it is considered as a kind of batch processing that operates graphical objects in a window or a panel. Figure 8.2 is a scene of algorithm animation of tower of Hanoi [146]. This animation is realized by the cooperation between a simple graphic editor and the scheme script language interpreter. No animation function is provided by the editor.

To make the animation, first, users draw towers, plates, and other explanatory texts using the editor. Next, they enter a script program to move these graphic segments on the basis of the *tower of Hanoi*'s algorithm. Then, users can animate the plates among the towers. The animation script is easily written with 219 lines in the scheme source code. The source code is described in Appendix C.

8.5 Discussion

As in the above animation example, updating WROs frequently brings a new problem: a *version problem*. When a WRO is updated, abstract events notifying the update are generated. However, when the events are processed, the state of the WRO is not always the same as the state at the point of the update time. Other programs may change the WRO before. This is the general problem of asynchronous message passing communication.

Here, consider the case of the animation sequence. The script program moves a graphical object "**A**" to the position p_1. Abstract events notifying the update are generated and sent to a user-agent. Before the user-agent processes the event, the script program moves **A** to p_2. When the user-agent retrieves information of **A** to draw it at p_1, the information of **A** has already been modified to p_2 by the above operation.

In the current BTRON2 window system, application programmers are responsible for this version problem. They should consider that an abstract event conveys only hint information on an update of a WRO. As future work, we are planning to adopt versions in WROs. In the system, abstract events contain time stamps, which are used with the `rea_obj()` window calls. The `rea_obj()` window call can retrieve interaction objects of the specified time stamp.

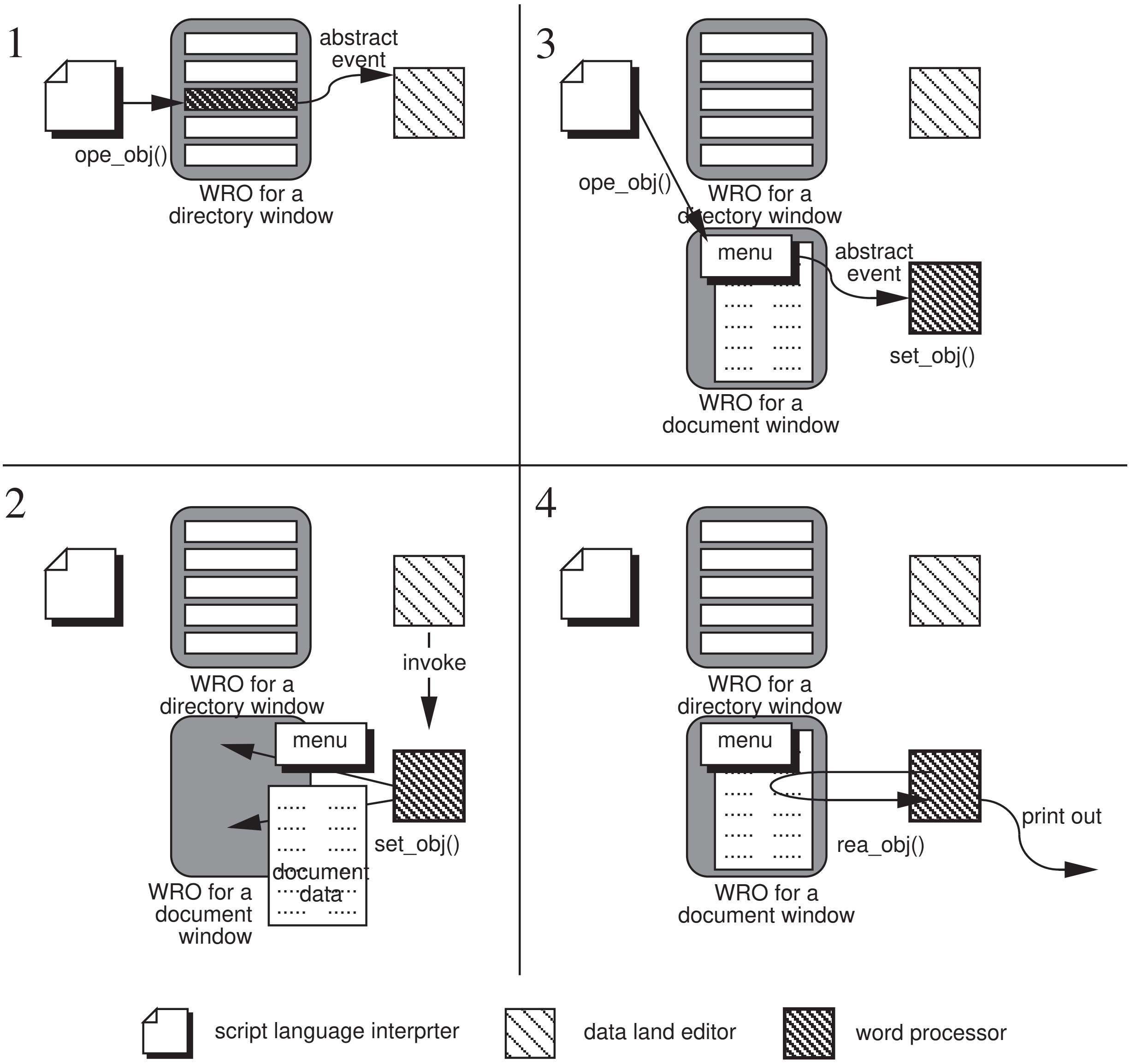


Figure 8.1: An action sequence of the scripting example

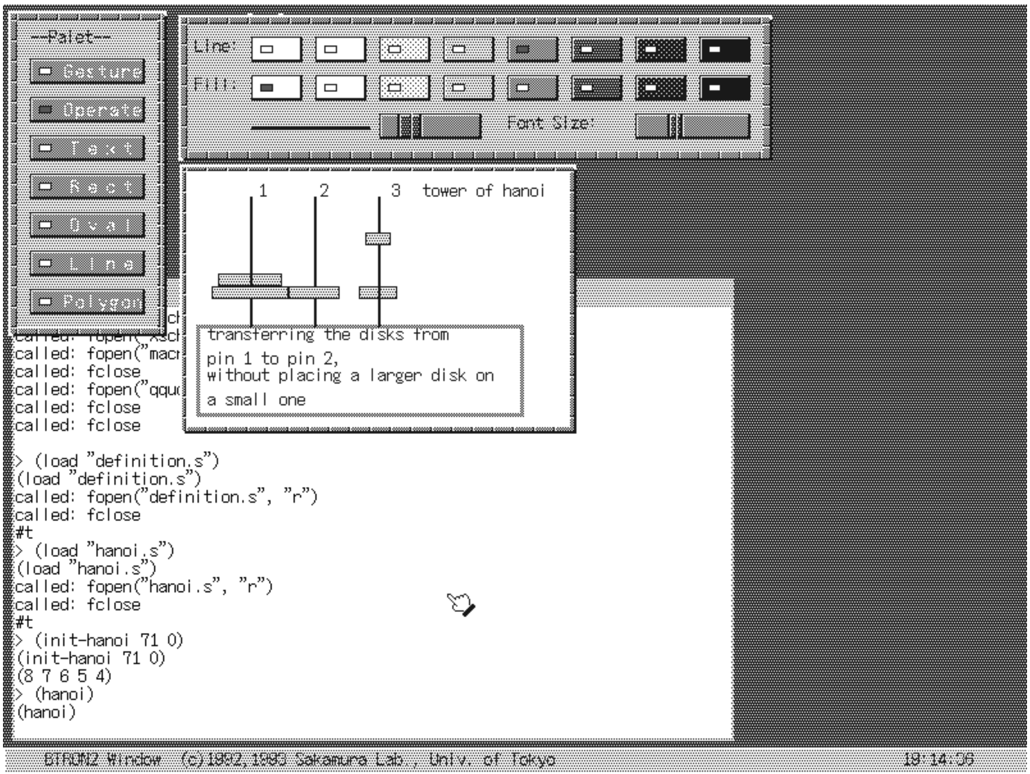


Figure 8.2: Algorithm animation of tower of Hanoi on the extended sample graphic editor

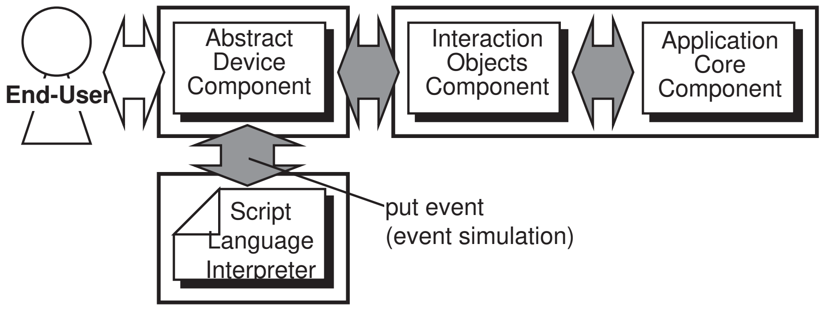


Figure 8.3: Scripting architecture by the event simulation technique

8.6 Related Work

Every scripting system controls interactive application programs from the external. Scripting systems can be classified in terms of how abstract they control application programs. There are scripting systems that control GUI applications by accessing its abstract device component or its application core component. The former method includes event simulation systems, automatic demonstration systems, and event macro systems. The latter method is a popular mechanism for toolkits of the application cooperation and the tool integration. On the other hand, our mechanism controls GUI applications in the intermediate component of these: the interaction object component. In the following, we compare other two mechanisms with ours.

Automatic demonstration systems (BTRON1 [78]), event simulation systems, and event macro systems generate a sequence of window events that simulate physical user's action. These systems access the abstract device component of GUI applications, and use window event mechanisms as its data and control transferring mechanism among a scripting program and application programs (Figure 8.3).

The main advantage of this method is that each application program invoked from the script program needs not support application collaboration functionality at all. On the contrary, there are several defects. First, it is hard to program scripts because the abstraction of script description is low. Second, the behavior of the scripts depends on the physical position of interaction objects and windows on a display. Third, even in using non-demonstration applications, the operation sequence is always displayed on a screen. This causes graphic processing, and the performance of the batch processing, therefore, becomes worse.

There are many toolkits managing cooperation among GUI applications such as BMS [31], ToolTalk [51, 138], and Macintosh AppleEvent Manager [6], and tool

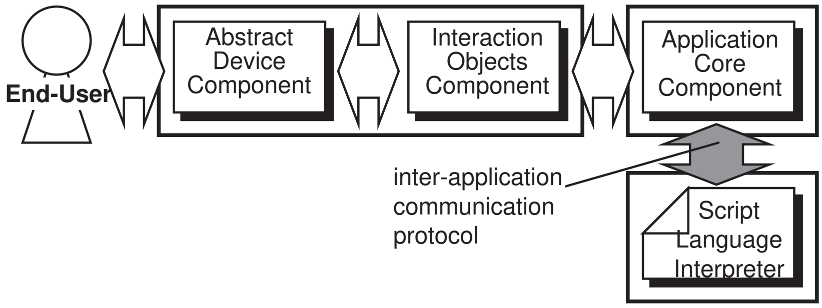


Figure 8.4: Scripting architecture by application cooperation toolkits

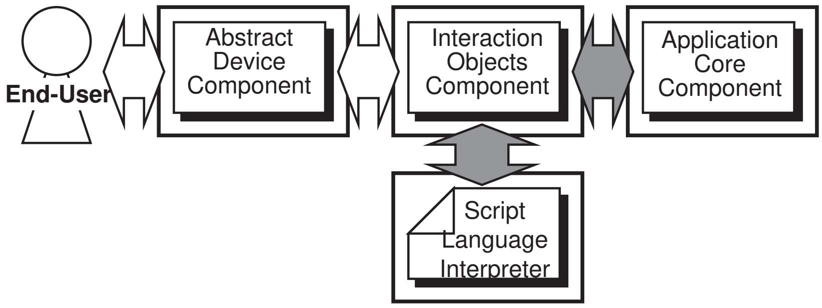


Figure 8.5: Scripting architecture of BTRON2

integration systems in the software engineering [21, 20, 32]. These systems facilitate the implementation of semantic data communication and control transfer among multiple application programs (Figure 8.4). In these systems, script programs access the application core component of GUI applications, and exchange semantic data to control them.

The main advantage of these systems is that they can implement flexible cooperation among multiple application programs. However, there are several defects. First, each application program invoked from the script program must support application collaboration functionality explicitly. Therefore, programmers must provide two interfaces for each functionality: interface to user's operations and script programs.

Compared with many other scripting facilities, we can notice that the scripting mechanism of the BTRON2 window system contains two main features: one is that its communication mechanism is based on the shared memory model. Another feature is that the abstraction level of communication data is the interaction object layer, which is just among the one of event simulation systems and application cooperation toolkits (Figure 8.5). This accomplishes advantages of both event simulation systems and application cooperation toolkits at the same time. For example, no special code only for the scripting

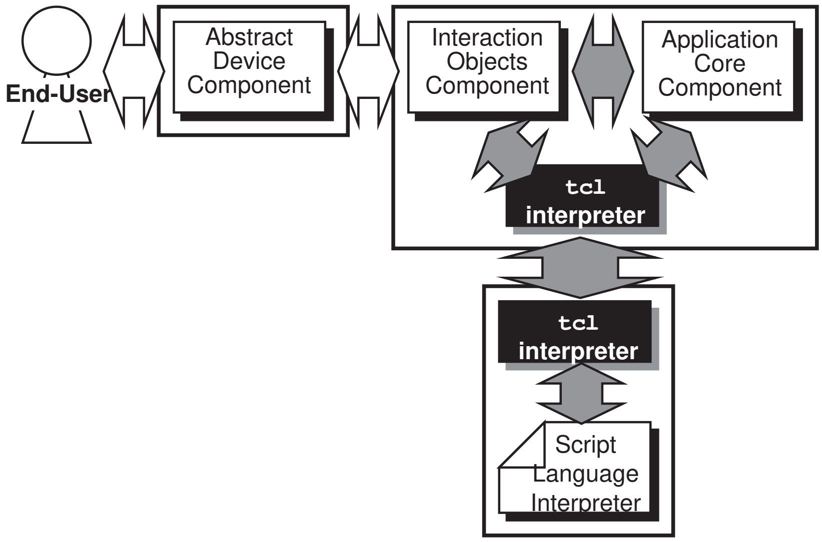


Figure 8.6: Scripting architecture of `tcl`

interface. Unlike event simulation systems, behavior of script processing is independent of location of control parts or windows. Lastly, by controlling abstract event distribution, script programs can display/hide its behavior on/from a screen.

An interesting approach to the scripting is to use an embedded language such as *tcl* [96]. *Tcl* provides an RPC-like mechanism for the cooperation among multiple application programs (Figure 8.6). The tcl script program can invoke every application linked with the tcl library.

8.7 Summary

This chapter has described the scripting user interface and its mechanisms of the BTRON2 window system. We have proposed the post-it metaphor that is scripting user interface model naturally unified with desktop environments. This metaphor provides two advantages over the simple coexistence of a command language interface and a desktop environment. First, user's view of the internal data of computers is common between the command language and the direct manipulation. Second, special application programs for batch processing primitives are not required since script programs operate interactive GUI programs as its primitives.

Next, we have described the implementation mechanism of the post-it metaphor. The most important feature is that script programs control GUI application programs by communicating with their interaction object components. In the BTRON2 window system, the interaction object component is a WRO. This accomplishes advantages of both event simulation systems and application cooperation toolkits at the same time. GUI applications require no special code only for being controlled by script programs. The script operations are enough abstract to make the script operations independent of locations of control parts or windows.

Chapter 9

Evaluation

We have experimentally implemented the BTRON2 window system, upon which we have built a graphic editor as its sample application. Using the system, we have measured the performance of the BTRON2 window system to verify the effectiveness of our implementation mechanisms. This chapter shows the information on the prototype BTRON2 window system, and the result of the performance evaluation.

9.1 Prototype System for Evaluation

9.1.1 Target Hardware and Software Environments

The target machine of our prototype implementation of the BTRON2 window system is the PC9340 [83] workstation (Figure 9.1). It includes a GMICRO/100 (Clock: 20MHz, 8.1 MIPS) [129] based on the TRON VLSI CPU specification [116], 16M bytes of main memory, and 100M bytes hard disk. It is equipped with an Ethernet board.

Its kernel is mainly based on the ITRON2 specification [113]. Upon the kernel, a file system based on the ITRON/FILE specification [119] and graphic primitives based on the BTRON2 specification are equipped. As for its network system, we use a light weight real-time network protocol implemented in our laboratory for the window system, called NIEP (Network ITRON Efficient Protocol) [77, 76]. The shared-memory coherent algorithm of WROs is implemented using the reliable datagram primitive of NIEP.

9.1.2 Prototype System on PC 9340

We have implemented a prototype system of the BTRON2 window system on PC9340. Most parts of the system are coded in C, and their code sizes are shown in Table 9.2 and 9.3.

Figure 9.1: Outlook of PC 9340

Machine Type	PC 9340
Organization	Mitsubishi Co.
CPU	Gmicro/100 [116, 129] (Clock: 20M Hz, 8 MIPS)
Main Memory	16 M Bytes
Cache Memory	M32/CCM, 16 K Bytes
Disk	100 M Bytes
Display	800 $\times$ 600
Color	256
Network (Physical Layer)	Ethernet (10 M bps)
Network (Data Link Layer)	NIEP [77, 76]
OS Kernel	ITRON2 [113] ITRON/FILE [119, 54]

Table 9.1: Target environment of the BTRON2 window system

Window Real-Objects

WRO Data Structure locates in a shared memory pool of the ITRON2 kernel. In its R-tree index, the minimum and maximum numbers of entries in a non-root node are 5 and 10. The ones of a root node are 6 and 12.

WRO Manager is implemented with a library. Both application core units and user-agents are linked with the library, and issue window calls by the library function calls. The window call function accesses WRO Data Structure directly when there is no replica of the WRO. Otherwise, it translates the window call into a message packet, then it sends the message to the Distribution Server of its master WRO. Abstract events are implemented using asynchronous message passing facility, which is called *message buffer*, of ITRON2. When an application task initializes its WRO Manager library, a message box for receiving abstract events is created.

The Distribution Server is implemented by the ordinary ITRON task with the highest priority of all tasks related to the BTRON2 window system. It receives window call messages through a rendezvous port from the local machine. From remote machines, it receives through NIEP message primitive.

User-Agents

We have implemented a user-agent with multiple ITRON tasks. *Cyclic Task* is an ITRON task that is periodically waken up by a cyclic handler. It realizes flickering frames of selected objects, and an application task can control the flickering frequency easily by redefining the cyclic handler with the new cycle time.

User-agent scheduler is implemented with a cyclic handler with the 50 msec cycle time. It only issues the `rot_rdq()` system call of ITRON2, which rotates the task ready queue of the specified priority.

Redrawing task (RT) is implemented with an ITRON task, which is generated each time when a WRO is generated. In most case, RTs are in the sleep state. When a user-agent task requests a long-term redraw, it only wakes up its corresponding RT and gives it the region to draw. Then, the RT draws the specified region of a window it manages.

We have implemented *script interpreter task (SIT)* by porting and extending a freeware called xscheme [12].

Module	Size
User-Agent	265 K byte
Window Real-Object	138 K byte
WRO Manager	89 K byte
Distribution Server	49 K byte
Script Interpreter Task	304 K byte

Table 9.2: Execution code size of the BTRON2 window system

Module	Size
User-Agent (except window manager task)	6.9 K lines in C
WRO	7.1 K lines in C
WRO Manager	6.3 K lines in C
Distribution Server	0.8 K lines in C
Script Interpreter Task	14.2 K liens in C 0.35 K liens in xscheme
xscheme Original Code	12 K lines in C
System Call Interface Extension (61 calls)	1.3 K lines in C
WRO Interface Extension (25 calls)	0.9 K lines in C
Definition of Additional Symbols	0.35 K lines in xscheme

Table 9.3: Source code size of the BTRON2 window system

A Sample Graphic Editor

We have built a graphic editor upon the prototype system of the BTRON2 window system. It is implemented as a simple graphic editor. However, from the benefit of architectural advantages of the BTRON2 window system, it can work as a multi-user editor, a simple animation tool, a graphic editor with gesture interfaces, and so on. Its detailed architecture and implementation are described in Section 6.6.

9.2 Performance Evaluation

We have evaluated the performance by measuring two costs: one is the time necessary for a user's update operation in a shared window to be reflected on the user's screen. Another is the time necessary for the picking in a shared window. The former is the response time of actions implemented using the `upd_obj()` calls to WROs, and the latter is the one using only the `rea_obj()` calls.

9.2.1 Response Time to Update Operations

We have measured response time of a resize operation to a rectangle displayed in a shared window of a multi-user graphic editor. It is measured by the passing time since a user-agent issues an update call to a WRO, until the user-agent finishes drawing which reflects the update. The resized rectangle is the size of 30×30 pixel and filled with the 50% gray pattern. The shared window contains other 11 rectangles that have no overlaps with each other. The window is shared by one to five machines. Our measurement has been executed on both master's machine and replica's machine.

Figure 9.2 represents the result of the measurement on the replica's machine. This result shows that the response time is around 30 msec. When more than two machines share a WRO, the response time is independent of the number of the machines. On the contrary, on a master machine, the response time increases with number of sharing machines because of the cost to send window call request messages to replica machines (Figure 9.3).

Even if a large number of machines, for example 30 or 50, share a WRO simultaneously, the response time on replica machines is almost the same. However, the response time on the master machine and the notification time of the operation would increase relatively with the number of sharing machines. While the notification time needs not be short, too long notification time cannot be acceptable in practical use. To solve this problem, we must adopt a new implementation technique. If a high performance multi-cast facility of

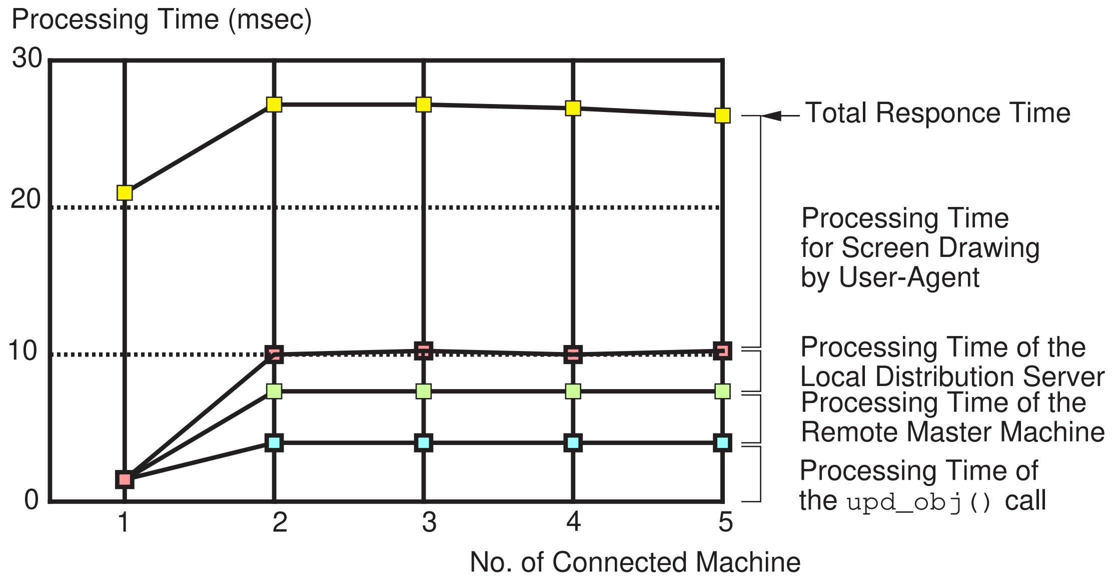


Figure 9.2: Response time to resizing action to a rectangular segment (on a replica machine)

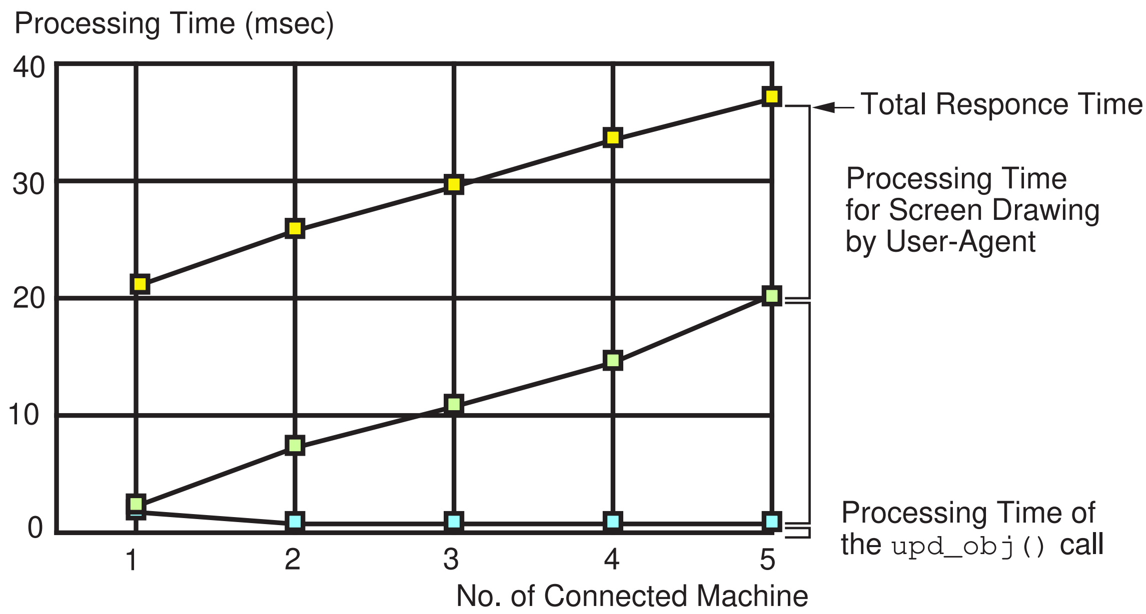


Figure 9.3: Response time to resizing action to a rectangular segment (on a master machine)

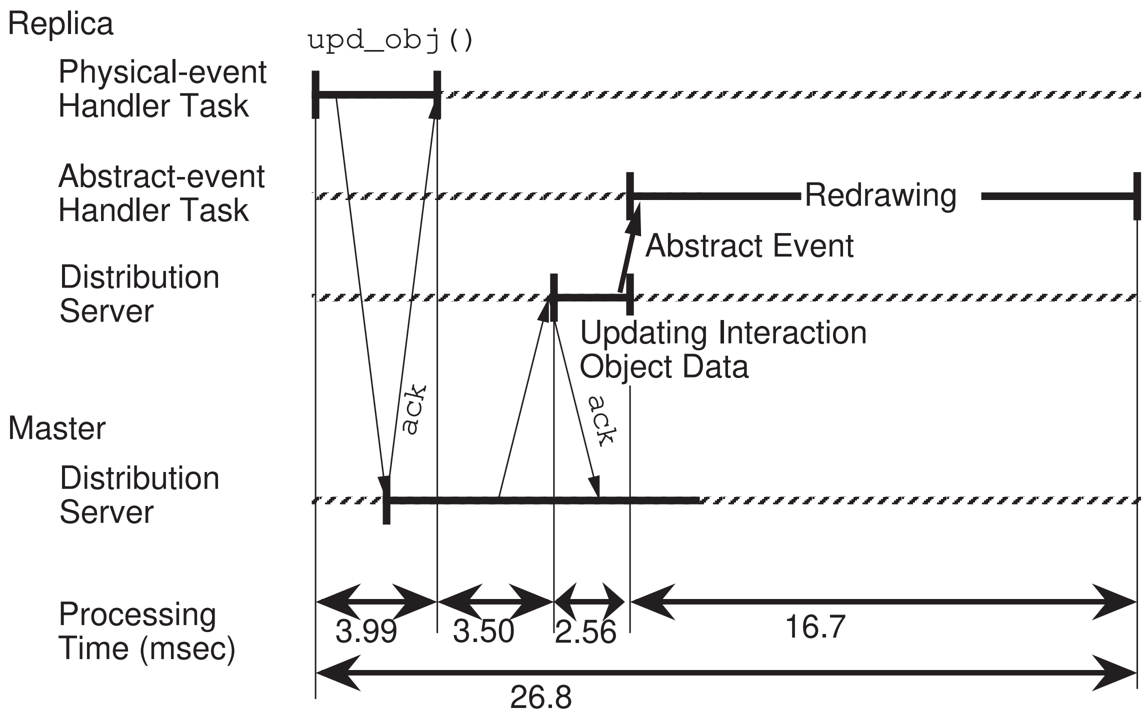


Figure 9.4: Cost analysis of the update operation processing

network system is available, we can solve most case of this problem.

Next, we compare our result with others. As for Teidan, [103] has reported the similar multi-user interface cost. Authors have measured only pure multi-user interfacing costs, and obtained the result that is about 30 msec. Upon our BTRON2 window system, we can calculate these costs easily by subtracting the response time for one machine from each response time. The cost of our system is about 5 to 6 msec. This shows that our result is five or six times better than the one of Teidan. This result is caused by several reasons. The first reason is the effectiveness of our implementation techniques such as R-Tree and full-replicated coherence. Moreover, our system keeps the consistency among multiple window views in a less abstract layer than Teidan. Thus, our system processes the feedback in the lower layer.

We have analyzed the result shown by Figure 9.2 more precisely. As shown in Figure 4.9, the response processing consists of a sequence of several kinds of processings. Figure 9.4 shows how much of the response time each processing accounts for.

As Figure 9.4 shows, when two machines share the same WRO, the total response time on the replica's machine is just 26.8 msec (average). Average processing time of the `upd_obj()` call is just 3.99 msec. Here, notice that the `upd_obj()` call only sends an

Sample Data	R-tree Num. of Objects	Upon Objects (Min - Max)	Out of Objects	Simple List (Head - Tail)
Sample Panel	6	0.58 - 0.60	0.18	0.030 - 0.135
Sample Rectangles	15	0.75 - 0.95	0.18	0.030 - 0.343
Figure of the TRON Project	118	0.99 - 1.38	0.99	0.085 - 6.706
Map of Japan	277	0.79 - 1.59	0.31	0.110 - 53.606

Table 9.4: Response time to a picking action (msec)

operation request message to the distribution server on a master machine, and immediately returns. Then, the average passing time from the return until local distribution server is invoked is 3.50 msec. During this, the master's distribution server processes the request message, and forwards it to the distribution server of the source machine. The time required for updating local WRO Data Structure by the distribution server is 2.56 msec. The rest 16.7 msec out of the 26.8 msec is spent for re-drawing by user-agent. From this result, we know that the cost of the coherence mechanism of WROs is small compared with other GUI processing costs, such as re-drawing.

9.2.2 Response Time to Picking Operations

Another measurement is the response time to a picking operation, which searches an interaction object just under the cursor position. It can indicate the effectiveness of the R-tree index because the picking operation is processed locally using only the read window calls and independent of the network delay of coherent algorithm.

We have measured the response time to picking operation applied to several sample data sets. To clarify the effect of R-tree, we have measured it using a special WRO implemented only with simple list structure. The result is expressed in Table 9.4. In the Table, the "out of objects" means queries of points that are not contained even by the root node of R-trees.

This result shows two benefits of the R-tree structure over the simple list structure. First, the efficiency of the R-tree structure is apparent when the number of objects is large. This is because it decreases the number of comparing whether a queried point is contained by each object. Second, the difference between minimum and maximum response time of the R-tree is very small because it is a height-balanced tree.

9.3 Summary

This chapter has described the detailed information on the prototype system of the BTRON2 window system. We fully utilize functionality of the ITRON kernel to increase its performance and to make the implementation simple. However, the BTRON2 window system can be implemented upon other conventional multitask operating systems.

Moreover, this chapter has reported the results of its performance evaluations. From the results, we can confirm the following issues.

1. R-tree is effective both to shorten and to stabilize the response time, especially when a WRO contains a large number of interaction objects.

2. The coherence algorithm of WROs works in practically sufficient performance. This results in the good responsiveness for the update operation to a window.

Chapter 10

Conclusion

10.1 Summary and Conclusion

Recent great efforts of many user interface researchers have brought us a lot of advanced human-computer interaction (HCI) techniques. However, most computer users do not benefit from the results. This is because implementations of these techniques require complicated mechanisms that are not supported by conventional user interface platforms. Thus, we have started this research with the aim of furnishing a new user interface platform that works as a good implementation environment of the new techniques. In this research, we have proposed a new GUI software architecture and a window system, called BTRON2 window system, based on the architecture, which contains novel features as follows:

1. It exports interaction objects so that external GUI applications even on different machines can share them.

   Our architecture puts segmented data of interaction objects in distributed shared memories, which are abstractions of GUI windows and called Window Real-Objects (WROs). WROs contain only static data of interaction objects, and each machine locally contains their methods. GUI applications can communicate with each other by sharing interaction objects in the common WRO.

2. GUI software consists of multiple cooperative execution units sharing the common WRO.

   This results from the high modularity of parallel programming in the shared memory paradigm.

3. Data and methods of interaction objects are separated.

This approach is the opposite to the object-oriented approach, in which data and methods are always encapsulated in one unit. Our architecture separates the data and the operations into different components so that users can access the same interaction object data with different methods at the same time.

This feature is based on our basic insight into user interfaces that user interfaces should adapt dynamically according to the characteristics of the user, the environment, target machine, and so on. The separation of data and method of interaction objects is the fundamental mechanism of this adaptable user interface.

4. For a window system, our architecture provides rather highly abstract API, but requires no high abstraction of programming language.

   In our architecture, GUI application programmers need use only function calls (window calls) and asynchronous messages (abstract events). Mostly, these abstractions can be handled by any programming languages.

The main contribution of this dissertation is to propose a new user interface software architecture that facilitates GUI applications to cooperate with other unknown GUI programs. This dissertation has shown this mechanism can solve integratedly the difficulty in implementing some new interaction techniques, such as application cooperation, script processing, multi-user interface, and application-transparent supports for users with special needs.

The focus of our discussion is which layer of the GUI software architecture should handle the inter application communication. Our strategy is that the interaction object component should handle the communication among GUI applications.

Our architecture realizes the above new HCI techniques by communicating with other programs in the interaction object component. Through discussions in this dissertation, we have shown the flexibility of the interaction object sharing scheme. Applying the shared interaction object scheme to new HCI techniques, we have obtained several novel benefits to application programmers for each technique as follows:

1. Application programming aspect:

   The capability to decompose a GUI application into multiple cooperative units allows end users to combine different application programs from different sources, which behave in the same window as if they are organized as one application program.

   On the other hand, each cooperative decomposed unit can run in parallel and on a suitable machine for its execution. In our architecture, several classes of parallel programming are available. We have verified popular three classes of parallel programming: result parallelism, specialist parallelism, and agenda parallelism.

Also, application developers can program each decomposed unit in the suitable programming language. This allows the developers to implement GUI applications using multiple programming languages.

2. Multi-user interface aspect:

   The BTRON2 window system realizes multi-user interfaces by exporting interaction objects to other user-agents. This shared interaction object scheme imposes no special framework on the application core component and semantic data of groupware. On the other hand, the shared interaction object scheme implements the shared view in a lower layer than most groupware toolkits. In spite of this, our architecture can provide shared views as flexible as the groupware toolkits.

3. Scripting aspect:

   Upon the BTRON2 window system, application programmers need not embed code for both user interfaces and scripting interfaces in an interactive GUI application, unlike the tool integration toolkits nor the application cooperation toolkits. Moreover, unlike automatic demonstration system, scripting systems, and event macro systems with the event simulation scheme, application programmers can describe script programs in high abstraction.

4. Assistive technology aspect:

   The shared interaction object scheme enables to build screen reading software or braille output software independent of each application program.

Furthermore, to make our architecture practical, this dissertation has proposed several implementation techniques such as R-tree, full-replication coherence algorithm, and multi-tasking with real-time scheduling,

We have verified the effectiveness of the techniques by performance evaluations on our prototype system of the architecture. The result shows that the response time for a user's resizing operation to an interaction object is about 50 msec, and that the response time is independent of the number of sharing users. The response time for a user's picking operation is less than 1 msec when the number of interaction objects on a window is less than 100.

10.2 Research Direction for the Future

The field of the user interface software research has not been exploited fully. Conventionally, most user interface researchers have devoted their efforts into developing new interaction techniques. Efficient implementation of these techniques is rather out of their interest. In implementing the BTRON2 window system, we have encountered many problems about implementation mechanisms. We have solved many of the problems, and, however, some problems remain unsolved. For example, efficient spatial locking algorithm, the version problem, techniques for concurrent access to R-tree, and partial replication of R-tree are among them. In our prototype system in the current version, these problems are solved by ad hoc mechanisms or unsolved yet. We are planning to apply more theoretical methodologies to these problems.

Furthermore, we are planning to port the BTRON2 window system on different computing environments. For example, our current prototype system does not use hardware- or OS-supported distributed shared memories. This is because we want to show the applicability of our idea to most popular computing environments. When such distributed shared memories are available, more efficient implementation mechanisms would be obtained upon the facilities. Thus, the efficient implementation on such hardware- or OS-supported distributed shared memories is our important future work.

Appendix A

Summary of the C Interface of WROs

The following is extracted from the draft manual titled "C Interface to BTRON Graphical User Interface Shell (version 0.1)" [62].

A.1 Functions to Start and to Quit

- `W ini_sys(ID mbfid, MID mymid);`

  The `ini_sys` call initializes the library of the WRO programming interface.

- `W qui_sys();`

  The `qui_sys` call quits from the library of the WRO programming interface.

A.2 Functions to Operate WROs

- `W set_wro(MID mid, UW width, UW height, WindowHint *hint);`

  The `set_wro` call creates and sets a WRO.

- `W del_wro(W wroid);`

  The `del_wro` call deletes a WRO.

- `W map_wro(W master, REPLICAS replica);`

  The `map_wro` call maps a WRO to another WRO to make a shared window.

- `W rea_wro(W wroid, WROE *wroe);`

  The `rea_wro` call reads parametarized data from a WRO.

- `W upd_wro(W wroid, WROEMASK mask, WROE *wroe);`

  The `upd_wro` call updates parameter values of a WRO.

A.3 Functions to Operate Abstract Events

- `W gme_aev(UW wroid);`

  The `gme_aev` call requests a WRO to send abstract events to the caller process.

- `W dgm_aev(UW wroid);`

  The `dgm_aev` call requests a WRO to stop sending abstract events to the caller process.

- `W get_aev(AEV *aev, TMO tmout);`

  The `get_aev` call gets an abstract event from an event queue.

A.4 Functions to Operate Context of WROs

A.4.1 Functions to Operate Patterns

- `W set_pat(UW wroid, PATSEG *patseg, UH size, UH hint);`

  The `set_pat` call sets a pattern segment in a WRO.

- `W del_pat(UW wroid, UW id, UH hint);`

  The `del_pat` call deletes a pattern segment from a WRO.

- `W rea_pat(UW wroid, UW id, PATSEG *p2patseg, UH hint);`

  The `rea_pat` call retrieves a pattern segment from a WRO.

A.4.2 Functions to Operate Mask Patterns

- `W set_msk(UW wroid, MASKSEG *maskseg, UH size, UH hint);`

  The `set_msk` call sets a mask pattern segment in a WRO.

- `W del_msk(UW wroid, UW id, UH hint);`

  The `del_msk` call deletes a mask pattern segment from a WRO.

- `W rea_msk(UW wroid, UW id, MASKSEG *maskseg, UH hint);`

  The `rea_msk` call retrieves a mask pattern segment from a WRO.

A.4.3 Functions to Operate Line Attributes

- `W set_lin(UW wroid, LATRSEG *latrseg, UH size, UH hint);`

  The `set_lin` call sets a line attributes segment in a WRO.

- `W del_lin(UW wroid, UW id, UH hint);`

  The `del_lin` call deletes a line attributes segment from a WRO.

- `W rea_lin(UW wroid, UW id, LATRSEG *latrseg, UH hint);`

  The `rea_lin` call retrieves a line attributes segment from a WRO.

A.4.4 Functions to Operate Color Maps

- `W set_col(UW wroid, COLMAPSEG *colmapseg, UH size, UH hint);`

  The `set_col` call sets a color map entry segment in a WRO.

- `W del_col(UW wroid, UW id, UH hint);`

  The `del_col` call deletes a color map entry segment from a WRO.

- `W rea_col(UW wroid, UW id, COLMAPSEG *colmapseg, UH hint);`

  The `rea_col` call retrieves a color map entry segment from a WRO.

A.5 Window Calls

- ```
  W set_obj(UW wroid, UW search_mode, UW type,
            UB *obj_body, UW size, UH hint);
  ```

  The `set_obj` call sets an interaction object to a WRO.

- ```
  W rea_obj(UW wroid, UW search_mode, SKEY *key,
             UB *obj_body, UW *r_size, UH hint);
  ```

  The `rea_obj` call retrieves interaction objects from a WRO.

- ```
  W del_obj(UW wroid, UW search_mode, SKEY *key,
            UH hint);
  ```

  The `del_obj` call deletes interaction objects from a WRO.

- ```
  W upd_obj(UW wroid, UW search_mode, SKEY *key,
            TGSF field, UB *newvalue, UH hint);
  ```

  The `upd_obj` call updates parameter values of interaction objects in a WRO.

- ```
  W ope_obj(UW wroid, UW search_mode, SKEY *key,
            UW operation, UW info, UH hint);
  ```

  The `ope_obj` call operates interaction objects in a WRO.

- ```
  W loc_obj(UW wroid, UW search_mode, SKEY *key,
            UW wocallclass, UH hint);
  ```

  The `loc_obj` call locks interaction objects in a WRO.

- ```
  W unl_obj(UW wroid, UW search_mode, SKEY *key,
            UH hint);
  ```

  The `unl_obj` call unlocks interaction objects in a WRO.
  ```

Appendix B

Summary of the SCHEME Interface of WROs

The following is extracted from the draft manual titled “XSCHEME Interface to BTRON Graphical User Interface Shell (version 0.2)” [63].

B.1 Functions to Start and to Quit

- ```
  (INI_SYS <mbfid> <mid>)
  ```

  The (INI_SYS...) call initializes the WRO programming interface.

- ```
  (QUI_SYS)
  ```

  The (QUI_SYS...) call quits from the WRO programming interface.

B.2 Functions to Operate WROs

- ```
  (SET_WRO <mid> <width> <height> <attribute>
           <left> <top> <right> <bottom> <pattern>
           <title width>)
  ```

  The (SET_WRO...) call creates and sets a WRO.

- ```
  (DEL_WRO <mid> <id>)
  ```

  The (DEL_WRO...) call deletes a WRO.

- ```
  (MAP_WRO <id> <mid> <id>)
  ```

  The (MAP_WRO...) call maps a WRO to another WRO to make a shared window.
  ```

- `(REA_WRO <wroid>)`

  The `(REA_WRO...)` call reads parametarized data from a WRO.

- `(UPD_WRO <wroid>, <mask>, <wroenv>)`

  The `(UPD_WRO...)` call updates parameter values of a WRO.

B.3 Functions to Operate Abstract Events

- `(GME_AEV <id>)`

  The `(GME_AEV...)` call requests a WRO to send abstract events to the caller process.

- `(DGM_AEV <id>)`

  The `(DGM_AEV...)` call requests a WRO to stop sending abstract events to the caller process.

- `(GET_AEV <timeout>)`

  The `(GET_AEV...)` call gets an abstract event from an event queue.

B.4 Functions to Operate Context of WROs

B.4.1 Functions to Operate Patterns

- ```
  (SET_PAT <id> <atr> <patid> <hsize> <vsize>
           <fgcol> <bgcol> <maskid>)
  ```

  The `(SET_PAT...)` call sets a pattern segment in a WRO.

- `(DEL_PAT <wroid> <patid> <hint>)`

  The `(DEL_PAT...)` call deletes a pattern segment from a WRO.

- `(REA_PAT <wroid> <patid> <hint>)`

  The `(REA_PAT...)` call retrieves a pattern segment from a WRO.

B.4.2 Functions to Operate Mask Patterns

- ```
  (SET_MASK <id> <atr> <maskid> <hsize> <vsize>
            <len> "mask")
  ```

  The `(SET_MSK...)` call sets a mask pattern segment in a WRO.

- (DEL_MSK <wroid> <mskid> <hint>)

  The (DEL_MSK...) call deletes a mask pattern segment from a WRO.

- (REA_MSK <wroid> <mskid> <hint>)

  The (REA_MSK...) call retrieves a mask pattern segment from a WRO.

B.4.3 Functions to Operate Line Attributes

- (SET_LIN <wroid> <latrseg> <size>, <hint>)

  The (SET_LIN...) call sets a line attributes segment in a WRO.

- (DEL_LIN <wroid> <latrid> <hint>)

  The (DEL_LIN...) call deletes a line attributes segment from a WRO.

- (REA_LIN <wroid> <latrid> <hint>)

  The (REA_LIN...) call retrieves a line attributes segment from a WRO.

B.4.4 Functions to Operate Color Maps

- (SET_COL <wroid> <colmapseg> <size> <hint>)

  The (SET_COL...) call sets a color map entry segment in a WRO.

- (DEL_COL <wroid> <colid> <hint>)

  The (DEL_COL...) call deletes a color map entry segment from a WRO.

- (REA_COL <wroid> <colid> <hint>)

  The (REA_COL...) call retrieves a color map entry segment from a WRO.

B.5 Window Calls

- (SET_OBJ <id> <mode> <type> <dataBody>)

  The (SET_OBJ...) call sets an interaction object to a WRO.

- (REA_OBJ <id> <mode> <skey>)

  The (REA_OBJ...) call retrieves interaction objects from a WRO.

- (DEL_OBJ <id> <mode> <skey>)

  The (DEL_OBJ...) call deletes interaction objects from a WRO.

- `(UPD_OBJ <id> <mode> <skey> <field> <newvalues...>)`

  The `(UPD_OBJ...)` call updates parameter values of interaction objects in a WRO.

- `(OPE_OBJ <id> <mode> <skey> <ope> <info>)`

  The `(OPE_OBJ...)` call operates interaction objects in a WRO.

- `(LOC_OBJ <id> <mode> <skey>)`

  The `(LOC_OBJ...)` call locks interaction objects in a WRO.

- `(UNL_OBJ <id> <mode> <skey>)`

  The `(UNL_OBJ...)` call unlocks interaction objects in a WRO.

Appendix C

Script Program for Algorithm Animation

The following scheme source code is the sample script of the BTRON2 window system which implements algorithm animation of “Tower of Hanoi”.

```
;;
;;  Sample Program if BGSHELL/XSCHEME
;;  --- Tower of Hanoi Algorithm Animation -------
;;
;;  Copyright (C) 1993 by Koshizuka, Noboru
;;
;;        Sakamura Lab., Department of Information Science,
;;        Faculty of Science, Univ of Tokyo
;;        7-3-1, Hongo, Bunkyo-Ku, 113 Tokyo, Japan
;;        (TEL)      03-3812-2111 (ex) 4096
;;        (Internet) koshi@is.s.u-tokyo.ac.jp
;;

(set! _DISKHEIGHT 10)

(set! _black 50)
(set! _gray  51)
(set! _white 52)


(define (make_std_rect l t r b)
  (list 'g_store #x0001 _black _gray 0 l t r b))

(define (init-hanoi mbfid mid)
  (BEGIN

   ;; creating hanoi data structure
   (make-hanoi-data 3)

   ;; initialize BGSHELL
   (ini_sys mbfid mid)

   ;; Creating a WRO --> "wroid"
   (set! wroid   (rset_wro mid 800 600 0 100 100 300 300 1 3))
   (set! wroid2  (rset_wro mid 800 600 0 300 100 500 300 1 3))
   (map_wro wroid mid wroid2 0)

   ;; Creating standard patterns
   (set_pat wroid 0 _black 2 2 #x0fffffff #x80000000 7)
   (set_pat wroid 0 _gray  2 2 #x0fffffff #x80000000 3)
   (set_pat wroid 0 _white 2 2 #x0fffffff #x80000000 1)
```

```
    ;; Creating Line Segments representing towers
    (set_line wroid 0 GSOBJ (list g_store #x2 _black 50 120 50 20))
    (set_line wroid 0 GSOBJ (list g_store #x2 _black 100 120 100 20))
    (set_line wroid 0 GSOBJ (list g_store #x2 _black 150 120 150 20))

    ;; Creating Rectangle Segments representing Disks
    (put-disk (set_rect wroid 0 GSOBJ (make_std_rect 20 90 80 100)) 0)
    (put-disk (set_rect wroid 0 GSOBJ (make_std_rect 25 80 75 90))  0)
    (put-disk (set_rect wroid 0 GSOBJ (make_std_rect 30 70 70 80))  0)
    (put-disk (set_rect wroid 0 GSOBJ (make_std_rect 35 60 65 70))  0)
    (put-disk (set_rect wroid 0 GSOBJ (make_std_rect 40 50 60 60))  0)))

;; ---------------------------------------------
;; Number of tower's of hanoi is 3
;; ---------------------------------------------
(set! _NUMOFTOWERS 3)

;; =============================================
;; creating hanoi data
;; --- creating hanoi's towers with N-disks ---
;; =============================================
;;
;; hanoi data structure
;; +--------+--------------+---------------+--------+----------------+
;; |        |              |       0       |   1    |       2        |
;; +--------+--------------+---------------+--------+----------------+
;; |Tower 0 | hanoi_data[0]|Number of Disks|Disklist| x-axis of tower|
;; |Tower 1 | hanoi_data[1]|Number of Disks|Disklist| x-axis of tower|
;; |Tower 2 | hanoi_data[2]|Number of Disks|Disklist| x-axis of tower|
;; |   :
;;

(define (make-hanoi-data N)
  (begin
   (set! hanoi_data (make-array N _NUMOFTOWERS))
   (init-hanoi-data (- _NUMOFTOWERS 1))))

;; =============================================
;; initialize hanoi data of 0--n th towers
;; (recursive algorithm)
;; =============================================
(define (init-hanoi-data n)
  (cond ((>= n 0)
         (array-set! hanoi_data n 1 '())
         (set-disknum n 0)
         (set-position n (+ (* n 50) 50))
         (init-hanoi-data (- n 1)))
        (T NIL)))

;; =============================================
;; operations to hanoi data
;; =============================================

;; -------------------------------------
;; get the top disk of the tower "from"
;; -------------------------------------
(define (get-disk from)
  (cond ((> (get-disknum from) 0)
         (let ((disks (array-ref hanoi_data from 1)))
           (array-set! hanoi_data from 1 (cdr disks))
           (set-disknum from (- (get-disknum from) 1))
           (car disks)))
        (T NIL)))

;; -------------------------------------
```

```
;; get a disk on the top of the tower "to"
;; ------------------------------------
(define (put-disk disk to)
  (begin
   (set-disknum to (+ (get-disknum to ) 1))
   (array-set! hanoi_data to 1
              (append (list disk) (array-ref hanoi_data to 1)))))

;; ------------------------------------
;; get/set the number of disks on the tower "at"
;; ------------------------------------
(define (get-disknum at)
  (array-ref hanoi_data at 0))

(define (set-disknum at num)
  (array-set! hanoi_data at 0 num))

;; ------------------------------------
;; get/set the position of x-axis of the tower "at"
;; ------------------------------------
(define (get-position at)
  (array-ref hanoi_data at 2))

(define (set-position at pos)
  (array-set! hanoi_data at 2 pos))


;; ---------------------------------------------------------------------------
;; Main part of Hanoi's Tower Animation Program
;; ---------------------------------------------------------------------------
(set! _STEP 5)
(set! _STEPWAITTIME 25)

(define (hanoi)
  (transfer 0 1 2 5))

;;-----------------------------
;; Main hanoi's tower Routine
;;-----------------------------
;; Transfer "n" disks from the "from"-th tower
;;                     to the  "to"-th tower
;;                     using the "via"-th tower
;;-----------------------------
(define (transfer from to via n)
  (cond((= n 1)
        (movedisk from to))
       (T
        (transfer from via to (- n 1))
        (movedisk from to)
        (transfer via to from (- n 1)))))

;;
;; Move a Disk from the "from"-th tower to the "to"-th tower
;;
(define (movedisk from to)
  (let* ((upheight   (- (* _DISKHEIGHT (get-disknum from)) 100))
         (downheight (- 90 (* _DISKHEIGHT (get-disknum to))))
         (movewidth  (- (get-position to) (get-position from)))
         (moveddisk  (get-disk from)))
    (put-disk moveddisk to)
    (move-step moveddisk 0 upheight   _STEP)
    (move-step moveddisk movewidth 0  _STEP)
    (move-step moveddisk 0 downheight _STEP)))

;;
;; Move a tgs object (whose id is "disk") is moved as much as ("dx", "dy")
;; in "step"-times
;;
```

```
(define (move-step disk dx dy step)
  (cond ((> step 0)
         (move1 disk (quotient dx step) (quotient dy step))
         (wai_tsk _STEPWAITTIME)
         (move-step disk
                    (- dx (quotient dx step))
                    (- dy (quotient dy step)) (- step 1)))
        (T NIL)))

;;
;; Move a tgs object (whose id is "disk") is moved as much as ("dx", "dy")
;;
(define (move1 disk dx dy)
  (let* ((d        (rea_obj wroid ID_SEARCH (list disk)))
         (databody (cadr d))
         (wotype   (car databody))
         (subtype  (logand wotype #xff00)))
    (cond ((or (= subtype RECT_ID)
               (= subtype RRECT_ID)
               (= subtype CIRCLE_ID))
           (let ((left   (list-ref databody 6))
                 (top    (list-ref databody 7))
                 (right  (list-ref databody 8))
                 (bottom (list-ref databody 9)))
             (rupd_tgs wroid ID_SEARCH (list disk) FRAME
                   (+ left dx) (+ top dy) (+ right dx) (+ bottom dy))))
          ((= subtype LINE_ID)
           (let ((left   (min (list-ref databody 4) (list-ref databody 6)))
                 (top    (min (list-ref databody 5) (list-ref databody 7)))
                 (right  (max (list-ref databody 4) (list-ref databody 6)))
                 (bottom (max (list-ref databody 5) (list-ref databody 7))))
             (rupd_tgs wroid ID_SEARCH (list disk) FRAME
                   (+ left dx) (+ top dy) (+ right dx) (+ bottom dy)))))))

;;=========================================================================
;;  Copyright (C) 1993 by Koshizuka, Noboru
;;
;;        Sakamura Lab., Department of Information Science,
;;        Faculty of Science, Univ of Tokyo
;;        7-3-1, Hongo, Bunkyo-Ku, 113 Tokyo, Japan
;;        (TEL)      03-3812-2111 (ex) 4096
;;        (Internet) koshi@is.s.u-tokyo.ac.jp
;;=========================================================================
```

Bibliography

[1] T. Abe, K. Maeno, S. Sakata, and H. Fukuoka. Distributed Cooperative Control for Sharing Applications Based on MERMAID (Multimedia and Multiparty Desktop Conferencing System). *Journal of Informantion Processing Society of Japan*, Vol. 34, No. 6 (Jun. 1993), pp. 1406–1416. in Japanese.

[2] T. Ajisaka. Open CASE Platforms. *Computer Software*, Vol. 10, No. 2, (Mar., 1993), pp. 100-108, in Japanese.

[3] S. Ahuja, and N. Carriero. Linda and Friends. *IEEE Computer*, Vol. 19, No. 8 (Aug. 1986), pp. 26–34.

[4] M. P. Altenhofen. *Using and Porting shX – for version 1.0 –*. Digital Equipment Corporation, Nov. 1990.

[5] APPLE COMPUTER INC. *Inside Macintosh*. Addison-Wesley, 1986.

[6] APPLE COMPUTER INC. Apple Event Manager. In *Inside Macintosh, Volume VI*. Addison-Wesley, 1991.

[7] APPLE COMPUTER INC. *Inside Macintosh, Volume VI*. Addison-Wesley, 1991.

[8] L. Bass, and J. Coutaz. *Developing Software for the User Interface*. Addison-Wesley, 1991.

[9] L. Bass, E. Hardy, K. Hoyt, M. R. Little, Jr., and R. Seacord. Introduction to the Serpent User Interface Management System. Technical Report CMU/SEI/88-TR-5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, Mar. 1988.

[10] L. Bass, E. Hardy, K. Hoyt, M. R. Little, Jr, and R. Seacord. The Serpent Runtime Architecture and Dialogue Model. Technical Report CMU/SEI/88-TR-6, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, May 1988.

[11] R. J. Beach, J. C. Beatty, K. S. Booth, D. A. Plebon, and E. L. Fiume. The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program. *ACM Transactions on Graphics*, Vol. 16, No. 3 (Jul. 1982), pp. 277–287.

[12] D. M. Betz. *XSCHEME: An Object-oriented Scheme – Version 0.28 –*, 1991.

[13] R. Bisiani, and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Transactions on Computers*, Vol. 37, No. 8 (Aug. 1988), pp. 930–945.

[14] D. Borrow, S. Mittal, and M. J. Stefik. Expert Systems: Perils and Promise. *Communications of the ACM*, Vol. 29, No. 9 (Sep. 1986).

[15] N. Carriero, and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Survey*, Vol. 21, No. 3 (Sep. 1989), pp. 261–322.

[16] N. Carriero, D. Gelernter, and J. Leichter. Distributed Data Structures in Linda. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages* (1986), pp. 236–242.

[17] M. J. Cecil. *XTV: A User's Guide – Revision 2.1 –*, Aug. 1991.

[18] D. R. Cheriton. Problem-Oriented Shared Memory: A Decentralized Approach to Distributed Systems. *ACM Operating System Reviews*, Vol. 19, No. 4 (Oct. 1985), pp. 26–33.

[19] D. R. Cheriton. Problem-Oriented Shared Memory: A Decentralized Approach to Distributed System Design. In *Proceedings of the Sixth International Conference on Distributed Computing Systems* (May 1986), pp. 190–197.

[20] D. Clément. A Distributed Architecture for Programming Environments. In *Proceedings of the Fouorth ACM SIGSOFT Symposium on Software Development Environments* (Dec. 1990), pp. 11–21.

[21] D. D. Cowan, T. M. Stepien, R. Ierusalimschy, and C. J. P. Lucena. Application Integration: Constructing Composite Applications from Interactive Components. *Software—Practice and Experience*, Vol. 23, No. 3 (Mar. 1993), pp. 255–275.

[22] P. Dewan. A Tour of the Suite User Interface Software Practice and Experience. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (1990), pp. 57–65.

[23] P. Dewan, and R. Choudhary. Flexible User Interface Coupling in a Collaborative System. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems* (1991), pp. 41–48.

[24] P. Dewan, and R. Choudhary. Primitives for Programming Multi-User Interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Nov. 1991).

[25] P. Dewan, and R. Choudhary. A High-Level and Flexible Framework for Implementing Multiuser User Interfaces. *ACM Transactions on Information Systems*, Vol. 10, No. 4 (Oct. 1992), pp. 345–380.

[26] D. Duis, and J. Johnson. Improving User-Interface Responsiveness Despite Performance Limitations. In *Thirty-Fifth IEEE Computer Society International Conference* (1990), pp. 380–386.

[27] P. K. Egbert. UIPEX: Design of the Application Programmer Interface. Master's thesis, The Graduate College of the University of Illinois at Urbana-Champaign, 1990.

[28] C. A. Ellis, and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD '89 Conference on the Management of Data* (May 1989), pp. 399–407.

[29] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, Vol. 34, No. 1 (Jan. 1991), pp. 38–58.

[30] A. Forin, R. Bisiani, and F. Correrini. Parallel Processing with Agora. Tech. Rep. CMU-CS-87-183, Computer Science Department, Carnegie Mellon University, Dec. 1987.

[31] B. Fromme. Why You Need BMS. *SUN Expert Magazine*, Vol. 4, No. 3 (Mar. 1993), pp. 61–67.

[32] D. Garlan, and E. Ilias. Low-cost, Adaptable Tool Integration Policies for Integrated Environments. In *Proceedings of the Fouorth ACM SIGSOFT Symposium on Software Development Environments* (Dec. 1990), pp. 1–10.

[33] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1 (Jan. 1985), pp. 80–112.

[34] S. J. Gibbs. LIZA: An Extensible Groupware Toolkit. In *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems* (May 1989).

[35] R. Gold, B. Buxton, S. Feiner, C. Schmandt, p. Wellner, and M. Weiser. Ubiquitous Computing and Augmented Reality (Panel Discussion). In *Proceedings of ACM SIGGRAPH '93 Conference* (Aug. 1993), pp. 393–394.

[36] A. Goldberg. *Smalltalk-80 : The Interactive Programming Environment* Addison-Wesley, (1985).

[37] W. D. Gray, B. E. John, and M. E. Atwood. The Precis of Project Ernestine, or, An Overview of a Validation of GOMS. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 307–312.

[38] M. Green. The University of Alberta User Interface Management System. In *Proceedings of the ACM SIGGRAPH Symposium '85* (1985), pp. 205–213.

[39] A. Guttman. R-Trees: A Dynamic Index Structure for Spacial Searching. In *Proceedings of the ACM SIGMOD Conference* (Jun. 1984), pp. 47–57.

[40] M. Hagiya. Graphical User Interface and Environments for Its Development. *bit*, Vol. 21, No. 6 (1989), pp. 51–62. in Japanese.

[41] H. R. Hartson, and D. Hix. Human-Computer Interface Development: Concepts and Systems for Its Management. *ACM Computing Surveys*, Vol. 21, No. 1 (1989), pp. 5–92.

[42] T. R. Henry, S. E. Hudson, and G. L. Newell. Integrating Gesture and Snapping into a User Interface Toolkit Interaction Techniques. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (1990), pp. 112–122.

[43] R. D. Hill. Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction – The Sassfras UIMS. *ACM Transactions on Graphics*, Vol. 5, No. 3 (Jul. 1986), pp. 179–210.

[44] D. Hix. Generations of User-Interface Management Systems. *IEEE Software*, Vol. 7, No. 5 (Sep. 1990), pp. 77–87.

[45] S. E. Hudson. UIMS Support for Direct Manipulation Interfaces. *ACM Computer Grahics*, Vol. 21, No. 2 (Apr. 1987), pp. 120–124. Proceedings of the ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[46] S. E. Hudson, and R. King. A Generator of Direct Manipulation Office Systems. *ACM Transactions of Office Information Systems*, Vol. 4, No. 2 (Apr. 1986), pp. 132–163.

[47] S. E. Hudson, and R. King. Semantic Feedback in the Higgens UIMS. *IEEE Transactions of Software Engineering*, Vol. 14, No. 8 (Aug. 1988), pp. 1188–1206.

[48] H. Ishii. Trends of Groupware Technology. *Journal of Information Processing Society of Japan*, Vol. 30, No. 12 (Dec. 1989), pp. 1502–1508.

[49] H. Ishii. Computer-Supported Cooperative Work. *Computer Software*, Vol. 8, No. 2 (Mar. 1991), pp. 110–122. in Japanese.

[50] W. Jones, P. Williams, G. Robertson, V. Joloboff, and M. Conner. In Search of the Ideal Operating System for User Interface. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (1990), pp. 31–35.

[51] A. Julienne, and L. Russell. Why You Need TOOLTALK. *SUN Expert Magazine*, Vol. 4, No. 3 (Mar. 1993), pp. 51–58.

[52] KABA. *An Introduction to Wnn + GMW*. Iwanami-shoten, 1990. in Japanese.

[53] K. Kawamura. Speech Sound User Interface Toolkit, Feb. 1993. Bachelor's Thesis of the Department of Information Science, Faculty of Science, the University of Tokyo.

[54] Y. Kisuki, H. Tsubota, and K. Saitoh. MR3200F: File System Based on the ITRON/FILE Specification. *Proceedings of TRON Technical Workshop*, Vol. 2, No. 3 (Oct. 1989), pp. 43–51. in Japanese.

[55] M. J. Knister, and A. Prakash. DistEdit: A Distributes Toolkit for Supporting Multiple Group Editors. In *Proceedings of the ACM CSCW '90* (Oct. 1990).

[56] R. Kondo, N. Koshizuka, and K. Sakamura. An Extension of BTRON HMI Specification for Multi-user Applications. *Proceedings of TRON Technical Workshop*, Vol. 5, No. 2 (Mar. 1993), pp. 31–37. in Japanese.

[57] N. Koshizuka, H. Takada, M. Saito, Y. Saito, and K. Sakamura. Implementation Issues of the TACL/TULS Language System on BTRON. In *TRON Project 1989* (1989), Springer-Verlag, pp. 113–132.

[58] N. Koshizuka, H. Takada, and K. Sakamura. Window Shared Data Pool. *IPSJ SIG Notes on Operating System*, 52–2 (Sep. 1991), IPS Japan. in Japanese.

[59] N. Koshizuka, H. Takada, and K. Sakamura. Window Shared Data Pool: A New Window System for BTRON2. In *Proceedings of the Eighth TRON Project Symposium* (Nov. 1991), IEEE Computer Society Press, pp. 151–168.

[60] N. Koshizuka, and K. Sakamura. An Overview of the BTRON Enableware. In *Proceedings of TRON EnableWare Symposium '92* (Dec. 1992), pp. 12–21. in Japanese.

[61] N. Koshizuka, M. Uematsu, N. Kimura, and K. Sakamura. Design and Implementation of the EnableWare Specification – A Human-Machine Interface for Physically Challenged People. In *Proceedings of the Ninth TRON Project Symposium* (Dec. 1992), IEEE Computer Society Press, pp. 23–39.

[62] N. Koshizuka. *C Interface to BTRON Graphical User Interface Shell (DRAFT) – version 0.1*. Sakamura Laboratory, Department of Information Science, Faculty of Science, University of Tokyo, May 1993.

[63] N. Koshizuka. *XSCHEME Interface to BTRON Graphical User Interface Shell (DRAFT) – version 0.2*. Sakamura Laboratory, Department of Information Science, Faculty of Science, University of Tokyo, May 1993.

[64] N. Koshizuka. Spatial Locking Algorithm. Internal Document in Sakamura Laboratory, University of Tokyo, 1993.

[65] N. Koshizuka, and K. Sakamura. Window Real-Objects: a Distributed Shared Memory for the Distributed Implementation of GUI Applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Nov. 1993), pp. 237–247.

[66] N. Koshizuka, and K. Sakamura. Highly Responsive Implementation of the BTRON2 Window System. In *Proceedings of the Tenth TRON Project International Symposium* (Dec. 1993), IEEE Computer Society Press, pp. 82–93.

[67] N. Koshizuka, and K. Sakamura. Design and Implementation of Multiuser Interface System Based on the Shared Interaction Object Model. in Japanese, in Submitting, 1993.

[68] N. Koshizuka, T. Mutoh, and K. Sakamura. TRON Human-Machine Interface Specifications for Everyday Life. *Journal of Information Processing Society of Japan*, Information Processing Society of Japan, (1994), in Japanese, *to appear*.

[69] T. G. Lane. User Interface Software Structures. Technical Report CMU-CS-90-101, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, May 1990. Doctorial Thesis.

[70] K. A. Lantz, and W. I. Nowicki. Structured Graphics for Distributed Systems. *ACM Transactions on Graphics*, Vol. 3, No. 1 (Jan. 1984), pp. 23–51.

[71] K. A. Lantz, and W. I. Nowicki. An Empirical Study of Distributed Application Performance. *IEEE Transactions on Software Engineering*, Vol. SE–11, No. 10 (Oct. 1985), pp. 1162–1174.

[72] K. A. Lantz. Multi-process Structuring of User Interface Software. *ACM Computer Graphics*, Vol. 21, No. 2 (Apr. 1987), pp. 124–130. Proceedings of the ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[73] W. Lehler. PIX, the Latest NeWS. In *Thirty-Fourth IEEE Computer Society International Conference* (1989), pp. 239–242.

[74] J. Levine. Why a Lisp-Based Command Language ? *ACM SIGPLAN Notice*, Vol. 15, No. 5 (May 1980), pp. 49–53.

[75] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer* (Feb. 1989).

[76] H. Maezawa. CTC-RPC/NIEP: Consistent Time Constrained Remote Procedure Call on Network ITRON Efficient Protocol. Master's thesis, Graduate School of University of Tokyo, Division of Science, Feb. 1993.

[77] H. Maezawa, H. Takada, and K. Sakamura. Design and Implementation of NIEP: Light-weight Protocol for Efficient Realtime Communications. *Proceedings of TRON Technical Workshop*, Vol. 5, No. 2 (Mar. 1993), pp. 51–65. in Japanese.

[78] K. Masuda, M. Ando, Y. Mimura, and T. Hashizume. Auto-operation Capability on BTRON Specification. In *Proceedings of the 40th National Convention of IPSJ* (Mar. 1990), IPSJ, pp. 769–770.

[79] T. Masui. User Interface Construction Based on Parallel and Sequential Execution Specification. *Proceedings of the USENIX Winter '91* (Jan. 1991), p. 117–125.

[80] T. Masui. User Interface Programming with Cooperative Processes. In *Languages for Developing User Interfaces* (B. A. Myers ed.), Jones and Bartlett Publishers Inc., 1992, p. 261–277 (Chapter 15).

[81] T. Masui, K. Hanada, and H. Otokawa. Groupware Based on Shared-Space Communication. *Information Processing Society of Japan SIG Notes on Programming – Languages, Fundamentals, Applications–*, No. 10–7 (Jan. 1993), pp. 49–56. in Japanese.

[82] J. McCormack, P. Asente, and R. R. Swick. *X Toolkit Intrinsics - C Language X Interface: X Window System X Version11, Release 2*. MIT Project Athena, 1988.

[83] MITSUBISHI ELECTRIC CORPORATION. *PC 9340 Reference Manual – Ver. 1.10A*, Jan. 1992.

[84] S. Mittal, C. L. Dym, and M. Morjaria. PRIDE: An Expert System for the Design of Paper Handling Systems. *IEEE Computer*, Vol. 19, No. 7 (Jul. 1986).

[85] B. A. Myers. User-Interface Tools: Introduction and Survey. *IEEE Software* (Jan. 1989), pp. 15–23.

[86] B. A. Myers. Why are User Interfaces Difficult to Design and Implement? Technical Report CMU-CS-93-183, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, Jul. 1993.

[87] B. A. Myers, and M. B. Rosson. Survey on User Interface Programming Tools and Techniques. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 195–202.

[88] E. D. Mynatt and W. K Edwards. The Mercator Environment: A Nonvisual Interface to X Windows and Unix Workstations. GVU Technical Report GIT-GVU-92-05, Geogia Institute of Technology 1992.

[89] E. D. Mynatt. Auditory Presentation of Graphical User Interfaces. GVU Technical Report GIT-GVU-92-26, Geogia Institute of Technology 1992.

[90] A. Nagasaki, and Y. Tanaka. Realizing the Cooperative Work Field in IntelligentPad. *Journal of Informantion Processing Society of Japan*, Vol. 34, No. 5 (May 1993), pp. 1177–1187. in Japanese.

[91] R. R. Newman-Wolfe, M. L. Webb, and M. Montes. Implicit Locking in the Ensemble Concurrent Object-Oriented Graphics Editor. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Nov. 1992), pp. 265–272.

[92] J. Nielsen, and T. K. Landauer. A Mathematical Model of the Finding of Usability Problems. In *Proceedings of INTERCHI '93* (1993), pp. 206–213.

[93] D. R. Olsen, Jr., T. G. McNeill, and D. C. Mitchell. Workspaces: An Architecture for Editing Collections of Objects. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 267–272.

[94] Open Software Foundation *OSF/Motif Style Guide*. Prentice-Hall, 1990.

[95] J. A. Orenstein, and F. A. Manola. PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Transactions on Software Engineering*, Vol. 14, No. 5 (May 1988), pp. 611–629.

[96] J. K. Ousterhout. Tcl: An Embeddable Command Language. In *Proceedings of the Winter 1990 USENIX Conference* (1990), pp. 133–146.

[97] J. F. Patterson. Comparing Programming Demands of Single-User and Multi-User Applications. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Nov. 1991).

[98] J. F. Patterson, R. D. Hill, S. L. Rohall, and W. S. Meeks. Rendezvous: An Architecture for Synchronous Multi-User Applications. In *Proceedings of the ACM CSCW '90* (Oct. 1990).

[99] C. D. Peterson. *Athena Widget Set — C Language Interface: X Window System X version 11, Release 4*. MIT X Consortium, 1989.

[100] G. E. Pfadd, Ed. *User Interface Management Systems: Proceedings of the Seeheim Workshop*. Springer-Verlag, 1985.

[101] M. Prime. User Interface Management Systems – A Current Product Review. *computer graphics forum*, Vol. 9, No. 1 (1990), pp. 53–76.

[102] J. Rees, W. Clinger, et al. Revised$^{3.5}$ Report on Programming Language Scheme. 1989.

[103] J. Rekimoto. CSCW Platform System Teidan and its Concurrency Control Algorithm. *Computer Software*, Vol. 10, No. 1 (1993), pp. 51–62. in Japanese.

[104] R. J. Rost, J. D. Friedberg, and P. L. Nishimoto. PEX: A Network-Transparent 3D Graphics System. *IEEE Computer Graphics & Applications* (Jul. 1989), pp. 14–26.

[105] K. Sakamura. A Proposal of New Unified Model of Man-Machine Interaction in the BTRON Environment. *Journal of Information Processing Society of Japan*, Vol. 26, No. 11 (Nov. 1985), pp. 1321–1328. in Japanese.

[106] K. Sakamura. BTRON: The Business-oriented Operating System. *IEEE Micro*, Vol. 7, No. 2 (Apr. 1987), pp. 53–65.

[107] K. Sakamura. *The Making of TRON*. Kyoritsu Publishing, 1987. in Japanese.

[108] K. Sakamura, Ed. *BTRON: TRON Keyboard Operation Summary*. TRON Association, 1988.

[109] K. Sakamura, Ed. *Enableware Specification – Extracts from BTRON MMI External Specification*. TRON Association, 1988. in Japanese.

[110] K. Sakamura. TACL: TRON Application Control-flow Language. In *TRON Project 1988* (1988), Springer-Verlag, pp. 79–92.

[111] K. Sakamura, Ed. *BTRON1 Specification Software Specification*. TRON Association, 1989.

[112] K. Sakamura. The TRON Intelligent House. *IEEE Micro*, Vol. 10, No. 2 (Apr. 1990), pp. 6–7.

[113] K. Sakamura, Ed. *ITRON Specification ITRON2*. TRON Association, 1990.

[114] K. Sakamura. Programmable Interface Design in HFDS. In *TRON Project 1990* (1990), Springer-Verlag, pp. 3–22.

[115] K. Sakamura. Design Policy of the Operating System Based on the BTRON2 Specification. In *TRON Project 1990* (1990), Springer-Verlag, pp. 103–118.

[116] K. Sakamura, Ed. *Specification of the Chip Based on the TRON Architecture*. TRON Association, 1990.

[117] K. Sakamura. TRON Application Projects: Gearing Up for HFDS. In *Proceedings of the Eighth TRON Project Symposium* (Nov. 1991), IEEE Computer Society Press, pp. 2–14.

[118] K. Sakamura, Ed. *BTRON2 Kernel Standard Handbook*. Personal Media Corporation, 1992. in Japanese.

[119] K. Sakamura, Ed. *ITRON/FILE Standard Handbook*. Personal Media Corporation, 1992. in Japanese.

[120] K. Sakamura, Ed. *Outline of CTRON*, Vol. 1 of *Original CTRON Specification Series*. Ohmsha, Ltd., Nov. 1992. in Japanese.

[121] K. Sakamura ed., TRON HMI Working Group (K. Sakamura, N. Koshizuka, T. Muto, et. al.), *TRON Human Interface Specifications*, TRON Association, 1993.

[122] K. Sakamura. Building Human Interfaces for Computerized Environments. In *Proceedings of the Tenth TRON Project Symposium* (Dec. 1993), IEEE Computer Society Press, pp. 21–28.

[123] K. Sakamura. Infrastructures for an Age of Computerized Environments. In *Proceedings of the Tenth TRON Project Symposium* (Dec. 1993), IEEE Computer Society Press, pp. 2–14.

[124] K. Sakamura, K. Tsurumi, and H. Kato. Applying the μBTRON Bus to a Music LAN. *IEEE Micro*, Vol. 8, No. 2 (Apr. 1988), pp. 60–66.

[125] H. Samet. Hierarcical Representation of Collections of Small Rectangles. *ACM Computing Surveys*, Vol. 20, No. 4 (Dec. 1988), pp. 271–309.

[126] R. W. Scheifler, and J. Gettys. The X Window System. *ACM Transactions on Graphics*, Vol. 5, No. 2 (Apr. 1986), pp. 79–109.

[127] B. Shneiderman. Direct Manipulation: A Step beyond Programming Languages. *IEEE Computer* (Aug. 1983), pp. 57 – 69.

[128] B. Shneiderman. Human Values and the Future of Technology: A Declaration of Empowerment. In *Proceedings of ACM SIGCAS Conference: Computers and the Quality of Life* (Sep. 1990), pp. 1 – 6.

[129] T. Shimizu, T. Yoshida, Y. Saito, M. Matsuo, and T. Enomoto. A 32bit Microprocessor Based on the TRON Architecture: Design of the GMICRO/100. In *Proceedings of the 33rd IEEE Computer Society International Conference* (1988), IEEE Computer Society Press, pp. 30–33.

[130] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the Star User Interface. *Byte*, Vol. 7, No. 4 (Apr., 1982).

[131] D. C. Smith and J. Susser. A Component Architecture for Personal Computer Software. In *Languages for Developing User Interfaces* (B. A. Myers ed.), Jones and Bartlett Publishers Inc., 1992, p. 31–56 (Chapter 4).

[132] R. Stallman. *GNU Emacs Manual, Sixth Edition, Emacs Version 18*. Free Software Foundation, 1987.

[133] R. M. Stallman. EMACS the Extensible, Costomizable Self-Documenting Display Editor. In *Proceedings ACM SIGPLAN/SIGOA Conference Text Manipulation* (Jun. 1981).

[134] STANFORD UNIVERSITY. *InterViews Reference Manual, Version 2.5*, 1989.

[135] M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meeting. *Communications of the ACM*, Vol. 30, No. 1 (Jan. 1987), pp. 32–47.

[136] M. Stumm, and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, Vol. 23, No. 5 (May 1990), pp. 54–63.

[137] SUN MICROSYSTEMS, INC. *NeWS 1.1 Manual*, 1988.

[138] Sun Microsystems, Inc. *ToolTalk Service: An Inter-Operability Solution*. PTR Prentice Hall, 1993.

[139] Sun Microsystems,Inc. OPEN LOOK Graphical User Interface Functional Specification, Release 1.0. Addison-Wesley, 1989.

[140] H. C. K. Sung. UIPEX: A 3D Graphics Extension to the X Window System. Master's thesis, The Graduate College of the University of Illinois at Urbana-Champaign, 1990.

[141] P. P. Tanner, S. A. MacKay, and D. A. Stewart. A Multitasking Switchboard Approach to User Interface Management. In *Proceedings of the ACM SIGGRAPH '86 Conference* (Aug. 1986).

[142] R. N. Taylor and D. Clément. Separations of Concerns in the Chiron-1 User Interface Development and Management System. In *Proceedings of INTERCHI '93* (Apr., 1993), ACM, pp. 367–374.

[143] C. Thorén, Ed. *Nordic Guidelines for Computer Accessibility*. Nordiska Nämnden för Handikappfrågor, 1993.

[144] G. C. Vanderheiden. Making Software More Accessible for People with Disabilities – A White Paper on the Design of Software Application Programs to Increase Their Accessibility for People with Disabilities –. *ACM SIGCAPH Newsletter*, No. 47 (Jun. 1993), pp. 2–32.

[145] B. F. Webster. *The NeXT Book*. Addison-Wesley Inc., 1989.

[146] P. H. Winston, and B. K. P. Horn. *LISP – Third Edition*. Addison-Wesley Publishing Co., 1989.

Publications by the Author[1]

1. Noboru Koshizuka, Toshio Mutoh, and Ken Sakamura, "TRON Human Interfaces Specification for Everyday Life", *Journal of Information Processing Society of Japan*, 1994, *in Japanese, to appear*.

2. Tomoyuki Tanuma, Noboru Koshizuka, and Ken Sakamura, "Design and Implementation of Tools for Manipulating OS Resources on BTRON", In *Proceedings of the Fifteenth TRON Technical Workshop*, Vol. 6, No. 1, TRON Association, Mar., 1993, *in Japanese, to appear*.

3. Noboru Koshizuka and Ken Sakamura, "Highly-Responsive Implementation of the BTRON2 Window System", In *Proceedings of the Tenth TRON Project International Symposium*, IEEE Computer Society Press, Dec., 1993, p. 82–93.

4. Akira Matsui, Noboru Koshizuka, and Masaaki Kobayashi, "How to Educate a True Beginner about Computers", In *Proceedings of the Tenth TRON Project International Symposium*, IEEE Computer Society Press, Dec., 1993, p. 95.

5. Noboru Koshizuka and Ken Sakamura, "Window Real-Objects: A Distributed Shared Memory for Distributed Impelmentation of GUI Applications", In *Proceedings of ACM Symposium on User Interface Software and Technology (UIST '93)*, Association of Computer Machinery, Nov., 1993, p. 237–247.

6. Ken Sakamura ed., TRON HMI Working Group (Ken Sakamura, Noboru Koshizuka, Toshihisa Muto, et. al.), *TRON Human Interface Specifications*, TRON Association, 1993, (English version of the publication 7).

7. Ken Sakamura ed., TRON HMI Working Group (Ken Sakamura, Noboru Koshizuka, Toshihisa Muto, et. al.), *TRON Human Interface Specifications for Computers in Everyday Life*, Personal Media Corporation, 1993, *in Japanese*.

[1] Sorted by Publication Date

8. Michimasa Uematsu, Noboru Koshizuka, and Ken Sakamura, “A Proposal of Multilingual Representaion Mechanism on BTRON”, In *Proceedings of the Forteenth TRON Technical Workshop*, Vol. 5, No. 2, TRON Association, Mar., 1993, p. 1–18, *in Japanese*.

9. Reiko Kondo, Noboru Koshizuka, and Ken Sakamura, “An Extension of BTRON HMI Specification for Multi-user Applications”, In *Proceedings of the Forteenth TRON Technical Workshop*, Vol. 5, No. 2, TRON Association, Mar., 1993, p. 31–38, *in Japanese*.

10. Michimasa Uematsu, Noboru Koshizuka, Hiroaki Takada, and Ken Sakamura, “BTRON Multilingual Panel Manager”, In *Collected Papers on BTRON Multilingual Processing*, TRON Association, Jun., 1992, p. 83–108, (English version of the paper 17).

11. Noboru Koshizuka, and Ken Sakamura, “Software Architecture and Application Program Interface Model for the BTRON GUI Shell”, *Technical Report 92-4*, Department of Information Science, Faculty of Science, University of Tokyo, 1992.

12. Noboru Koshizuka, Reiko Kondo, and Ken Sakamura, “A Proposal of Special User Interface Parts Model”, In *Proceedings of the Eleventh TRON Technical Workshop*, Vol. 4, No. 3, TRON Association, Mar., 1992, p. 35–62, *in Japanese*.

13. Noboru Koshizuka, Michimasa Uematsu, Norio Kimura, and Ken Sakamura, “Design and Implementation of the EnableWare Specification — A Human-Machine Interface for Physically Challenged People”, In *Proceedings of the Ninth TRON Project Symposium*, IEEE Computer Society Press, Dec., 1992, p. 23–39.

14. Noboru Koshizuka and Ken Sakamura, “An Overview of the BTRON Enableware”, In *Proceedings of the TRON Electronic Prosthetics Symposium ’92 (TEPS ’92)*, TRON Association, Dec., 1992, p. 12–21, *in Japanese*.

15. Noboru Koshizuka, Hiroaki Takada, Ken Sakamura, “Window Shared Data Pool: A New BTRON Window System”, In *Proceedings of the Eighth TRON Project Symposium*, IEEE Computer Society Press, Dec., 1991, p. 151–168.

16. Noboru Koshizuka, Hiroaki Takada, Ken Sakamura, “Window Shared Data Pool”, IPSJ SIG Notes on Operating System, 52–2, Information Processing Society of Japan, Sep., 1991, *in Japanese*.

17. Michimasa Uematsu, Noboru Koshizuka, Hiroaki Takada, and Ken Sakamura, "BTRON Multilingual Panel Manager", In *Proceedings of the Tenth TRON Techniqcal Workshop*, Vol. 4, No. 1, TRON Association, 1992, p. 1–15, *in Japanese*.

18. Noboru Koshizuka, Naoki Inoue, Hiroaki Takada, Nobuhiko Nishio, and Ken Sakamura, "Interface Manager Based on Abstract Event for BTRON", In *Proceedings of the Sixth TRON Technical Workshop*, Vol. 2, No. 4, TRON Association, 1990, p. 1–22, *in Japanese*.

19. Noboru Koshizuka, Masaharu Saitoh, Yasushi Saitoh, Hiroaki Takada, and Ken Sakamura, "Implementation Issues of TACL/TULS Language System on BTRON", In *TRON Project '89–'90*, Personal Media Corporation, 1990, p. 123–143, *in Japanese*, (Japanese version of the paper 21).

20. Noboru Koshizuka, Masaharu Saitoh, Yasushi Saitoh, Hiroaki Takada, and Ken Sakamura, "Implementation Issues of TACL/TULS Language System on BTRON", *Technical Report*, Department of Information Science, Faculty of Science, University of Tokyo, 1989 (Reprint from the paper 21).

21. Noboru Koshizuka, Masaharu Saitoh, Yasushi Saitoh, Hiroaki Takada, and Ken Sakamura, "Implementation Issues of TACL/TULS Language System on BTRON", In *Proceedings of the Sixth TRON Project Symposium (TRON Project 1989)*, Springer-Verlag, Dec., 1989, p. 113–132.