1987

Partitioning the Process of Interaction: An Abstract View

Balachander Krishnamurthy

Report Number:
87-705

PARTITIONING THE PROCESS OF INTERACTION: AN ABSTRACT VIEW

Balachander Krishnamurthy

CSD-TR-705
September 1987

Partitioning the Process of Interaction: An Abstract View

Balachander Krishnamurthy

Department of Computer Sciences
Purdue University
W Lafayette, IN 47907

September 7, 1987

Abstract

We present an abstraction mechanism to identify the layers of interaction between the user and application programs in a workstation environment. Our first goal is to reduce the task of the interface programmer and enable him to provide a uniform interface to a variety of application programs. We require the interface to be customizable as well. Secondly, we want to minimize interaction related information in the application programs and reduce the task of the application programmer as well.

The abstraction mechanism has two parts: one (package configuration file, or PCF) which consists of interaction information extracted from the application program, and the other (user configuration file, or UCF) which is used by the user to specify his style of interaction. The information extracted from the application programs is encoded into the PCF and a generic parser parses this information at session start time. Similarly, the information relating to the user's bindings from input events to the actual functions he wishes to invoke, his choice of styles of interaction with the various application programs, etc., are gathered in the UCF which is consulted both at startup time and during the course of the session as well.

1 Introduction

Software systems are constantly getting complex and users are faced with the task of accessing an ever-increasing set of programs. A key factor in computing is the interaction process between the user and the programs. Often, the interface to a program ends up being the deciding factor in its usefulness. The application programmer constructs programs, the interface programmer builds interfaces to the programs for the user to use. In this paper we attempt to define the roles of the application programmer, the interface programmer, and the user, to improve the process of interaction. The user can interact with the various application programs in different styles. A goal of this work is to delineate a model of interaction that would provide the user with a uniform, yet flexible style of interaction with the various application programs. We do this by providing the proper abstraction between the user and the application programs.

The collective term for the various application programs accessible to the user and the ways in which they can be accessed is an *interactive system*. We formally define an interactive system as a set of input devices and a *window manager* that manages a set of windows used to partition the user's contexts. The window manager multiplexes user input and redirects it to processes running in the windows. Output from the processes is demultiplexed by the window manager as well. Our aim in this paper is to study the different parts of the model of interaction and come up with a firewall between the application programs and the users. The firewall enables an uniform, customizable interface to be constructed.

The rest of this paper is divided into seven sections. We begin by motivating the need for a uniform, customizable interface and exploring some of the problems in existing interactive systems. The need to partition the tasks of the interface programmer and the application programmer is discussed next. After presenting the abstraction for partitioning we examine if application programs need be modified to improve the interaction. We then consider the interface between the application program and the window system as well as the interface between the window system and the user. Another report [7] covers the implementation of a prototype window system based on the model presented in this paper.

2 Motivation

We are interested in establishing the tasks of the application programmer and the interface programmer. The goal is to design a uniform and customizable interface to an interactive system consisting of diverse application programs. The window manager demultiplexes user input and redirects it to the various application programs. Likewise, the output generated in the programs is redirected to the appropriate places. Thus, the window manager performs the role of an input/output mediator between the user and the application programs. The advantage of such a break up is that the application program is divorced from having to gather input and a uniform input gathering scheme can be used across all application programs. Thus, our task is to look at the role of the window system in partitioning the interaction process, and the abstractions needed for such a partitioning. We seek to reduce the work of the application programmer by removing interface related details from the application programs.

One of the contributors to the goal of uniformity is the concept of generic commands. Uniformity of user interface demands uniform access to all objects in the environment. The user should be able to use different subsystems (packages) without having to learn a new command syntax. The command interface, which is usually the crucial part as far as the user is concerned, should be generic. A short list of select commands applicable across various contexts has been termed *generic commands* in the literature. The Xerox STAR environment [10] and the Vitrail system [14] use generic commands widely. Our approach differs from these systems in the sense that our focus is on uniformity *with* flexibility.

Generic commands enable the user to interact with several subsystems in a similar manner, and are responsible for maintaining uniformity and coherence in the user interface. Generic commands are bound to the appropriate backend command in the current context. We defined generic commands to be the small set of commands that are used across a wide variety of contexts. A majority of the commands in each of the subsystem can be matched by a small set of generic commands.

The generic command layer maps the user-issued generic commands to appropriate commands in the current context. The task of building this map is left to the interface programmer. The interface programmer takes the pre-existing set of generic commands and maps them into the appropri-

ate backend commands of a particular application program. Thus, the user need not be aware of the subsystem specific bindings. We want to maintain this transparency to prevent the user from having to learn a new set of commands when he uses a new application program. The user can continue to think logically of what a command should do in the present context and have that translated into action without worrying about the specifics of the application program. Further, the user can aid in perpetuating this transparency by having his own binding from input events to the generic command. Thus, a user would continue to issue the same input event that he issues for *delete* in all contexts, and the same input event would perform the *generic-delete* in the new context. The interface programmer should have already bound *generic-delete* to the appropriate delete action in the new context.

Figure 1 shows the bindings from input events to the generic commands and from generic commands to the various backend functions in the application programs ($AP_1, \ldots, AP_n$). Even though the figure shows the bindings from the input events to the generic commands to be one-to-one, more than one input event can be bound to a single generic command. But the binding from the generic commands to the backend functions in a given application program is always one-to-one.

An additional advantage of the generic command scheme is that it can be used in an ordinary terminal with special keys (such as function keys) being used for generic commands. In workstations with bitmapped displays, generic commands can be presented in the guise of menu-items with pointing devices such as mouse being used to invoke them. As the bindings from input events to generic commands fall under the category of *local* commands, they can be changed by the user at any time. Such a change is transparent to any of the application programs that the user may access.

Generic commands provide uniformity in key bindings both from the keyboard as well as pointing devices such as a mouse. Currently the bindings from mouse transitions are done in an arbitrary manner in various applications. For example, in SunView [12], the right button stuffs selected characters in the editing context while it *undoes* an insertion in the context of a tool used to construct icons. By treating a mouse transition as a binding to a generic command, we can expect the same mouse transition to perform similar actions in all applications. The mouse performs

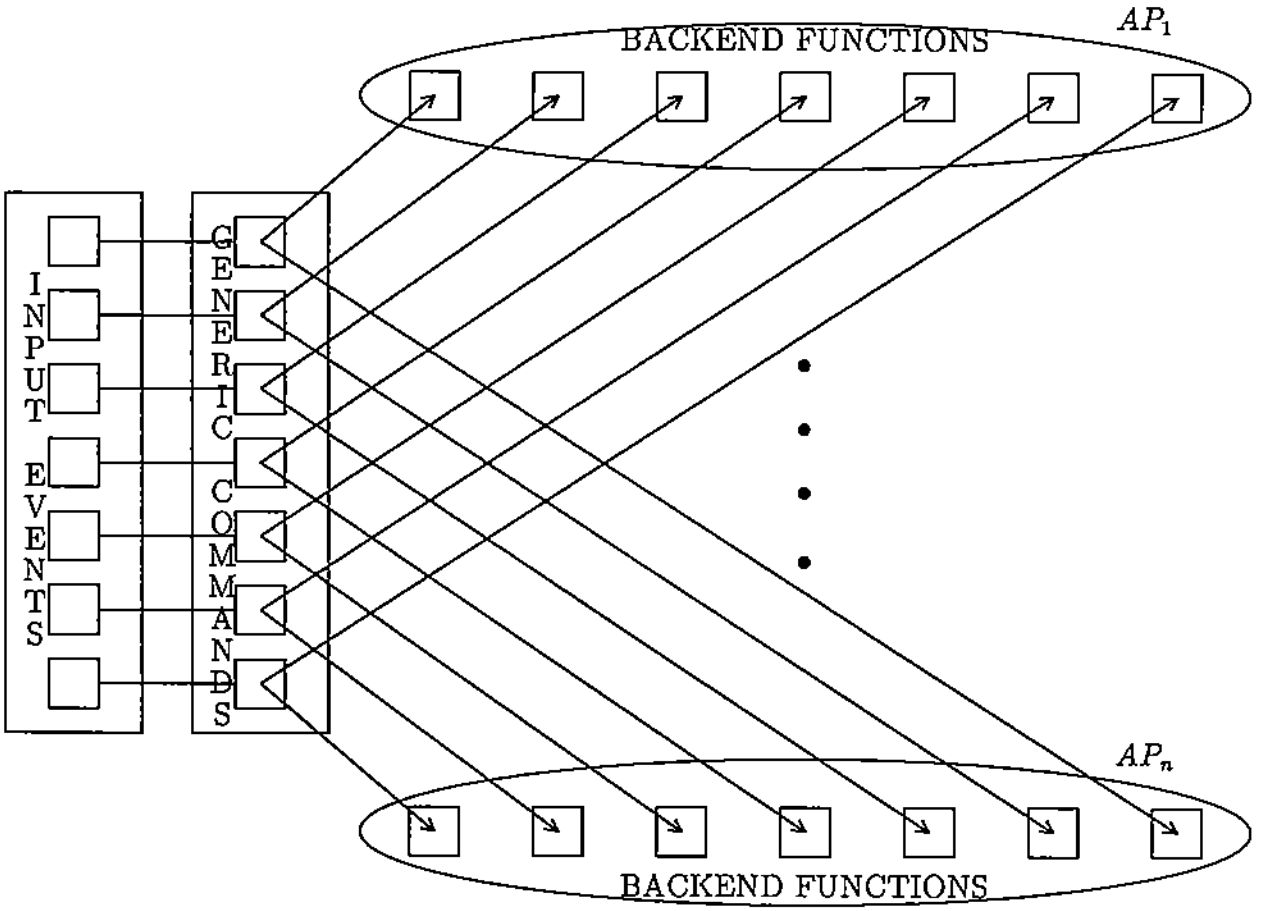

Figure 1: The two level binding

generic window system operations while executing window manager specific commands. An example of an environment where mouse clicks have generic operations to a certain extent is the Macintosh personal computer. Apple set a user interface standard by having a file menu with commands like *Open*, *Close*, *Save*, *Quit* etc, for every application. Similarly, most of the applications have an *Edit* menu with the commands *Undo*, *Cut*, *Copy*, and *Paste*. In editors, these commands do editing operations, but in editing resources *(Resedit*, a resident system modifying program), they delete, copy and move resources. Unfortunately, some commands are not always used to do the obvious things—*Open* does not always open a file. The bigger drawback is that the bindings from these generic commands cannot be changed.

Generic commands are not limited to user interface research: user interface research is now having an effect on the design of operating systems. We present an example of how the need for uniformity in user interfaces has led to a novel file naming scheme. In devising a scheme to address the file naming problem two approaches are common: one either adopts an existing naming scheme or invents a new one. In the XINU operating system [1] we find a novel scheme: addition of a syntactic naming mechanism that accommodates several different underlying schemes. The namespace software is responsible for transforming the user supplied names into names suitable for the underlying system. The advantages of such a scheme are clear: new file systems can be added dynamically without recompilation of programs that use them and existing file systems and devices can be integrated into a uniform namespace. The namespace scheme is thus similar to the generic commands idea. The mapping is done at a higher level and at a level accessible to the user. As the namespace collects the various file naming schemes into a single access scheme (the user opens files the same way) uniformity is achieved. Yet there is flexibility in that new and different file naming schemes can be used alongside existing ones.

We believe that a uniform input gathering scheme applied by the window system outweighs the potential advantages of some application programs using special input gathering techniques. The special input gathering scheme of certain application programs can be generalized and incorporated on a global basis in the window system's input gathering mechanism. The window system bears the responsibility of providing alternate means for the user to take advantages of special features offered by individual application

programs. Let us consider the example of completion. The UNIX command interpreter *C-shell* [6] is capable of completing the file names that the user types when the file name completion character is issued. Another application program, *webster*, a English language dictionary lookup program, takes words as input, and performs word-completion when a completion character is typed. If however, *C-shell* is invoked from within a window system where the file name completion character has a different significance, then the user will not be able to take advantage of the file name completion ability. The completion character, has been usurped by the window system.

The logical nature of the completion character is robbed of its value by tying in a physical input event in the source code. A *generic* completion event that all application programs can use, and one that can be defined and changed at will by the user at the window system level, solves the problem of application programs usurping different input events for similar functions. With such flexibility one has to safeguard against incompatible changes. Also, an escape mechanism is needed if the completion physical input event needs to be interpreted in a different sense.

The window system should be responsible for performing the completion task as the potential for flexibility in the user's style of interaction will be higher. In a uniform model of input gathering, completion should be provided at all levels by the window system. Completion should be provided any time the user has to supply an argument from a finite set of arguments. When using an electronic message system that provides *folders* to store context specific messages, the window system should build a *user-specific* table of folders and permit the user to specify the least ambiguous substring of a folder name. The same holds true for file names, commands and arguments, names of groups in electronic bulletin board systems, etc. Completion is harder in a text-editing environment as the range of completion is potentially infinite. The completion idea has been implemented in different guises in extensible editors. For example, Emacs has user-specified abbreviations. Global abbreviations work across all contexts while local abbreviations are context specific. Thus, the same abbreviation string can be expanded to different strings in different contexts. While this might seem analogous to the completion table formed in our example of folders, the difference is that the completion table can be automatically generated, whereas the abbreviations have to be specified by the user.

The notion of a single input event to request completion in all contexts is

an example of our generic input event idea. Other examples of generic input events that can be used across various application programs include a query event that returns a list of possible completions, and a help event which provides documentation on commands and aids the user during command construction.

We conclude that the window system should enable the users to deal with generic operations in various application programs via logical input events. In the next section we expand on the partitioning of the interaction task.

3 Advantages of partitioning

By partitioning the user/application interface, we can partition the roles of the application programmer and the interface programmer. The task of the application programmer is simply writing the application program. More to the point, it should not be their task to worry about the interface at a low level. The application programmer cannot be expected to provide a variety of interfaces for a single application program. However, he should not prevent or hinder the interface programmer from doing so. Building a firewall between the application programmer and the user will aid in the clear delineation of tasks of the application programmer.

One important fact we have to keep in mind is that while we advocate partitioning the *interaction tasks* between the user and the application programs there is no necessity for the interface between the user and the window system to be different from that between the user and the application program. In Section 7.5 we will expand on this notion of uniformity in the user's interaction with the window system and the application program.

4 Abstractions for partitioning

We have thus seen that the window system can provide a *uniform*, *customizable* front end for the different application programs by taking over the task of gathering and interpreting the input. Similarly the output from the application programs can be handled by the window system to provide the user with greater leverage in dealing with the output. The partitioning

of tasks between the application programs and the window system thus demarcates the lines of control. The application program's task is to accept input that has been checked for syntactic correctness and service the request. The speedup at this level is done by caching as much information about the commands as possible. After execution of the command, the application program returns the raw output to the window system. The window system decides how to display the output based on the user's preferences. Thus, there are two interfaces to deal with: one is the interface between the application programs and the window system, and the other is the interface between the user and the window system. We now introduce our abstractions for the conceptual break up of the interactive system.

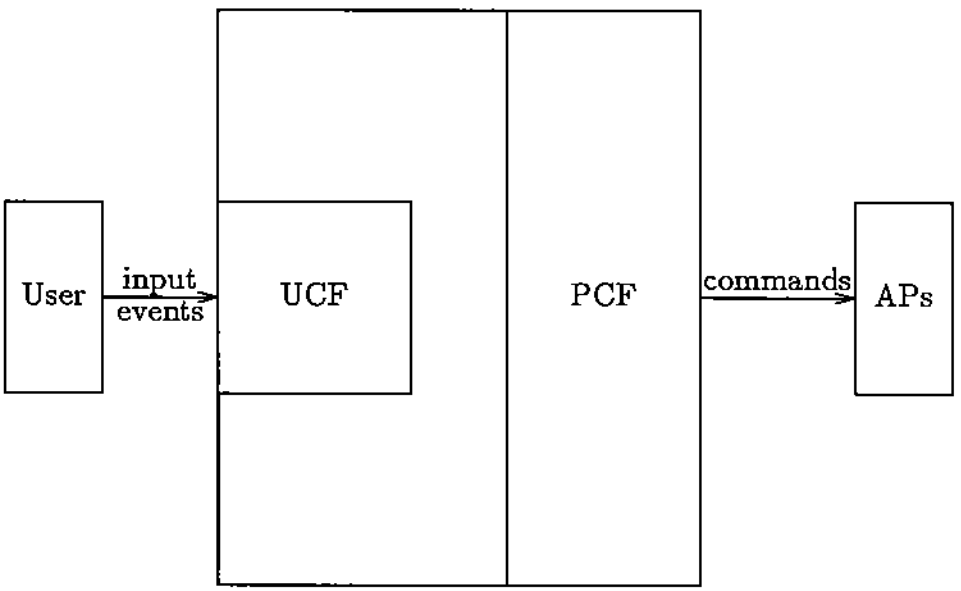

Figure 2: Firewall between the user and application programs

We break the interaction process into two stages with the two parts of our abstraction being responsible for each of the stages (Figure 2). The first abstraction represents the interaction between the user and the window system is termed the *user configuration file* or UCF. Input events from the user are handled in consultation with the UCF. The second abstraction is the interaction between the window system and the application pro-

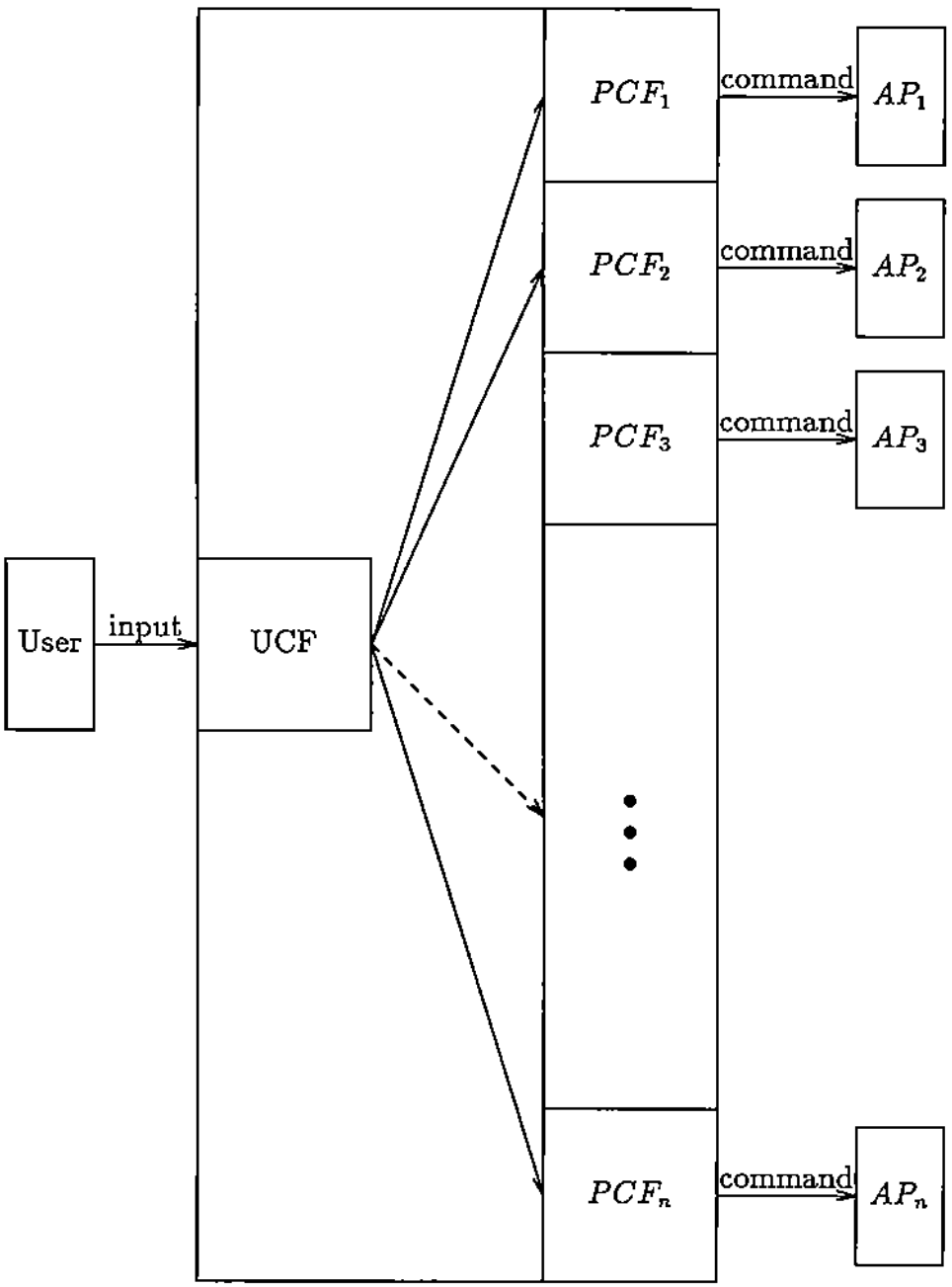

Figure 3: The UCF/PCF/AP division

gram. The abstraction representing this interaction is termed the *package configuration file* or PCF. The input events have now been converted into commands and are executed by the application program. The UCF and the PCF constitute a firewall between the user and the application program, insulating the user from the potentially non-uniform, non-customizable aspects of application programs. A detailed explanation of the actual makeup of the PCF and UCF is presented in [7]. Figure 3 represents a complete interactive system with several application programs and PCFs.

5 Should application programs be modified?

While partitioning, we had to decide if the application program had to be modified to support our interface requirements. In particular, to provide configurable, customizable styles of interactions, locally and across all application programs, what are the changes required in the application program? The answer depends on the degree of flexibility we want in the interactive system. We were able to demonstrate our ideas without making any changes to the application program. However, there are certain advantages to changing the application program. The information needed *a priori* about the application program, that we gather and encode in the package configuration file, can be obtained *on the fly* if the application program can be modified slightly to transmit the required information to the window system on startup. Thus, the window system would be an intelligent, learning window system that is capable of *learning* about the application program and constructing different styles of interaction based on the current version of the application program. The window system will remain oblivious to changes in the application program.

6 Application/window system interface

In our model we are interested in interaction in a window system that provides a front end to several different application programs. Examples of application programs are mail-handlers, revision control systems, event-based schedulers, debuggers. The application programs are distinct units that use the window system specifically for input and output. The user's

input (command) is sent to the application program after a potential partial interpretation. The application program acts on the input data, executes the command, and returns the output. Rather than forcing the application program itself to display the output, we allow the window system to interpret the output locally on the workstation and display it in a user-specified format. The user can specify *how* he would like to have the output displayed. The application program may well be executing at a remote site and so interactive manipulation can be sped up by permitting the local workstation to have complete control over the output. For example, an application program that draws graphs in a window, can have the image scaled to the user's liking at the workstation level without requiring the application program to do the work. Apart from lowering the communication overhead, local processing power of the workstation is better utilized.

We agree with Coutaz's [2] approach of strict separation of application programs from the interactive interface (the window system and the user helper process) with the stipulation that the strictness can be altered depending on the architecture upon which the interactive system is constructed. We now look at the communication between the workstation software and the application programs to see if it can be genericized.

6.1 Communication

The granularity of communication between the window system and the application program is a *command line*. The command line consists of the *command*, *flags*, *options*, and *arguments*. A command is a request made by the user to the interactive system to perform some action on his behalf. A *flag* is a boolean value and specification of the flag implies it being true while its absence signifies the opposite. An *option* on the other hand has two components: the name of the option and its value—a (*name,value*) pair. An option in general can be one of a few types, such as character strings or integers. Flags and options are associated with the semantics of the command in that their presence or absence signifies a difference in the *nature* of the command. An *argument* is separate from a flag as well as an option, as arguments do not affect the nature of the command. Commands act *on* arguments. Our definition here differs from a proposed standard for UNIX system commands presented in [5] only in the separation of the notions of flags and options.

All, except the command itself are optional. The window system bundles the command line and sends a request to the application program in the form of a *remote procedure call.* The command is executed and the output is sent back to the calling routine, which remains blocked till it receives a reply from the application program. The window manager has the potential to do more in certain commands where the interface programmer would like to take advantage of some available knowledge. For example, if the interface programmer knows about the display hardware he may provide the user with choices in fonts and the window manager can inspect the user's current choices before displaying the output.

Traditionally, application programs take it on themselves to provide documentation for the commands and reporting error messages. In our model, to speed up the response we download objects relevant to the interaction process into the window system, lowering the communication overhead between the application program and the window system.

The objects that are downloaded into the window system to reduce the communication overhead are information about the arguments, documentation, and error messages. The model is open in the sense that further objects that need to be downloaded can be done with only minor additions. Information regarding the arguments, flags, options of the valid commands in the application program can be used to assist the user in specifying commands. At the same time, local syntactic checking can be done on the user supplied arguments. Local checking at the workstation level prevents the user from having to wait a lengthy time only to find out that the arguments he supplied were of the wrong type or were otherwise invalid. Often, during interaction the user tries to obtain some quick help for a particular command, either for finding syntactic details of the command or some semantic detail. Rather than invoking the help facility of the application program, the window system caches a condensed help message for the individual commands. The documentation, being closer to the user, can be displayed as needed, obviating repeated interaction with the application program. Our model assumes a hardware configuration where the window system has reasonable amount of local memory to hold such data.

When an error is discovered, the application program notifies the remote procedure calling routine of the error. The window system is responsible for handling the display of the error message in a uniform manner. Once again user intervention is possible—he can specify where and how error

messages should be displayed. The window system takes this specification into account before displaying the error message. In matters regarding physical information, such as where output should be displayed and where viewports should be located, the application program can provide default values. However, rather than being under the exclusive control of the application program, the user should be able to specify *his* defaults. Further, he should be able to specify all the physical attributes at startup time and modify them during the session. Finally, he should be able to make these changes locally for a particular application program or globally for all application programs.

6.2 Can the interface be generic?

Simply partitioning the tasks between the application program and the window system will not suffice. We have to answer the following questions:

- Who is responsible for partitioning and performing the initial work?
- Can the interface between various application programs and window system be generalized?

The interface between the application program and the window system is devised by the interface programmer when the interactive system is constructed. The interface programmer has the knowledge about the application program as well as the interface details. He is well equipped to provide an interface in keeping with existing interfaces. A goal of our model is to reduce the task of the interface programmer by marking the boundaries at which the interface programmer should partition the tasks between the application program and the window system.

The basic unit of communication between the window system and the application program is the command line. As a significant percentage of application programs have the simple protocol of taking a command line as input and executing it, there need be no significant differences between the interfaces. Thus, it is possible to derive a common schema by which we can represent the interface between the various application programs and the window system. We can then automate the process of interaction between the application programs and the window system. Based on this idea, we developed an encoding scheme that can be parsed by a generic

parser. It remains to be studied if the various styles we demonstrate can also be generated.

7 Window system/user interface

Having seen the interface between the application program and the window system, we now turn our attention to the user interface. Several of the software environments that have been developed in the past few years are either oriented toward the programmers (known as "programming environments") or toward non-programmers as an aid for interaction with the underlying system. Some of them recognize the crucial difference between advanced and novice users, and try to ensure that the interface provided to the user is friendly. Other systems are explicitly oriented towards the expert users.

The problems that arise in the area of user interfaces are manifold, but they can be condensed into a search for an ideal interactive program development environment that can also be used by the novice users to perform their tasks. The ideal user interface, in our view, should satisfy several criteria including customizability, extensibility, user friendliness, and expert friendliness. Customizability is the ability to tailor the interface to one's liking. Extensibility is the ability to augment the list of available commands using the system primitives themselves. User friendliness has been defined as "a measure of the distance between the things the user thinks about doing and the things the user actually can do in the system" [4]. Expert friendliness has been stressed as a valid need: "For experts, the desire for common operations to require a minimum of human effort often rightly takes precedence over the desire for the greatest possible uniformity or simplicity in the human interface" [3]. Other criteria can be viewed as aids in improving the interface *per se*. These include minimization of effort and delay, immediate feedback [8], natural interface, readily available help [9] among others. A factor not mentioned above, viz., *uniformity* has been already discussed in Section 2 under generic commands.

While discussing the window system/user interface we take all the above mentioned factors into consideration. Our interest is in constructing an interface that has the desired features at the right level in the interactive system. Features such as *customizability*, *configurability*, *extensibility* should

not result in increased space requirements or response delays. We take a closer look at whether these features are mutually exclusive, and if they are always desirable. Our approach is to examine the primitives of interaction involved at each layer of the interactive system.

7.1 User interface

Comer defines user interface as the hardware and software with which users interact to specify computation and observe the results. He further states that "the term usually refers to the interactive software that accepts commands from the user and carries out the processing they specify"[1]. While a user interface can be considered to be the communication protocol between the user and the machine, we are concerned here with the interface between the window system and the user. The window system has the task of gathering the raw input issued by the user via a variety of input devices such as keyboard, mouse, or tablet, and constructing a context-dependent command out of it. We defined an *input event* in our model as a discrete, recognizable event emanating from any of the input devices. The window system should be able to serialize the input, ensure that the input gets redirected to the right place and permit alteration of the focus of attention.

7.2 Customizability

If the window system were capable of doing its task in a pre-defined manner alone, then it is not a flexible entity. A chief requirement of the window system has to be that it be changeable. If a keystroke is bound to an action there should be a provision to change this binding, either to bind another keystroke to the same action or to change the binding of the keystroke to another action. The same holds good for any style of interaction, be it keyboard oriented, or mouse. For the same application program, different styles of interaction are possible. The task of deciding the style of interaction for a particular application program or for all application programs should be left to the user. The task of the interface programmer is simply to *enable* different styles of interaction for all application programs. The user might prefer to use the same style of interaction for all application programs or a different one for each. He should be able to specify his choice of a style of interaction at startup time or use a global default. Further, he should

be able to change the style *while* interacting with the application program either locally (that application program alone) or globally (all application programs).

Some of the common *input styles* include *keyboard-style*, in which a user types a command line, *mouse-style*, in which the user selects commands from a fixed menu or a pop-up-menu. Similarly there are different *output styles*. For example, the output from commands can be displayed in the same window in which input was issued, or in a different window. The user should be able to specify and modify his input styles as well as his output styles.

We thus have two measures of customizability. First, it is the ability to specify and modify, both locally and globally, the choices of styles of interaction with the application programs. Second, it is the ease with which an advanced user can modify the interface, without the novice user having to specify a large number of default option values. In the remainder of this section we consider the various aspects of customizability.

7.2.1 Keymaps

We define a *keymap* as the binding from the keystrokes to the functions they invoke. A keymap is a many-to-one, or one-one function. Even though it is called a keymap, input events from pointing devices can be bound to functions. The notion of a keymap is integral to customizing the interface to an interactive system.

In our model, we have broken down the keymap into three layers (figure 4). The first layer is the physical input event (PIE) layer, the set of physical events that the user can issue. These include keyboard events, mouse transitions etc. The second layer is the actual input event (AIE) layer consisting of the events emanating from the input hardware after the user has issued physical events. The third layer is the logical input event (LIE) layer where the physical events are converted to logical events which are then bound to generic commands or backend functions. Figure 4 displays the various stages in the binding of the input events to the backend functions executed. The user issues physical input events via the keyboard and mouse input devices. These are then converted to actual input events by the input hardware. The window system subsumes the time factor (deciding what the complete input event is) and converts it to a logical input

event using the logical input event handler. The logical input event is then bound to a actual backend function which is then executed.

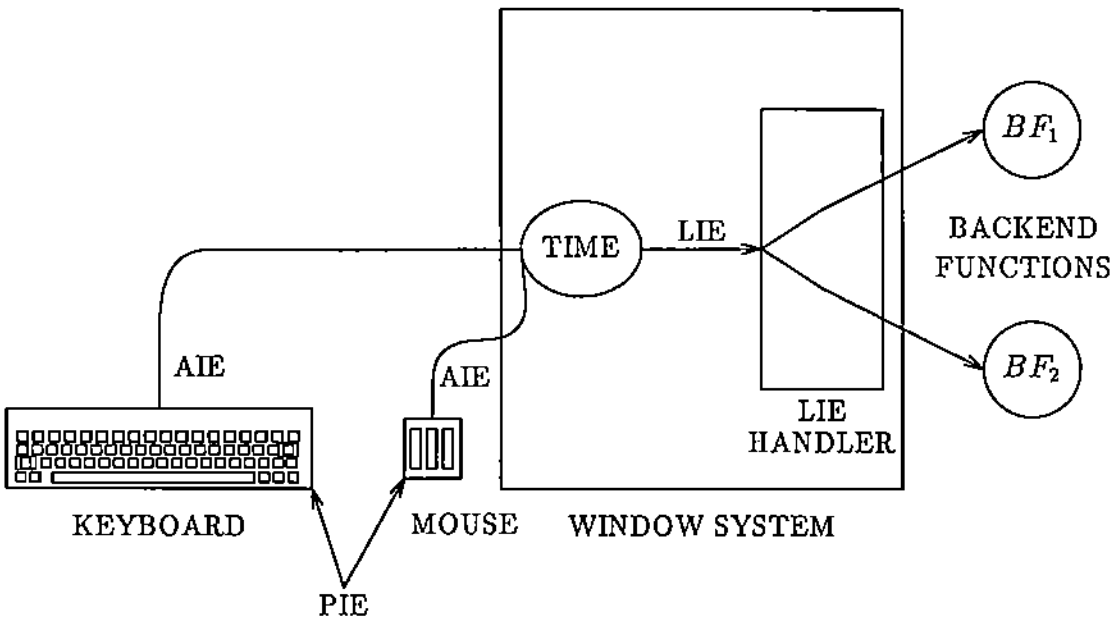

Figure 4: The three layer keymap model

When interactive systems exhaust single physical input event bindings to functions, they tend to use multiple input events as a two-level binding. The first of these multiple physical input events is called a *prefix* key. The prefix key causes the input handler to branch to an appropriate map, and the subsequent physical input event directs the binding to the appropriate function. Interactive systems can support multi-level *keymaps*. Unfortunately, the prefix-keys are "hard-wired" into the interactive system, i.e. once a certain key has been chosen as the prefix key, every keystroke sequence beginning with the prefix will have to be interpreted via the secondary keymap. We should be able to

- dynamically define prefix keys,
- change the meaning of prefix keys, and
- remove the prefix property of a key.

We can do all these by using the generic command layer to do the actual prefix specification. We move the interpretation of the prefix property of a prefix key away from the control of the backend functions and into the window system where the interface programmer can decide how a key should be interpreted. By introducing the notion of *logical prefix-keys*, we permit the user to define prefix-keys on the fly with mnemonic prefix-keys. For example, the user may want to define "window-prefix" for commands dealing with window operations, "file-prefix" for commands dealing with files, etc. At any time the user can redefine the binding of a logical prefix key to any physical key of his choice. The thrust of this idea is to ensure complete customizability and prevent interactive system designers from burying decisions affecting the interface deep in the system. Present day interactive systems display several examples of such premature design decisions.

The strength of our model as far as customizability results from the user's ability to rebind any input event of his choice to the generic commands. While the bindings from the generic commands to the actual backend functions are decided by the interface programmer, the user can continue to exercise his choice of issuing input events to invoke the generic commands. The ability to define prefix-keys is actually an extension of the keymap idea.

7.2.2 Styles of interaction

An interaction style is the manner in which the user interacts with each of the interface objects, such as windows and menus. The interaction style involves input events, feedback, and is a dynamic entity as opposed to the static nature of interface objects. The user specifies and dynamically modifies his style of interaction at both the input and output phase. If the user wished to interact with all application programs uniformly, he specifies a single global default style and interacts with all application programs in a similar manner. He can also interact with certain application programs in a different style. We contend that giving the user the freedom to decide his interaction style is a sound decision on the part of the interface programmer.

Figure 5 depicts the interaction between the user and the window system via different interaction styles. The SOIs are the various styles of interaction and the resulting command is then passed to the application program (AP). Figure 6 shows how the user can interact with different application

programs via different styles of interaction. To ensure uniformity he can simply choose a single style of interaction and use that to interact with all application programs.

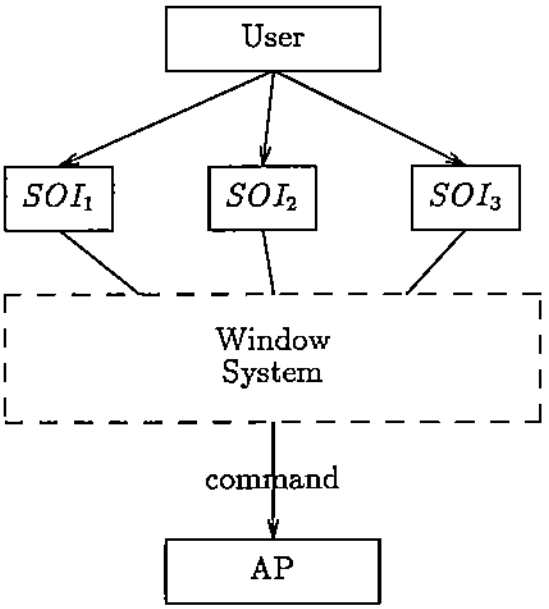

Figure 5: Styles of interaction between user and window system

The style of interaction should not force differences in issuing generic input events. Let us consider *generic-help*. If the user is interacting via the keyboard, he might construct the command name and then invoke the generic help function. He should be able to do the *same* even if commands are being issued via other input devices. Issuing the same generic input events regardless of the style of interaction is a test of uniformity. This uniformity is not obtained at the cost of customizability of the style of interaction. We demonstrate that uniformity and customizability are *not* mutually exclusive through our implementation of the model.

7.3 Extensibility

Extensibility has been defined as the ability to augment the list of available commands using the system primitives themselves. Teitelman [13] views extensibility as a means to accommodate a variety of programming styles.

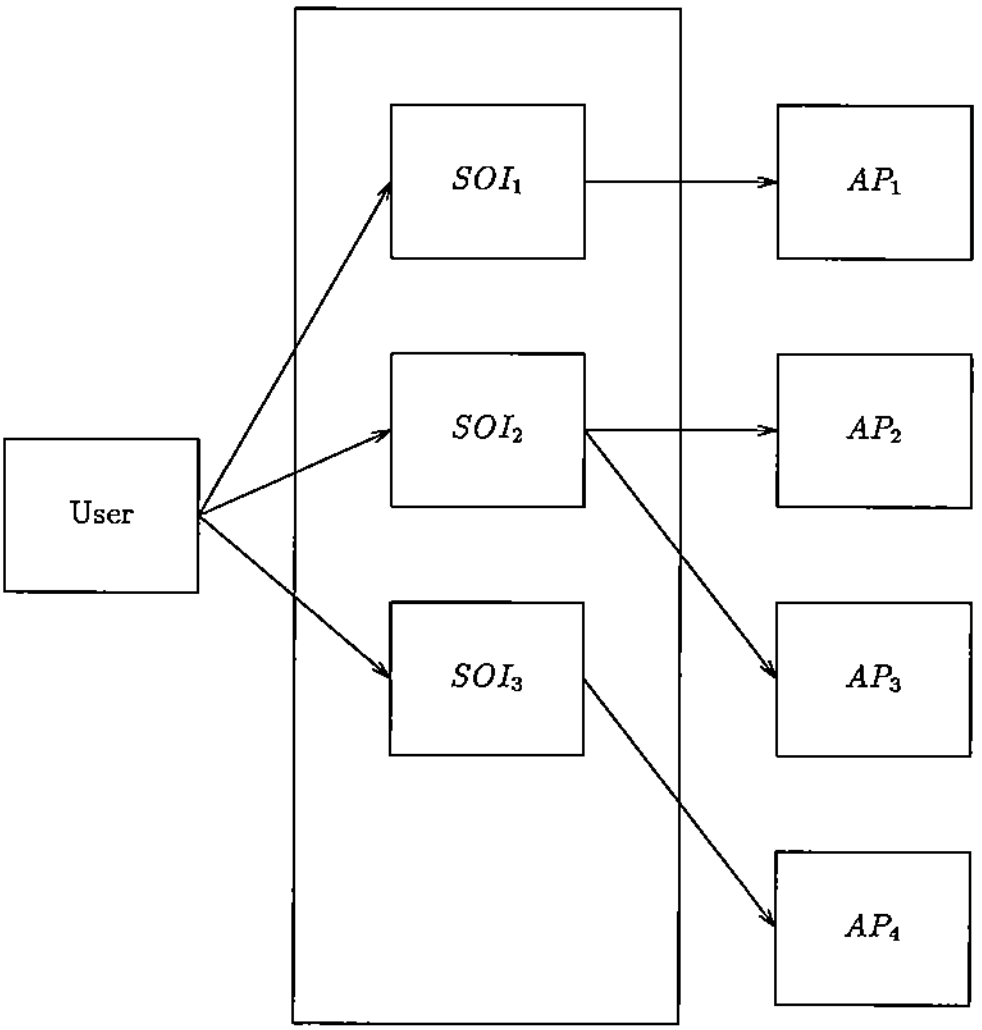

Figure 6: Styles of interaction with multiple application programs

Extensible editors provide primitives and language support to extend and modify the editor using the editor's own primitives. Emacs is an example of an extensible editor. As Stallman [11] says, users should not be limited by the decisions of the implementors—we extend this caveat to any interactive system, not just an editor.

The crucial elements that aid in extensibility are the hooks present in the underlying system. Hooks permit the user to call functions at specific occasions. These occasions are triggered by explicit user activity. For example, when the user switches his context, he may have some action performed. Upon entering a context that deals with electronic mail, the system may automatically see if there is any new mail for the user. The user should have the power to override any such default action with his own hooks. Further, wholesale changes in the interface can be made possible if the basic model enables replaceable parts.

7.4 Can the interface principles co-exist?

In the preceding sections, we have seen several interface principles including uniformity and customizability. In this section, we discuss their relative merits and see if it is possible and desirable for an interactive system to offer these features simultaneously. Two problems arise in this area. First, a uniform interface runs the risk of becoming rigid, as commands may be similar but lacking in flexibility. Second, a customizable system that can be modified readily degenerates and becomes impossible to be used by a different user, as the system has been modified significantly to meet the requirements of the individual user. Thus, it would not be possible for a user to walk up to the workstation of another user and use the same interactive system.

The first problem of a system becoming too rigid owing to uniformity is solved by ensuring that uniformity is provided at the right level. Generic commands provide the uniformity retaining the flexible nature of the interface. The users are free to rebind input events of their choice to the generic commands, without affecting the commands that get executed. The choice of what commands get executed has been made by the interface programmer. Generic commands also solve the second problem we raised—that of a readily modified system. The bindings of another user to the generic commands can be trivially altered, but as the bindings from generic commands

to actual functions have not changed, the user will not notice a difference. Regardless of how the generic commands are issued, the same actual commands get executed. Uniformity can thus co-exist with customizability.

7.5 Window system/user vs. user/application interface

While we have advocated uniformity in the user's interface with the various application programs in the interactive system, we have not discussed the interface between the user and the window system vis-a-vis the interface between the user and the application programs. The user should be able interact with the window system in precisely the same manner he interacts with the various application programs via the window system. In particular, we want the user to be able to interact with the window system via various interaction styles.

We first examine the purposes behind the user's interaction with the window system. The window system is a vehicle for interacting with the various application programs. As our model of interactive system provides for several different interaction styles there has to be provision to switch between these styles. The user has to communicate with the window system to modify the interface features, such as input and output styles.

The user interacts with the window system at two levels. At the application program level, any changes he wishes to make in his style of interaction with that application program is done by interacting with the window system. He should also be able to modify the styles of interaction of *all* the application programs by issuing a single global change request. By providing a *separate* application program whose task is simply to interact with the user and gather the changes he wishes to make in his environment, the window system can ensure uniformity. As the implementation chapter will show, this is precisely what we do.

Our model could be considered uniform and complete only if the user is able to communicate with the window system in precisely the same manner in which he communicates with the various application programs. In particular, it should be possible for him to communicate with the window system in the same way he can communicate with the application programs. Further, as part of our customizability requirements, there should be no un-

changeable bindings in the window system, just as we require that there be none in the application programs. In particular, the user should be able to specify and change the input events used to interact with the window system. Customizability is not simply imposing one's style of interaction while interacting with an application program: it is also the ability to interact with the window system itself in one's choice of a style of interaction.

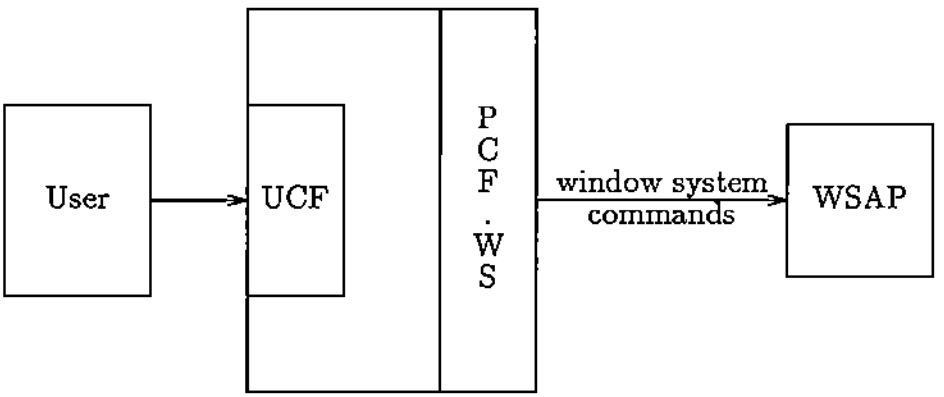

Figure 7: Interaction with the window system

Further, as a measure of completeness of our model, the interface programmer should not have to write any special code to make uniform interaction with the window system possible. In a complete model, the window system should be treated as yet another application program. Thus, if one were to write configuration files to enable interaction with the various application programs, it should be possible to construct a configuration file for interaction with the window system itself. The task of this application program would be co-ordinating global modifications to the interaction process. As shown in Figure 7, the interface programmer constructs a PCF for the window system specific commands (PCF.WS) and the user interacts with the window system application program (WSAP) just as he interacts with any other application program. Just as the power of a systems programming language is measured by writing a compiler for it in the same language, we should be able to get a measure of the power of our model of a window system by making it generate parts of itself *and* use the window system to modify itself.

An implementation of a window system based on the model presented above is described in [7].

8 Conclusion

In this paper we discussed the abstractions in partitioning the interface between the user and application program. We showed how factors such as uniformity and customizability can co-exist. Our aim was to reduce the work of the interface programmer by providing a scheme to encode interaction details enabling generation of a uniform and customizable interface. Further the firewall between the application programs and the user help move low level unchangeable bindings from application programs to the window system where it can be changed. The user can impose his style of interaction on all application programs via the user configuration file.

References

[1] D. Comer. *Operating System Design, Vol. 2: Internetworking with Xinu*, chapter 15: A Syntactic Namespace, pages 239–261. Prentice-Hall Inc., 1987.

[2] J. Coutaz. *A Paradigm for User Interface Architecture.* Technical Report CMU-CS-84-124, Carnegie-Mellon University, 1984.

[3] L. P. Deutsch and E. A. Taft. *Requirements for an Experimental Programming Environment.* Technical Report CSL-80-10, Xerox Palo Alto Research Center, June 1980.

[4] A. Goldberg. The influence of an object-oriented language on the programming environment. In *Proceedings of the 1983 ACM Computer Science Conference*, pages 35–54, February 1983.

[5] K. Hemenway and H. Armitage. Proposed syntax standard for UNIX system commands. In *Uniforum Conference Proceedings*, January 1984.

[6] W. N. Joy. *Introduction to the C-Shell, 4.2 BSD Reference Manual.* University of California, Berkeley, August 1983.

[7] B. Krishnamurthy. *Partitioning the Process of Interaction: An Implementation.* Technical Report CSD TR 706, Department of Computer Sciences, Purdue University, September 1987.

[8] R. Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, March 1982. Available as Technical Report CMU-CS-82-113.

[9] T. Moran. An applied psychology of the user. *Computing Surveys*, 13(1), March 1981.

[10] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the star user interface. *Byte*, 7(4):242–282, April 1982.

[11] R. M. Stallman. *Interactive Programming Environments*, chapter 15: Emacs: The Extensible, Customizable Self-Documenting Display Editor, pages 300–326. McGraw-Hill Book Company, 1984.

[12] Sun Microsystems Incorporated. Sunview programmer's guide. February 1986.

[13] W. Teitelman and L. Masinter. The interlisp programming environment. *Computer*, 14(4):25–34, April 1981.

[14] A. Wegmann. Vitrail: a window manager for an office automation system. *ACM SIGOA Bulletin*, 5(1–2):1–12, June 1984.