STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

AN INTRODUCTION TO THE FRONTEND

Donald I. Andrews
Beverly R. Boli
Andrew A. Poggio

January 5, 1977

ARC Catalogue Number 28743

An Introduction to the NSW Frontend

PREFACE

This document introduces the NSW Frontend to users and tool-builders. It should be read before going on to A GUIDE TO THE COMMAND META LANGUAGE AND COMMAND LANGUAGE INTERPRETER [1] and the FRONTEND SYSTEM DOCUMENTATION [2]. A brief overview of the NSW is provided, followed by a discussion of the Frontend design and system components, features to be added to the Frontend in the future, Frontend interfaces, and tool characteristics.

NSW OVERVIEW

ARPANET technology is now well established. Recently it has moved from the development to the production phase. Sixty-four IMPs with 120 host computers are currently connected to the network.

Although the physical resources of the network are on a sound footing, the communication protocols that use these resources are still at a rudimentary level. Only TELNET and FTP are widely used. These primitive protocols impose little semantics on their messages. When a user connects to a host via TELNET, he must have an account on the host, know the host's login and operating system conventions, know which programs exist on the host and how to run them, know the communication conventions of each program, know how files and directories are described, and so on.

The user must also know the TELNET and FTP conventions to get to the host, to switch between hosts, and to leave the host. If he goes to a different host, he must know these same things for the new host, and for every other host to which he wishes access.

The NSW was designed to alleviate this situation by creating a framework for sharing computer resources based on a communication network and improved management control of resources. The NSW goal is a network-wide "virtual" operating system spanning multiple hosts.

Rather than talking to individual hosts through TELNET, the NSW user communicates with a single program, the NSW operating system. This system communicates with the different hosts, thus hiding their idiosynracies from the user. In fact, the user communicates only with a single component of the NSW, the Frontend. The primary virtue of the Frontend is that it imposes order on the plethora of host-specific and program-specific protocols that exist in the ARPANET world. It uses a single, uniform, powerful protocol for all communication.

A brief description of the Frontend and other NSW components is provided below. This is followed by an extensive discussion of the design and development of the Frontend.

NSW Components

NSW software is comprised of the following components:

1) The Frontend system that provides terminal access to the ARPANET, a set of services creating a coherent NSW user

environment, and an environment to decrease the cost of new tool creation.

2) The Works Manager that supplies special services such as resource control, authentication, record keeping, file system and file transfer, and management aids.

3) Protocols and conventions needed between the Frontend and Works Manager, Frontend and tools, and Works Manager and tools.

4) The Foreman that furnishes tools with a uniform operating environment and access to NSW resources.

5) The tools (computers and software) that reside in the NSW.

The individual components of NSW software are viewed as independent, concurrent processes, even if they reside on the same host.

The NSW system provides the user with access to a number of general or specialized tools in such a way that the communication conventions he uses remain constant, even though the particular vocabulary changes from tool to tool as appropriate to describe that tool's functions. This user interface resides not only on a PDP-1Ø but also on a dedicated Frontend computer (PDP-11) for better response and less expense.

We anticipate that heavily used tools or commands will, in time, actually be executed on the dedicated Frontend computer. In addition to increased system responsiveness, this will reduce network communication and will protect users from network or large-computer unavailability.

Works Manager

The Works Manager is fundamentally a purveyor and allocator of resources. It maintains a list of the rights, privileges, and responsibilities of the users known to the system, and a list of the available tools. This information allows the Works Manager to control access to the tools. The Works Manager also maintains the file system catalog to control the existence of copies of files on different hosts in the network.

Protocols (MSG)

The standard communication between two NSW processes will be by messages, expressed in an 8 bit message format, and dispatched through a message-handler named MSG.

A process sends a message requesting a service and later receives a message in response. Two NSW processes may establish a direct network connection, if desirable. But the initial communication between them and the agreement to set up the direct connection are accomplished via MSG.

In the ARPANET implementation, the several MSGs are privileged processes on their hosts, with exclusive rights to reserved network sockets. This permits the NSW to bypass a local host's login procedure when executing a process on that host.

Foreman

The Foreman is the local-to-the-tool component of the NSW. Each instance of a tool has a Foreman responsible for the smooth operation of the tool with other NSW components. Along with the Frontend and Works Manager, the Foreman provides the tool instance with its NSW execution environment.

The Foreman has at its disposal empty file directories for tools running on that host. When the Works Manager wants to run a tool on a particular host, it sends a message to the Foreman asking it to load and start up an instance of the tool process. The tool then runs out of (or "in") a local directory assigned to it.

To open an input file the tool passes the NSW Filename to the Foreman, requesting the file. The Foreman then sends a message to the Works Manager, requesting that a copy of the file be sent to the local directory assigned to the tool. When the copy has arrived in that directory, the Foreman returns to the tool the local directory name of the copy, which it will have "opened" for the tool in the local file system. A similar process closes an output file.

Tools

Initially most NSW tools will be preexisting tools that were designed outside the NSW framework. Some preexisting tools

(such as NLS) and many new tools will be configured as "split" tools--that is, divided into command execution functions in one package (the Backend), and user interface functions such as command recognition, prompting, and feedback in another (the Frontend).

The Frontend (FE) and Its Role in the NSW Environment

The Frontend provides the interface between the NSW user and other NSW components, primarily the Works Manager and Tools. The user will always talk to the Frontend, which runs on a machine as close to the user as possible. The FE gives user input, in some form, to the appropriate component and puts user-destined output where the user can see it.

The NSW Frontend is divided into three logical modules, each of which is responsible for specified tasks or areas of operation. In the current implementation the Frontend consists of an interprocess communication module, a terminal handler, and the Command Language Interpreter (CLI).

The terminal handler is responsible for managing the terminal. It knows what kind of terminal is being used and any special characteristics about the terminal, and handles terminal input and output. It includes window and text manipulation primitives for display terminals. Other Frontend modules operate exactly the same way regardless of the terminal type.

The CLI is a software package that carries on a dialog with the user, helping him specify commands in a command language. The grammar, or command language specification, and the CLI form a general production system: a wide variety of command languages can be implemented including pseudo-English. The grammar is a data base from which the CLI operates. In addition to commands, it contains instructions to collect parameters from the user and to invoke functions in a tool or tools. The CLI can quickly switch from one grammar to another.

The tool builder writes his command language in the Comand Meta Language (CML), a high level command language specification language. He specifies exactly what interactions take place between the human user and the CLI. That specification is 'compiled' to form the grammar data base which the CLI uses as described above.

The user communicates with the Works Manager by way of a CLI running the NSW-EXEC grammar: the CLI/NSW-EXEC collects commands from the user and makes service requests on the Works Manager. From the Frontend point of view, the most significant request he may make is to run a tool.

When a user runs a tool, the Frontend provides the user access to the tool. How this is done depends on the tool. To run a split tool, the CLI obtains the grammar from the Works Manager, a Foreman/tool instance is created, and the CLI/tool-grammar communicate with the Foreman/tool via MSG. There are two ways to run an unsplit tool: the CLI can obtain a "transparent grammar", which simply passes the characters typed by the user to the unsplit tool, or it can obtain a tool-specific grammar, which would provide an improved user interface.

DESIGN FACTORS AND REQUIREMENTS FOR THE FRONTEND

The Frontend is designed to meet user interface, efficiency, and system requirements. The major factors considered are listed here, with a discussion of each one below.

Fast command specification response for the user.

Efficient transmission.

Uniform user interface and effective help facilities.

A central "coordination" function allowing user control of the user interface.

A separate command language.

Typewriter terminal and two-dimensional display terminal interface.

An environment to encourage experimentation with new tools.

User-level programming.

Fast Response. Compute-bound programs and interactive programs usually work against one another in a time-sharing system. Normally the first users to complain about response are experienced users of interactive programs. The command recognition task involves frequent process activation and terminal handling that are expensive in large scale time-sharing systems, but it does not require high power computing. Thus it makes sense to separate the command recognition and terminal handling functions, interactive in nature, from the command execution functions, slightly to very compute-bound in nature.

The Frontend is designed to give the best possible command specification response to the user by breaking the command recognition functions away from the tool. This makes it possible to move the CLI function closer to the user. The CLI can be on a dedicated host, a minicomputer, or ultimately a one-user mini/micro at or within the user's terminal.

Efficient Transmission. Providing the user with a highly interactive, well engineered user interface may conflict with minimizing transmission costs and process activation costs. Some systems provide interactive interfaces but transmit on

each character (resulting in one or two characters per packet on the ARPANET), while others provide no feedback to the user until he types "end-of-line" or "transmit" (i.e., a line-at-a-time system).

The Frontend seeks to resolve the dichotomy by interacting with the user until a complete and syntactically correct command is specified. Then the command specification is sent to the tool. Of course this does not reduce network transmission if the Frontend is just as far from the terminal as the tool, but the Frontend has been designed for the purpose of placing it as close to the terminal as possible.

Maximum effectiveness is obtained when the Frontend resides on a dedicated minicomputer hardwired to the terminal, or within the terminal. In a dedicated system or non-swapping time sharing system, the expense of character-at-a-time activations is minimized. Small computers are less expensive per computing operation for the kind of operations performed by the CLI.

The command-at-a-time nature of such a configuration makes excellent use of the transmission medium--even better than the line-at-a-time terminal--because commands often are longer than one line. Furthermore, the commands are syntactically correct when transmitted; if the user backspaces within a command specification or aborts a command, no transmission takes place. The dedicated character-at-a-time service that the user gets is far better than character-at-a-time response obtained from a large-scale time sharing system.

Uniform User Interface. In an environment such as the NSW there may be many tools available to any one user. The user should be able to learn to use many NSW tools with a minimum of study or instruction. A good way to do this is to make the commands for all NSW tools have as much in common as possible and to provide effective help features.

The Frontend provides every command language with a set of control functions common in many tools, such as command abort, command accept, and backspace character. The Frontend makes these functions uniform over all NSW tools that are "split" into separate Frontend and Backend modules. In addition, an individual user can specify what his own set of control function keys are to be. The Frontend also makes additional functions available to the user that may not be common in all tools, such as repeat command and back up within a command.

The Frontend also provides the following Help features for all tools:

noise words--words in a command supplied by the grammar writer to help the user understand the purpose of the command and what input is expected of him.

prompts--symbols (frequently a word or abbreviation) printed by the system to indicate what command input is expected next from the user.

? facility--typing a ? at any point in a command shows the command alternatives available to the user at that point.

echoing--a system response to something the user types. Echoing is presented just as it is designed for the tool; the Frontend does not modify it.

In addition, by typing a HELP function key the Help tool is invoked. This presents the user with English prose descriptions of the command he is specifying or any other command or term relevant to the tool he is using (provided, of course, the term has been defined in the tool description file accessed by the Help tool).

All types of help are available in all "split" tools and the user can always get help in exactly the same way in all grammars.

Central Coordination. Imagine an NSW user working with several tools, and consider some services that could be performed for him, over all tools, if there were a central coordination point to do them. The user could specify which keys performed which command language control functions; he could specify how much command feedback and command prompting he was to see; he could define and start up Command Sequences (umbrella commands that automatically invoke several other commands); statistics on his command usage and his errors could be compiled; he could construct "meta" tools, multi-function tools that call upon several tools to perform them; and the user could actually have several tools working for him at once, though he would only talk to one at a time.

The Frontend is such a central point. The Frontend is designed to be with the user at every point. It knows who he is and what tool(s) he is running. It will adjust certain user interaction characteristics to suit each user by consulting the User Profile, a data structure containing information such as a

user's control function set, and feedback and prompting parameters. The Frontend is designed to permit recognition and execution of Command Sequences, and to gather statistics.

This central Frontend function also makes other enhancements possible. The user can run several tools or single grammars that call on many tools to present a unified user environment. He can also specify that he wants to perform some task and have that automatically translated into the necessary commands and executed for him (e.g., "run this program" given a source file).

Separate Command Language. When a new tool is developed it is not always clear what is the best command language. Command language design problems are frequently not apparent until the command language is used, sometimes by many individuals. Obviously it is advantageous to have the command language somewhat separate from the rest of the tool, and easily changed. Command languages could then be tailored for different communities of users; that is, functions that are really the same could be named or controlled differently by users in different communities.

The command languages that drive the Frontend are written in the Command Meta Language (CML), a high level and rather simple language used to produce command languages. New grammars are easily made by editing the CML specification and compiling it. Running a tool with a new user interface amounts to using an experimental grammar with the Frontend. A grammar can be compiled and tested in a few minutes, without the user leaving his terminal or getting a special Frontend. Using the CML allows a consistent user command language interface across tools.

Terminal Interface. Most programs work for many kinds of terminals because they treat each terminal as though it were a typewriter, expecting the operating system closest to the terminal to do any special handling for the terminal (e.g., padding). Potentially, displays offer much more to the user than typewriters, especially when they have pointing devices, because they can be programmed to operate in a two dimensional way. Yet there is no uniform set of display manipulation commands, so that any programs that make use of them as two-dimensional displays are usually tied to a particular brand.

The Frontend provides the tool with a set of primitives for

writing on a typewriter terminal. It also contains a two-dimensional display package that provides the tool with primitives for finding out about the terminal characteristics (e.g., typewriter or display, screen size), writing on a display screen, defining windows (sections of the display screen), and manipulating text on the screen, independent of the display terminal brand. The display package is designed to interface to a wide variety of alphanumeric displays including SRI Line Processors (which have a pointing device). A tool's grammar can test Booleans to find out if the terminal is a typewriter or a display, so that the grammar can actually be different for displays (e.g., the user may be able to point to something on the screen; the tool would get the "address" of the item he pointed to in the command specification).

Innovative Environment. One of the original NSW goals is to create a marketplace for tools. It is going to be a busier marketplace if it is relatively easy to introduce, change, experiment with, and verify tools.

One of the Frontend goals is to make it less expensive to develop an NSW tool. The tool builder need not build a user interface. He need only write a grammar and a set of execution functions.

User-Level Programming. Frequently users of interactive systems find themselves repeating the same command or series of commands over and over, for example:

Compile a program
Load the program
Run the program.

Many systems combine the most common series into a single command, but no system can anticipate all of the possible combinations, any more than a programming language can anticipate all of the possible programs. Therefore, the Frontend is designed to provide a Command Sequence facility, a way for the user to collect a group of commands into a single executable unit. We call Command Sequences "user-level programs" because they are written in the languages (commands) the user normally uses. Command Sequences are decribed more fully below in "Future Additions to the Frontend."

Frontend System Components

Although the Frontend is responsible for presenting a consistent and responsive user interface for all tools, the user is not conscious that he is going through the Frontend. He seems to be talking directly to the tool. In fact, of course, the Frontend is providing him with many services, some of which he uses at all times (such as the Command Language Interpreter) and others he may request only when necessary (such as the Help tool). The various components that make up the Frontend, listed briefly here and described more fully below, may reside on one machine or many, although they are most effective on machines near the user.

(1) A formal language, the Command Meta Language (CML), for specifying NSW user interfaces.

(2) A compiler for that formal language that runs under TENEX as a subsystem or from NLS and produces tool grammar data structures.

(3) Tool grammars, products of the CML compiler or any other such program.

(4) A Command Language Interpreter (CLI) that processes a tool grammar and interacts with the user in specifying syntactically correct commands to the NSW.

(5) A User Profile facility consisting of a tool and data base that allow the Frontend interactions to be tailored to the individual preferences of the users.

(6) A semantic Help tool, which is employed by the Frontend when the user requests help with a tool or a command. It is presumed that each tool, in addition to supplying the Frontend with a grammar, will also supply it with the name of a Help data base whose structure and content are the responsibility of the tool builder/supplier.

The Command Meta Language

The Command Meta Language (CML) is a language for describing the command syntax of the user interface to application programs. CML syntax relies on two simple concepts--alternation (denoted by /) and succession (indicated by juxtaposing elements). For example, a command with branching choices in it may be represented by a command

word, followed by several command words separated by a /. Further interactions are specified by following elements. CML semantics are composed of built-in functions, semantic conventions, and parse functions.

The CML produces the full syntactic description of a command. The Backend uses execution functions to perform the low-level semantics of the command. In other words, the CML describes how the command "looks" to the user, rather than what it does inside the tool.

The CML program defining the user interface for a tool is compiled by the CML compiler to produce object code (called a grammar) which is interpreted by a Command Language Interpreter (CLI). The Command Language Interpreter is cognizant of the device dependent feedback and addressing characteristics of the user's terminal and it manipulates the terminal, according to the grammar, in a way most suitable for that device. The grammar author need not know the type of terminal that will be used.

The Compiler

CML programs are compiled to produce a binary data base (a grammar) that drives the CLI. The CML compiler is a TENEX subsystem that accepts ASCII text files as input and produces TENEX compatible "REL" or relocatable binary files as output. The compiler also runs under the NLS system to compile NLS structured files.

Grammars

A grammar, or compiled CML program, describes the command words, noise words, selection requests, and so forth, that make up the user interface to a tool or subsystem. It calls on Backend procedures that execute the actual command functions. The way the grammar interacts with the Command Language Interpreter is described below.

The Command Language Interpreter (CLI)--the CML "Machine"

The Frontend's command parsing capability could have been implemented in several ways; for example, table driven techniques or a subroutine per command approach could have been used. Instead, an interpreter and a set of instructions for an idealized machine were designed. The tool grammar

serves as the program and the Command Language Interpreter executes the instructions.

Conceptually, the grammar is a computer program which the CLI "executes". The instruction set for this "fictitious" computer (the CLI) has been carefully chosen to correspond to the CML primitives. The programs are organized in such a way that a minimum of program memory is required.

This approach has several advantages: it is general; the tool dependent part (grammar) is relatively small; and when found inadequate, the instruction set can be expanded. The generality and flexibility of the CLI are especially important design considerations.

A CML grammar is actually a pseudo-computer program that embodies the command language discipline for a tool. The CLI embodies the interaction discipline for all grammars. Thus, while commands (and arguments) may be quite different from one tool to another, the discipline for specifying, aborting, accepting, rejecting, and interrupting commands is always the same.

The CLI interacts with the user to help him specify commands for the system to execute. It prompts him for the type of input required (if the user wants it to), shows him the syntactic form of specific commands on request, shows him his actual alternatives at any point in the specification of a command on request, and can invoke the prose description Help facility.

The CLI provides a terminal-independent interface to tools. This means that even though the creators of a tool envisioned the user sitting at a typewriter terminal, the user who happens to be using a display terminal with a pointing device may be able to interact with the tool in a two dimensional sense, pointing to arguments on his screen instead of typing them. For tools that make more extensive use of a display terminal, the CLI presents primitives for allocating windows on the display and allows the tool to write, delete, or move items displayed within the windows. In addition, the creators of a display oriented tool can program in contingencies to handle a typewriter terminal, and even present different commands to the user where necessary.

The Help Tool

The Frontend helps the user specify commands by providing command word recognition, noise words, prompts, and a "next option" list. All of these are based on the grammar and contain no semantics beyond the meaning implicit in the command names and noise words.

If the above facilities are not sufficient because of uncertainty about a basic concept or vocabulary word, or the user wishes more information about the effects or use of a command, he can type the HELP key (or <CTRL-Q>) to use the Help facility. Help is implemented as a special tool rather than a Frontend function. When Help is invoked, information about the command being used is passed to the Help tool so it can take the user to an initial point that describes the command. If the user is not in the middle of specifying a command, the tool takes him to appropriate information about the system he is using, or else he may type in a term and the correct description will be furnished.

Once in the Help tool, a simple set of command conventions and the organization of the data base allow the user to easily reference related subjects, move to new subjects, or move up to higher level descriptions.

A structured Help data base, derived from the grammar syntax and text provided by the tool implementers, describes in English the intended use of the tool and its commands.

User Profile Facility

The User Profile facility consists of two parts. Together they give the user the ability to tailor the Frontend user interaction to suit his proficiency and/or preferences.

The Profile itself is a data structure loaded by the Frontend when the user is authenticated by the Works Manager. It is unique to each individual user, describing to the CLI how much prompting and feedback the user wants, the command word recognition scheme he wishes, and many other idiosyncratic features of the user interface.

The second part is the User Profile Tool, which allows the user to modify his profile data base and consequently the behavior of the system.

FUTURE ADDITIONS TO THE FRONTEND

Design and implementation of the features described in this section are in various stages of completion. Some features will be implemented in the short term; others are still being designed, with various possible design alternatives being considered. To simplify their descriptions, each feature is described in the present tense, except where design options are open.

Command Sequence Facility

A Command Sequence (CS) is a pre-defined operation using one or more different, but already existing, user-level commands. The user need not learn a new language to write a Command Sequence. It is written in textual form, exactly as the user would see the commands when actually executing them, and assigned a unique name. The Command Sequence can be given a command syntax and invoked by its name, just like any other command. The only difference is that its execution causes other pre-defined commands to be executed.

A Command Sequence may be defined with a command, a CS-defining tool, or a text editor. Using a text editor to put a Command Sequence into a file has the disadvantage that mistakes are easily made in writing down a series of command words, even for very familiar commands. A more promising approach is to enter the command at the terminal as usual, but have it placed into a file rather than executed. This method makes syntactic errors impossible. An option is to execute the command in addition to putting it into a file, to make sure it works.

To make a Command Sequence "active", the user specifies to the Frontend that he wants to use a particular Command Sequence or set of Command Sequences. He then types the Command Sequence name when he is at the top level command state in whatever (split) grammar he is using (e.g., the EXEC). The Command Sequence name is recognized as a command. Confirmation starts the Command Sequence execution.

Three Levels of Command Sequence Capability

The first level of Command Sequence capability simply allows the user to place a Command Sequence and text in a file and have it executed automatically from there; the system behaves exactly as if the user had specified commands and typed in text. A Command Sequence of arbitrary length can be placed in the file. The entire sequence or any designated part of

it can be executed. The sequence can be executed immediately, saved and executed later, executed more than once, or executed as part of the execution of another sequence (sub-routine).

The second level of capability allows the collection of parameters from the user in addition to specifying commands and textual input. The Command Sequence may prompt for and collect several parameters from the user for input to the Command Sequence commands. The parameters may be collected at the beginning of Command Sequence execution or at the point when they are needed.

Third level capability requires decision making on the part of the Command Sequence. The decision may be based on user input or command results. This makes it possible for Command Sequences to handle error conditions properly and to perform operations in a variety of ways, as directed by the user at Command Sequence execution time.

For the second and third level capabilities, the system will provide a "Control Language" which will contain an iteration command, a conditional command for testing the success or failure of other commands, a way to collect parameters from the user when a Command Sequence is executed, and other programming kinds of features.

The Command Sequence facility is a feature of the Frontend and is not tied to a particular grammar or tool. Hence the Command Sequence execution can invoke any tool (split or unsplit) and perform any command in the tool.

Multiple Tools

Because the Frontend process is not directly involved in the execution of tool commands, the user may run several tools concurrently. Typically, the user will interact with one tool for awhile, escape back to the EXEC and run another tool, then escape back again and resume the first, etc.

The display user can divide his screen into windows and run a different tool in each window. He may even be able to determine with which tool he wishes to interact by placing his cursor in the proper window.

Meta Tools

Occasionally a user may need more than one tool to accomplish his task. A specialized tool that incorporates commands from several tools meets this need. As the name implies, a meta tool grammar makes use of several tools to perform high level commands. The user does not necessarily know what tools are invoked--he thinks in terms of commands in the meta tool.

Although meta tools could be implemented as Command Sequences, there may be cases in which the user wants to include many commands, making it more efficient to make them grammars instead of Command Sequences. A meta tool also solves the problem of restricted tool (or file) access. When a user does not have access to a tool (or special file) but would like to use a few commands in that tool, he could be given access to the meta tool, which would only make limited use of the restricted tool.

Tool Interaction

The meta tool feature is a way to combine capabilities of several tools in ways predetermined by the meta tool commands. Tool interaction is a way to combine tool capabilities in unforeseen ways.

Tool interaction is based on the idea that every tool will support the concept of a "source address". This may be a line number in an editor, a message sequence in a mail system, or an NLS address. Every tool provides a way for the Frontend to request that it make an NSW file, given a source address.

Tool interaction is necessary whenever the user wants to supply something from one tool as input to another. Typically, the user runs both tools concurrently. He executes the command in the first tool, but as input to the command he gives the Frontend a source address and indicates that it be applied to the second tool. The source address is specified in terms proper for the second tool. The Frontend then asks the second tool to create a file containing the text addressed, using that as input to the first tool's command.

Terminal Linking

The Frontend permits one user to talk (i.e., type) to another by means of terminal linking. Linking terminals in NSW is

roughly parallel with the RSEXEC linking capability. One Frontend contacts another (after asking the Works Manager where it is) and sets up a two-way link, perhaps across hosts.

A more interesting and powerful capability is display screen linking. This allows two users at display terminals to share a screen while simultaneously carrying on a conversation by telephone or conventional linking.

This feature works across hosts, for users with different brands of display terminals and different screen sizes, and without cooperation from any tools.

User Statistics Collection

The Frontend is an ideal place to install measuring devices. Tool builders and managers often ask two common questions: (1) how are users using the tool, and (2) what level of service are they getting. One measurement package in the Frontend collects data to answer both of these questions for all split tools. Further, the data could be used by tutorial-type help systems to point out common errors, level of command usage by the user, etc.

The statistics package has two components. A measurement package in the Frontend collects data on command usage by user, user errors, and tool responsiveness. The result is a data file for each user session. An analysis tool summarizes the data in a variety of ways, as specified by the analysis tool user (e.g., tables and graphs of averages, distributions of various measurements). The analysis tool also provides an interface to user programs so that users, managers, and/or tool builders can write their own analysis programs.

TOOL CHARACTERISTICS

Each tool has two important characteristics for the Frontend: composition (split or unsplit) and Frontend communication mode. Basically, split tools use the Frontend for command recognition while execution functions are performed by the Backend; in unsplit tools Frontend and Backend functions are combined. But there is a spectrum of intermediate configurations.

(1) At the low end of the spectrum, unsplit character-at-a-time tools expect character-at-a-time interaction with the terminal, (i.e., the CLI is transparent). For line-at-a-time tools the CLI can be equally transparent and the transmision discipline more efficient, but the user interface capabilities will be limited.

(2) In a primitive split tool the grammar may send one character at a time to the tool--and essentially implement TELNET. (No improvement is made over 1.)

(3) The grammar may collect strings from the user (observing termination characters) and send them on to the tool. This implements local echoing/buffering and minimal editing, and gains transmission efficiency.

(4) Or the grammar may recognize commands and translate them into text strings as input for the tool--which will look like normal user input for that tool.

Any existing tool may be split as described in configurations 2 through 4, gaining definite benefits and requiring absolutely no changes to the tool (e.g., TECO, SOS, etc.). With slightly more effort the two-dimensional display facilities of the CLI Frontend can be utilized to give such a tool a two-dimensional interface, again without altering the tool itself.

(5) The other end of the spectrum includes split tools like NLS that contain only execution functions in the Backend, and interface with the grammar and the Frontend functional primitives. In addition to providing good FE service, such tools should be considerably less expensive to build than present unsplit tools.

An unsplit tool could be used as a split tool in schemes 2 through 4 above.

There are two possible modes of communication between the Frontend and a tool: character-oriented and message-oriented communication.

Character-oriented communication is implemented as a Telnet connection (obtained from MSG-3 or other means). Message-oriented communication is done via MSG-3 messages. Although it would be possible for message-oriented communication to be done over direct connections obtained from MSG-3, we have not seen a need for that. Existing unsplit tools and split configurations 2 through 4 above require character (Telnet) communication. More sophisticated split tools require message (MSG-3) communication.

FRONTEND INTERFACES

The Frontend interfaces to tools, the Works Manager, and the Foreman by means of interface modules within the Frontend. As explained above, there are two types of interface, character oriented (for unsplit tools) and message oriented (for split tools and the Works Manager).

Throughout the development of the Frontend it has been necessary to implement several interface modules, because the interprocess communication mechanism has completely changed from time to time. Some modules were also used for more expedient debugging.

The Frontend proper is completely independent of the interface modules--the same Frontend object file may be loaded with any of the interface modules to create a Frontend that uses the desired interprocess communication mechanism. Those interface modules are described briefly below. Any given Frontend will contain only one interface set (character- and message-oriented communication). The standard NSW Frontend will use the MSG-3 Telnet character interface and the MSG-3 message interface.

Character-Oriented Interfaces

In early NSW systems the TENEX Frontend obtained job-relative network sockets which were used by a User Telnet process within the Frontend. That User Telnet process then communicated with the encapsulated unsplit tool much as any User Telnet.

More recent versions obtain Telnet connections to the unsplit tool process (by process name) by way of the MSG-3 Telnet direct connection feature. The Frontend then communicates as above, with the added feature that the unsplit tool input may be taken from the keyboard or generated by the grammar. The tool output may also be captured and displayed to the user as desired. This provides for the implementation of local command recognition, as outlined in the previous section.

Message-Oriented Interfaces

Message-oriented communication has three aspects: communication medium, message semantics, and data structure within messages.

The message format for Frontend communication has been well defined. (See "How Tools Interface to the CLI" in A GUIDE TO THE CML AND CLI.) Elements of the message format determine the

semantics of the message, typically "procedure invocation" and "reply". By convention the data structure format is determined by the communication medium.

The data representations are listed here. Format definitions appear in Appendix 3 of A GUIDE TO THE CML AND CLI.

PCPB36

A 36-bit word data structure that contains integers, indicies, Booleans, character strings, and lists of the above.

PCPB8

Similar to PCPB36 but the entire structure is made up of 8-bit bytes--indicies are 2 bytes long, integers are 4 bytes, etc.

L1Ø

A 36-bit (or 16-bit) word oriented non-linear data structure paralleling PCPB36 and PCPB8 but containing memory addresses. This is simply an in-memory representation of PCPB36. It is more suitable for program manipulation, while PCPB36 and PCPB8 are more suitable for transmission.

CONCLUSION

The NSW Frontend system provides the kind of consistent, responsive user environment necessary for the maturation of a network marketplace. It also opens up other possibilities. The ease with which the user interface can be changed and user statistics gathered makes the system an excellent vehicle for human factors research. Automatic adaptation of both the grammar and User Profile data structures according to the user's proficiency with a tool is possible. The system also provides an environment for applying advanced technology, such as computer voice recognition, to change the physical nature of the man/machine interface.

REFERENCES

1. Andrews, Donald I., Beverly R. Boli, and Andrew A. Poggio. A Guide to the Command Meta Language and Command Language Interpreter. Augmentation Research Center, Stanford Research Institute, Menlo Park, California. February 3, 1977. (28744,).

2. Andrews, Donald I., Lawrence L. Garlick, and Andrew A. Poggio. Frontend System Documentation. Augmentation Research Center, Stanford Research Institute, Menlo Park, California. February 3, 1977. (28745,).