THE COMMAND META LANGUAGE SYSTEM

Charles H. Irby
Stanford Research Institute *
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200 Ext. 2013

ABSTRACT

A formal language system for the specification of control interfaces between human users and application programs is described. The formal specification is compiled and the resulting "program" is processed by a Command Language Interpreter to implement the human interface. The Command Language Interpreter communicates with the application program through various protocols which are briefly described. The Interpreter also makes use of a user-specific data base that allows the system to be tailored to individual preferences.

Keywords: User Interface, Human Interface, Formal Language, Command Interpreter, Computer Emulation, Command Language System, Frontend System, Network Access, Coherent Command Language.

INTRODUCTION AND MOTIVATION

To facilitate the easy formal description, implementation, and modification of the user interface to a range of interactive application programs, the Augmentation Research Center (ARC) at SRI has developed the Command Meta Language (or CML) System. This was an outgrowth of earlier efforts to accomplish the same goals at ARC [4][13][1][16][5]. This work was undertaken to support the NLS Knowledge Workshop tool system [2][3][5][12] and the National Software Works (or NSW) [15][20][21]. Both of these systems are accessed through the ARPA Network.

The intention of both the NSW system and the NLS system is to provide the user access to a number of general or specialized application programs (what we call *tools*) in such a way that the command discipline he uses remains constant even though the particular command vocabulary changes from tool to tool, as appropriate, to express that tool's functions (see 4 below). These tools may well be executing within different operating system environments on different computers in a computer network. Our many thousands of hours of terminal and network experience and user observation/analysis have shown us the importance of this coherence and consistency across an array of tools that make up a complete "workshop" system to the user.

Part of the desired coherence of these systems is achieved through a common file system which is not discussed here; the rest is achieved through a Command Language Interpreter that acts as a mediator between the user and the application programs. This Command Language Interpreter, along with the formal language system that supports it, are the subject of this paper.

In the NSW and future releases of NLS, this user interface system will reside not only in a PDP-10 but also in a dedicated PDP-11 Frontend computer for better responsiveness and decreased expense. We anticipate that heavily used tools or commands will, in time, actually be executed in the Frontend computer. In addition to increased system responsiveness, this will reduce network communication and will afford users a certain amount of insulation from network or large-computer unavailability.

The execution functions that implement the semantics of tool commands are invoked by the Command Language Interpreter through one of several communications protocols [6][7][8][14]. Thus, the execution functions may be written in any language that can be interfaced to such a protocol (see Function Interface Protocol Section below).

The goals of this development were to:

1) Provide a means for easily changing and experimenting with the user interface to an interactive application program.

2) Allow for the independent manipulation of

a) the commands available to the user in particular application programs and
b) the interactive procedures and techniques he uses to specify commands in all application programs.

3) Provide builders of new interactive application programs with a facility for easily creating the user interfaces for their new programs.

4) Provide the user with consistent and coherent command language features across a collection of independently developed application programs, or what might be termed "tools."

Regardless of the application program to which the user is giving commands, he can do so using the same procedures and techniques for specifying which commands he wishes executed and for specifying arguments or parameters to commands; he always receives the same type of prompting, and requests help in the same way. In addition, the general syntactic form(s) of commands would be the same from tool to tool unless there were good reason for the tool to deviate from the standard. Of course the particular commands and vocabularies will vary with the tool, and, in fact, the same verbs may be used with quite different meanings in different tools, but at least most other aspects of the command language (including asking for help and being prompted for the proper type of input) would stay the same across tool boundaries.

5) Provide application programs with well-formed commands.

Many operating systems and application programs have elected to use half duplex, line-at-a-time terminals because of the increased computer and communication efficiency provided by this approach. Other operating systems and application programs have chosen, instead, to utilize character-at-a-time full duplex terminal disciplines because of the opportunity this provides for utilizing a more human-engineered command language.

The CML system is an attempt to combine these two approaches into a COMMAND-AT-A-TIME system, where the application programs do not directly interact with the terminal, but rather receive fully specified commands from the Frontend. At the same time, the Command Language Interpreter in the Frontend will attempt to provide the user with the best possible human-engineered command language discipline. This will be particularly effective in situations where the Command Language Interpreter is executing on a dedicated computer with the user terminals directly attached.

6) Provide a terminal-independent interface to the application programs.

Because the Command Language Interpreter handles all terminal interaction, it presents to the application programs a small number of virtual terminal classes. Thus, once an application program is developed, little attention need be given to the type or particular characteristics of the terminal the end user may choose to employ while using it. In fact, the cost of creating new application programs should be considerably reduced because of these facilities.

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 by pointing to arguments on his screen instead of typing them.

For tools that wish to make more extensive use of a display terminal, if the user has one, the CML interpreter presents primitives for allocating windows on the display and allows the tool to write/delete/move/make-invisible items displayed within the windows [11].

7) Make asynchronous operation possible.

In some instances, it may be possible for the execution of the user's commands to be accomplished in parallel with subsequent command specification and execution. This frees the user to do other things while a lengthy command is being executed by a tool.

8) Provide standard mechanisms for presenting status or error conditions to the user.

9) Provide the user with enhanced, consistent "help" facilities while using any application program.

10) Allow for a common statistics gathering point for analysing user interaction characteristics such as error rates, frequency of issuing given commands or groupings of commands, and average user-observed execution times for commands.

11) Provide a convenient way of grouping the commands available to the user.

THE APPROACH CHOSEN

To accomplish the above objectives, a hypothetical computer was postulated that was capable of interacting with an external agent (presumably a human user) in the course of executing its instructions. A "program" for this computer formed a tree-like data structure, that we called a Grammar. Instructions for this computer had logical "successor" and "alternative" address fields. The successor address pointed to another instruction to be executed if this instruction successfully executes. The alternative address fields formed a chain of instructions that were processed in parallel. That is, a set of instructions were processed concurrently such that when any one of the instructions in the set of alternatives succeeded, then the Program Counter was advanced to its successor. Should several succeed at once, then the Program Counter was advanced to the successor of the first instruction that succeeded in the chain. That instruction and its alternatives were then processed. The computer also had a single accumulator, an argument stack for procedure calls, and fully typed data objects.

At any point in time, the computer is attempting (presumably by interacting with a human user) to find a path through the tree. An illustration might be helpful here. At a certain point in time the Program Counter might point to an instruction to recognize a command word, a "reserved" word in the command language. This instruction might have alternatives for recognizing other command words. These command words might represent the verbs of commands the user can give to a tool or might represent refinements to a command already partially specified. The computer picks a path through these alternative command words (although how this is accomplished is left purposely unspecified here). It is precisely the "HOW" of this computer's path finding that embodies the human-factors considerations and human-computer interaction disciplines, which can, and in our case do, vary from user to user. Thus, how the system interacts with the user is independent of the particular commands available to the user; one can be changed while the other remains constant.

Given the above model for a computer (which we termed the Command Language Computer or CLC), we then developed a formal language (CML) and compiler for this computer along with one of many possible implementations of an emulator for the CLC. We call the emulator the Command Language Interpreter (or CLI). These facilities are used to specify the user interface for the NLS Knowledge Workshop tool system [2][19] and tools within the NSW. The form of the Command Meta Language and the CLC is the chief topic of this paper.

Embodied in the CLI are the principles for human-computer interaction that have evolved through many years of use and evaluation of NLS and other systems [19]. It is the CLI that 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) and, if the user requests, can show him the syntactic form of specific commands, can show him his actual alternatives at any point in the specification of a command, and can invoke a semantic help facility. This semantic help is derived from a structured data base provided by the tool implementers (along with the CML user-interface description). The help data base attempts to describe in English the intended use of the various commands and the tool as a whole. This data base is structured to allow the user to get to the information he needs quickly without scanning through pages of written material.

Thus, a tool is now seen to consist of three parts:
1) an execution module that carries out the commands specifiable by the user,
2) a CML description of the user interface, and
3) a semantic help data base.

To allow the user interface to be individually tailored, we have added a data base called the "user profile" which describes to the Command Language Interpreter how much prompting and feedback the user wants, what recognition scheme he wishes to use to select among command-word alternatives, and several other idiosyncratic features of the user interface. There is a special set of commands for modifying this data base and consequently the behavior of the system.

To facilitate user analysis, a user-statistics data base could be added in which the Command Language Interpreter could record which commands were used, whether or not errors were made in the specification of the commands, the execution time of the command, and other statistics.

And finally, to achieve terminal independance, a terminal handler component was added. This was discussed in [11] and is not discussed further in this paper.

In summary then, the Frontend system consists of the following:

1) A formal language (CML) for specifying user interfaces.
2) A compiler for that formal language that runs under the TENEX operating system.
3) Tool Grammars, products of the CML compiler (or any other such program).

4) A Command Language Interpreter that processes a tool Grammar in order to work with the user in specifying syntactically correct commands to the tool.
5) A user-profile data base that is used by the CML interpreter while interacting with the user. This data base allows the Frontend to be tailored to the individual preferences of the user.
6) An optional user-statistics data base, where, if desired, statistics could be accumulated on commands used by a user, error rates, and the like.
7) Access to a semantic help facility that is employed by the Frontend when the user requests semantic level help with a tool or a command. This help facility could also be kept informed of the user's dialog with the Command Language Interpreter and could have access to the tool grammar, and the user's profile.
8) A virtual terminal interface.

The following sections describe in more detail the Command Meta Language, characteristics of the Command Language Computer, and the Command Language Interpreter.

THE COMMAND META LANGUAGE

The syntax for the CML is described through the Tree-Meta alternation (denoted by /) and succession (denoted by juxtaposition) concepts [9][10]. The semantics are introduced via built-in functions, semantic conventions, and parse functions.

No attempt is made in CML to allow for the full semantic description of any command, but it is hoped that the Frontend interface (parsing and feedback operations) may be explicitly accommodated with these facilities. It is still necessary, and desirable, to use execution functions to perform the low-level semantics of the command. We call the collection of these execution functions and their support routines and data structures the tool "Backend." The Backend is viewed as a separate computational entity, perhaps executing on a separate computer in a computer network. The CML describes how the command "looks" to the user, rather than what it does inside the tool.

The CML supports zero look-ahead, phrase structured, context free command languages.

SYNTAX NOTES

The following meta symbols are used in this discussion of the CML:

.ID	An Identifier.
.SR	A quoted string.
/	Denotes alternatives. A/B means A or B.
%	Brackets comments.
()	Used for grouping to control precedence.
[]	Used to denote optional elements.
'	Precedes literal characters.
"	Encloses literal strings.
#X	At least one occurrence of whatever X is.
#<Y>X	At least one occurrence of whatever X is, separated by whatever Y is.
$X	Zero or more occurrences of whatever X is.
$<Y>X	Zero or more occurrences of whatever X is, separated by whatever Y is.
.NUMBER	a string of digits representing a non-negative (decimal or octal) integer.

PROGRAM STRUCTURE

The basic compilation structure of a CML program is described by:

```
file = "FILE" .ID $dcls $rule #subsys "FINISH";
```

The "file" construct brackets the definition of control language subsystems. Declarations of variables, execution, and parse functions may be made at this level. In addition, global parsing rules may appear here and be invoked in commands by simply specifying their names.

```
    subsys =
        "SUBSYSTEM" .ID % subsystem handle %
        "KEYWORD" .SR  % recognition string %
        #(command / rule)
        "END.";
```

The "subsystem" construct brackets a set of rules or commands (generally a set of related commands that the tool implementer wants to cluster together). Rules containing the reserved word COMMAND are linked together to form a command language subsystem or tool.

```
    command = (cmdrule/
              ("COMMAND" / "INITIALIZATION" /
              "TERMINATION" / "REENTRY")
                  rule);

    rule      = .ID '= exp '; ;

    cmdrule  = .ID "COMMAND" '= exp '; ;  %available as a convenience%
```

The tool may include a rule preceded by the reserved INITIALIZATION or TERMINATION. If specified, these rules will be executed once upon tool initialization/termination, respectively. This enables, for example, a tool to open and initialize a work file when it is started and to close it after the user's last command has been issued.

The subsystem may include a rule preceded by the reserved word REENTRY which will be executed upon reentry in the subsystem after executing commands in other subsystems. Note that a similar EXIT rule could also be added. This rule would be executed when the user temporarily suspends his dialog with a tool in order to give commands to another tool.

The Command Language Interpreter allows the user to freely move among subsystems. Thus, the user may give commands to one subsystem for a while, then give commands to another, and finally return to the first. The REENTRY rule will be executed when the user resumes giving commands to the first subsystem. This might be necessary, for example, to ensure that work files or data structures are still in a valid form.

Each rule/command is named with an identifier. This name may be used as a term in any other rule, indicating that the named rule is to be invoked at that point in the parse.

DECLARATIONS

Declarations are used to associate attributes and values with identifier names that are used in CML programs. If not declared, identifiers are defined by their first occurrence according to the following rules.

1) Identifiers appearing on the left hand side of an assignment statement are defined as "VARIABLES."
2) Identifiers followed by a subscripted list are assumed to be of type "FUNCTION."
3) All other undefined identifiers are assumed to be names of parse rules or commands.

The syntax of the declare statement is given by:

```
    dcls =
        ("CONSTANT" #<',> (.ID '= .NUMBER)/
        "DECLARE"
            ( ["VARIABLE"  /  "PARSEFUNCTION"] #<',> .ID /
            "FUNCTION" "PROCESS" '= .SR ', "PACKAGE" '= .SR ':
                #<',> (.ID ["OUT" "OF" "LINE"] ["PSEUDONYM" '= .ID]) /
            "COMMAND WORD" #<',> (.SR = .NUMBER [selector])));

    selector  = "SELECTOR"
        ('= builtin /
        #("TYPEIN"/"ADDRESS"/"POINT") '=
           (builtin/.ID %name of a selection function%));

    builtin  =
            ("CHARACTER"/"WORD"/"TEXT"/"INTEGER"/"NUMBER"/
            "FILENAME"/"NEWFILENAME"/"OLDFILENAME"/"VISIBLE"/
            "INVISIBLE"/"STRING"/"WINDOW"/"OCTALINTEGER");
```

If a declare attribute (VARIABLE or PARSEFUNCTION) is not given, type VARIABLE is assumed.

Semantics of the declare attributes are:

VARIABLE:
: a cell that holds pointers to fully typed CML data structures.

FUNCTION:
: arbitrary remote processing function in a tool Backend usually invoked to carry out all or part of the execution of a command.

COMMAND WORD:
: precedes a list of command-word strings (#<',>.SR) and indicates that the named command words are to have the specified integer tokens and that optionally they may also be used to collect arguments from the user (act as a selector). This may be in terms of built-in selection facilities or may indicate that user-supplied selection functions are to be called to effect the selection. These selection functions interact with the user and are similar to parse functions.

PARSEFUNCTION:
: a function which is used to extend CML. Such a function may interact with the user and may effect internal state information.

INSTRUCTION-PRODUCING CONSTRUCTS

Command-Word Recognition

The process of command-word recognition is independent of the description of the command words for CML. In the CML description, each command word is represented by the full text of the command word. The algorithm used to match a user's typed input against any list of alternative command words is known as command-word recognition, and is a function of the Command Language Interpreter.

Command words are written in the meta language as upper-case identifiers enclosed in double quote marks optionally followed by a set of command-word qualifiers.

```
    command-word     = .SR [ '! #qualifier '! ]

    qualifier        = "L2" / '.NUMBER;
```

If the user has specified that he wants some (supposedly frequently used) command words recognized based on their first letter and the rest only after typing an escape character, the CLI attempts to accommodate him. The tool implementer has control over which command words will be available to such a user via first letter recognition by assigning the L2 qualifier to command words which are not to be so recognized. The L2 denotes that it is a "second level" command word and of less probable interest to the user than its alternatives.

If it is desirable to change the integer token associated with a command word for particular uses of that command word, then the qualifier can be the desired new integer token.

Collecting arguments

Three types of argument selections are built into CML. They are Literal-typein Selection (LSEL), Destination Selection (DSEL), and Source Selection (SSEL).

The Literal-typein Selection is used to collect literal typein from the user, although it also allows him to point to text on his display instead of typing it.

A Destination Selection is used to allow the user to select one of several items the tool has presented to him. This can be done by pointing to it, using a pointing device, at a display terminal or by typing characters that the tool will interpret. For example, a tool may manipulate textual or graphical representations of data stored in a file. The tool might have a delete command and would use a Destination Selection to allow the user to specify the line in the drawing or the word in the text to delete. Thus, when the tool puts the display image on the screen, it does so using primitives in the Frontend that supply identifiers for elements of the display [11]. When the user points to an object on the screen the identifier for it is returned to the tool.

A Source Selection is similar to a Destination Selection but also allows the user to supply the argument as a literal typein.

These recognizers require some entity type as an argument (in the accumulator) and return (in the accumulator) a data structure which represents the selection.

The DSEL, SSEL, and LSEL functions perform all evaluation and feedback operations associated with the selection operations. The tool implementer may define new types of selections and define the data structure that is built as a result of the selection.

```
selection = ("SSEL"/ "DSEL"/ "LSEL") '( parameter ');
```

Command Confirmation

The process of command confirmation is represented in CML by a built-in parameterless function.

```
confirm  = "CONFIRM";
```

Yes/No Question Answering

The process of collecting a simple yes/no question answer from the user is represented in CML by a built-in parameterless function.

```
answer   = "ANSWER";
```

Function Execution

Functions may be invoked at any point in the parse by writing the name of some routine and enclosing a parameter list in parentheses. Remote functions are accessed through a procedure-call oriented protocol. Parse functions must be written in the source language of the CLI and must obey conventions established by the CLI. A discussion of these conventions is outside the scope of this paper. The actual arguments to remote procedures are passed by value and are converted to a protocol-standard representation (see Function Interface Protocol Section). For parse functions, parameters are passed by address.

When a remote call is made, a qualifier in square brackets [] may be used to indicate the name of a rule or to modify the declared IN LINE/OUT OF LINE attribute of the procedure. If a rule is named for an inline call it is assumed to be a help rule and will be processed if the called procedure requests it. If the call is out of line, however, the specified rule is processed immediately after issuing the call. Results from the called procedure can be assigned to variables through the use of the "->" construct. Such results are, in any case, always stored in RESULT1 through RESULT8. However, these built-in variables are set to NULL at the end of each command and after issuing each remote call.

```
control   = .ID % routine name % [call-qualifier]
                '( $<',> parameter ["->" #<',> .ID ] ');

call-qualifier = '[ .ID / "IN" "LINE" / "OUT" "OF" "LINE" '];

parameter = factor / tparameter;
```

```
tparameter =
     "VALUEOF" '( .SR ')      % command-word value %
   / '# .SR                   % same as VALUEOF %
   / "TRUE"                   % Boolean TRUE value (one) %
   / "FALSE"                  % Boolean FALSE value (zero) %
   / .NUMBER                  % arbitrary non-negative integer%
   / "TYPEWRITER"             % Boolean TRUE if user has a
                              typewriter terminal%
   / "DISPLAY"                % Boolean TRUE if user has a display%
   / "LINEATATIME"            % Boolean TRUE if user has a
                              line-at-a-time terminal%
   / "HALFDUPLEX"                    % Boolean TRUE if user has a
                              half-duplex terminal%
   / "BREAKPOINT"                    % Boolean TRUE if tool is at a break
                              point%
   / "HELPCODE"               % contains NULL or an integer help
                              code (see below)%
   / "RESULT1" ... /"RESULT8" % arbitrary results
                              from remote call%
   / "RESULT"                 %= RESULT1; arbitray result
                              from remote call%
   / "NODE"                   % logged in user name%
   / "PROJECT"                % logged in user account%
   / "CURRENTTOOL"            % name of tool to which user is
                              currently giving commands%
   / "ACTIVETOOLS"            % list of active tools (being run
                              concurrently%
   / "WINDOW"                 % identifier of current window
                              containing cursor %
   / "NULL";                  % null pointer value (zero) %
```

Feedback Control

The feedback control elements of CML are used to provide feedback in addition to the normal feedback generated by the recognizers. This is used to implement additional "noise words" and help feedback by:

1) adding feedback to the command feedback. A string may be added to the current command feedback by enclosing the quoted string in angle brackets.

extra-feedback = '< .SR '>;

2) replacing the last string in the command feedback. If the user's terminal allows, it is possible to replace the last string in the command feedback line by using the string replace facility. This is similar to (1) above except the previous string in the command feedback is deleted before adding the new string.

replace-extra-feedback = '<"..." .SR '>;

A function is also provided to initialize the command feedback mechanisms and clear the command-feedback area.

clear-feedback = "CLEAR";

Expression Definition

CML is an expression languge. Commands are defined as single expressions, and expressions are composed of successive/alternative expression factors. Alternative paths are indicated by the character '/ in the expression.

The nesting of expressions may be explicitly defined with parentheses, and brackets are used to delimit optional expression elements.

```
exp = #<'/>alternative;

alternative = #factor;

factor = terminal / '[ exp '] / '( exp ');

terminal =
      subname        %rule execution or assignment%
      / control      %function call%
      / confirm      %command confirmation%
      / feedback     %noise word feedback%
      / answer       %YES/NO answer to a question%
      / recognition  %built-in recognizers%
      / conditional  %IF statement%
      / show         %show variable to user%
      / abort        %abort current command%
      / resume       %resume remote procedure%
      / option       %get option character%
      / loop         %looping facility%
      / exit         %exit a loop%
      / tparameter;  %built-in values/variables/rules%

subname = .ID ['← parameter/ :← parameter];
```

The assignment operators include '← for total replacement of old contents with new and :← for appending a new element to a CML list variable.

```
conditional = "IF" ["NOT"] parameter
         [( '= / ">=" / "<=" / '< / '> ) (.ID/.NUMBER/"NULL")];
```

Parameters may be tested for TRUEness, FALSEness, NULLness, or for a binary relationship with some variable or integer.

```
show = "SHOW" ["CONFIRM"] '( parameter ');
```

Arbitrary parameters may be shown to the user. Appropriate conversions take place to make them intelligible if the parameter is not already a text string. Optionally, the user can be required to confirm that he saw the shown item before parsing can continue.

```
abort = "ABORT" ['( parameter ')];
```

It is sometimes necessary to designate that the current command specification should be aborted. This generally follows a test of a variable or a result of a remote procedure call. If the parameter is specified, it is shown to the user as in *show* above.

```
resume = "RESUME" ['( parameter ')];
```

When a remote procedure is called, it may, in the course of its processing, determine that it requires assistance from the Frontend/user. To avoid having to duplicate the processing that has already taken place, the Frontend provides a procedure that the remote procedure may call to get help. In addition to requesting certain types of built-in help, the remote procedure can request that a help rule associated with the original call be invoked. The built-in variable HELPCODE is set to a value provided by the remote procedure and the rule is executed. RESUME is used to allow the remote procedure to resume its processing, generally with a new parameter. ABORT may also be used to indicate to the remote procedure that the desired help could not be given.

```
loop =
    ("PERFORM" .ID "UNTIL" '( exp ')
    / LOOP '( exp '));

exit =   "EXIT";
```

The looping facility permits repetition of a parse rule until an exit condition is met.

The .ID following the reserved word PERFORM is the name of a parsing rule which is to be repeated. This rule is evaluated and then the expression following the UNTIL reserved word is evaluated. If the expression returns TRUE, then the loop is terminated and the next factor in the rule is evaluated. If the expression returns FALSE, then the rule identified by .ID is executed again. Note that this is a recursive execution of the rule, thus allowing the user to back up the parse state across repeated executions of the rule.

The LOOP construct does not use recursion and therefore does not allow the user to back up across iteration boundaries. An EXIT appearing within the expression being iterated will cause the iteration to terminate and the next factor in the rule is evaluated. The user may, of course, type an abort key and cause the iteration to be terminated along with the rest of the command.

THE COMMAND LANGUAGE COMPUTER AND EMULATOR

This section describes the Control Language Computer (CLC) and the current implementation of the Frontend's Control Language Interpreter (CLI).

The CLI is one of many possible implementations of the hypothetical CLC. It executes programs, called Grammars, that are produced by the CML compiler. Its principal function is to interact with the human user and allow him to enter the commands that are legal for the current tool, offering as much aid to the user as he wishes. When a command is specified, the CLI may issue one or more "procedure calls" to an execution module for the current tool. The CLI is cognizant of the device-dependent feedback and addressing characteristics of the user's terminal.

The CLC has one accumulator and a call stack. It processes its instructions in parallel. The current user alternatives are represented by separate instructions that are being processed in parallel.

The CLI is capable of prompting the user for the type of input required at the current point in command specification and responding to user requests for more detailed explanations of current alternatives, for full syntax of all or part of a command, or for full prose explanations of commands, command elements, or high level functions or concepts of a tool. In this last class, the CLI invokes a semantic help process which may execute on a separate computer in the network. The CLI makes use of a user-specific data base called the user profile which allows the CLI's interactions with this user to be tailored to him. To allow the user to change his user profile, a user-profile tool is made available.

The logical CLC instruction format is as follows:

OPCODE STATUS OPERAND ALTERNATIVE SUCCESSOR

where the fields are defined as follows:

OPCODE: This field identifies the type of instruction, such as "recognize a command word" or "load the accumulator from a variable". The current opcodes are enumerated below.

STATUS: This field's interpretation is dependent upon the opcode.

OPERAND: This field is used only with certain opcodes and contains the address of some additional data element that is necessary to execute the instruction. For the opcodes discussed above, this would be the address of the command word string and the address of the variable, respectively.

ALTERNATIVE: This field contains the address of another instruction to be executed in parallel with the current instruction. The set of instructions linked together by this field are called a "set of alternatives".

SUCCESSOR: This field contains the address of the (set of) instruction(s) to execute should the current instruction successfully execute.

It is worth pointing out that the instruction fields discussed above need not be present. In fact, there is no ALTERNATIVE field and all fields except the OPCODE field are optional and are not present in the instruction unless needed. Two bits and juxtaposition are used in place of the alternative field. When possible, this scheme is used also for the SUCCESSOR field.

DATA TYPES

The CLI environment supports fully typed data elements. These may be loaded into the accumulator, tested, stored into variables, passed to remote procedures, and so forth. The data types currently supported are:

String: Contains one or more character strings and an integer token specified in the Grammar (optional).

Command Word: Identical in structure to *String*. Data elements of this type contain command word strings instead of user-typed strings.

Address: Identical in structure to *String*. Data elements of this type contain strings typed by the user and meaningful to tools as identifying some object known by the tool.

Null: The null type. Useful as a default value.

True/False: Boolean values.

Point: A representation of an object on a display screen that has been pointed to by the user.

Block: A bit string. Useful in extending the data types for Parse Functions and tool supplied selection functions.

Integer: A 32 bit signed, two's complement integer.

List: A list of the above types.

The CLI consists of two major logical components, a sequencer and an instruction decoder/executor. The sequencer is responsible for selecting the set of instructions to execute in parallel and for selecting the set of instructions to execute next should one of the current instructions be successfully executed. The decoder/executor is responsible for decoding and executing individual instructions and is invoked by the sequencer.

THE SEQUENCER

The sequencer consists of two major procedures, PROCESS-A-RULE and PROCESS-A-SET-OF-ALTERNATIVES. These will be abreviated as P-RULE and P-ALTS, respectively.

Here, rule is used to mean a set of alternatives and all possible sets of alternatives that can be reached by successful execution of any instruction in any set. This describes a tree structure of instructions.

P-RULE is initially invoked to process the set of alternatives that form the base set of commands in the tool Grammar. P-RULE calls P-ALTS to process any single set of alternatives. When one of them successfully executes, P-ALTS returns to P-RULE, specifying which instruction succeeded. P-RULE determines the successor set of alternatives and again invokes P-ALTS to process them. This continues until a terminal point in the Grammar is reached. At this point, a command has been executed, book-keeping operations are performed, and the whole affair begins again.

In processing a set of alternatives, P-ALTS makes use of the array INSTRUCTION-FUNCTIONS. This array is indexed by the OPCODE value. It contains the address of the function to call to execute/decode this type of instruction and indicates whether or not the function may want to interact with the user in carrying out its function. P-ALTS establishes coroutine linkages with those functions that do wish to interact with the user and subroutine linkages with the others. P-ALTS does all the actual inputting from and echoing to the user. It maintains a list of all of the coroutine alternatives. When they have all asked for a character it acquires one and passes it to each coroutine (alternative) in turn. Each coroutine may return asking for another character; asserting that it has succeeded; asserting that it has succeeded but allowing for arbitration; asserting that it cannot succeed given current user input; or asserting that the parse state should be backed up.

THE EXECUTOR/DECODER

This component of the CLI is a set of procedures and coroutines that execute opcodes. Similar opcodes can often be handled by the same function. For opcodes whose successful execution requires interaction with the user, coroutines are used. For the rest procedures are used.

Perhaps the best way to describe this part of the CLI is to briefly list the currently defined CLC opcodes and their semantic meaning.

CLC Opcodes

RECOGNIZE COMMAND WORD: Interact with the user trying to match his input with the reserved command word associated with this instruction. Additionally, such command words are allowed to contain a "primary" or "secondary" status, where, in the event of a conflict, "primary" command words take precedence (see L2 discussion above). If a command word is correctly specified, and arbitration by P-ALTS indicates that this instruction should succeed, the accumulator is loaded with the token for this command word.

In the CLI, the function that performs this recognition supports four recognition schemes that can be set by each user:

Demand: This scheme requires the user to type a right delimiter character (usually space) to cause recognition to take place. P-ALTS invokes arbitration and may require the user to enter more characters in order to uniquely specify a command word among the set of alternatives.

Anticipatory: In this scheme, recognition is based on the minimum string of characters required to uniquely specify a command word among the set of alternatives.

Fixed: This scheme is identical to anticipatory but delayed until a fixed number of characters (currently three) have been entered.

First-letter: In this scheme, recognition is based on the first letter of the command word. If the instruction being processed indicates that this is a primary command word, then the first character the user types is compared with the first letter of the command word. If they match, the recognition function asserts success with arbitration by P-ALTS. If the current instruction represents a secondary command word, the user must first type an escape character (currently a space) to cause recognition of this command word. After the escape character has been entered, any of the recogntion schemes defined here may be applied to the recognition of the secondary command words. This secondary scheme is again specified by the user via his user profile.

LOAD VARIABLE: Load the value of the specified variable into the accumulator.

LOAD CONSTANT: Load the accumulator with the specified INTEGER constant.

LOAD COMMAND WORD: Load the accumulator with the token for the specified command word. The effect is just like *RECOGNIZE COMMAND WORD* but with no interaction with the user. This may also be used for loading arbitray text strings into the accumulator.

LOAD BOOLEAN: Set the accumulator to contain the value of Boolean TRUE or FALSE.

LOAD NULL: Set the accumulator to the value NULL.

STORE ACCUMULATOR: Store a copy of the value now contained in the accumulator into the specified variable. The previous contents of the variable are lost. Variables may be specified to be global or local. Local variables are cleared at the end of a command. Global variables maintain their values until explicitly changed.

APPEND ACCUMULATOR: Append the current accumulator value to a list data structure. If the specified variable does not contain a list data structure, one is created and the original contents of the variable are lost.

PUSH ACCUMULATOR: Push the current accumulator value onto the call stack.

CALL REMOTE FUNCTION: Pop the specified number of values from the call stack and pass them as arguments to the specified remote procedure. The remote procedure is contained in another process on the same or a different computer (assuming a network interconnects them). Such calls may be made "inline", in which case the CLI awaits its return (and results) before proceeding, or "out of line", in which case the CLI issues the call then immediately continues the parsing. In the latter case, an additional set of instructions may be specified that will get executed upon return of the procedure. The arguments and results are, of course, passed by value. In inline calls a help rule may be specified. This will get executed should the called procedure request help from the CLI (see *Resume* below).

GET CONFIRMATION: Obtain confirmation from the user before performing some operation. Currently this requires that the user type a specific confirmation character.

COLLECT AN ARGUMENT: Instruct the CLI to collect an argument of a specific type from the user. Argument types are declared in the CML Grammar. They may be built-in types or the collection function may be supplied along with the Grammar. Three types of argument collections may be specified as described above.

ANSWER: Collect a yes/no answer from the user. Loads a Boolean TRUE/FALSE into the accumulator.

OPTION: Collect an escape character from the user to allow him to utilize an infrequently-used or dangerous part of a command.

ABORT: Abort the current command specification, optionally displaying a message to the user.

PRESENT-NOISE-WORDS: Present the specified noise words to the user to help explain the semantics of a command. The user may specify in his user profile that he does not want to see noise words, in which case this instruction is a no-op.

TEST ACCUMULATOR: Test the current accumulator value for TRUEness, FALSEness, NULLness, equivalence to a variable or constant, or other binary relationships if the data types permit.

SHOW ACCUMULATOR TO USER: Generate a formatted character string representation of the current accumulator value and present it to the user, optionally requiring that the user type a confirmation character before he is allowed to continue.

EXECUTE A RULE: Process the specified rule as though it were physically substituted for this instruction. This is an extremely useful feature but greatly complicates the definition of "successor" and "alternative".

CLEAR COMMAND LINE: Cause a logical carriage return on the user's terminal. This is sometimes useful for formatting commands.

RESUME HELP: Return to a remote procedure with new arguments. When a remote function is called by the CLI, the called function can in turn call for help from the CLI. This may result from a syntactically correct but sematically incorrect value passed as an argument. Aside from certain built-in help facilities, the called routine can specify in the help call on the CLI that it would like a help rule invoked. RESUME is used in such a help rule to return to the remote procedure with new arguments.

CALL PARSE FUNCTION: Pop the specified number of values from the call stack and pass them as arguments to the specified Parse Function. Since it is recognized that the CML/CLI combination cannot meet all needs, a facility is provided for calling procedures that are loaded with the Grammar into the Frontend for processing by the CLI. These functions are somewhat limited in what they can do so that they cannot crash the CLI, but they are allowed considerable freedom to augment the normal facilities of the CML and CLI. It is hoped that as deficiencies are understood, facilities will be added to the CML language/compiler, the CLC, and the CLI to alleviate the need for the associated Parse Functions. In the meantime, the use of Parse Functions will serve as a stop gap.

LOOP and *EXIT LOOP*: Allow for iteration in addition to the already available recursion.

An example here might help the reader understand the nature of these instructions. Consider the CML rules

textitem = "LINE" / "WORD" / "CHARACTER";

rule =   "DELETE"

```
    ( type ← textitem <"at"> where ← DSEL(type)
    / type ← "FILE" <"named"> where ← LSEL(type) )
    delete ( type, where );
```

In this example, the user would be able to "delete" "lines", "words", "character", or "files". In the first three cases, the noise word "at" is typed to him and the user is then expected to specify the line/word/charcter to be deleted. This is done either by typing characters the tool will interpret (for example, a line number or a search pattern) or by pointing to it if the user is interacting via a display terminal. Associated declarations would indicate how this was actually to be done.

In the case of deleting files, the user is given the noise word "named" and expected to type the file name.

In all of these cases, the remote procedure "delete" is called. The type of item to be deleted and the specification of where it is or what it is named are passed as arguments.

The CLC instructions that are produced for this are as follows (omitted ALT fields imply ALT=0, omitted SUC fields imply SUC=NEXT):

```
textitem:     RECOGNIZE("LINE")[ALT=textitem+1, SUC=0]
textitem+1:   RECOGNIZE("WORD")[ALT=textitem+2, SUC=0]
textitem+2:   RECOGNIZE("CHARACTER")[SUC=0]

rule:     RECOGNIZE("DELETE")
rule+1:   EXECUTE(textitem)[ALT=rule+2, SUC=rule+7]
rule+2:   RECOGNIZE("FILE")
rule+3:   STORE(type)
rule+4:   NOISEWORD("named")
rule+5:   LSEL(type)
rule+6:   STORE(where)[SUC=rule+11]
rule+7:   STORE(type)
rule+8:   NOISEWORD("at")
rule+9:   DSEL(type)
rule+10:  STORE(where)
rule+11:  LOAD(type)
rule+12:  PUSH
rulg+13:  LOAD(where)
rule+14:  PUSH
rule+15:  CALL(delete,2)[SUC=0]  -- top two items on stack are passed as arguments
```

These instructions require 41 8-bit bytes to encode.

FUNCTION INTERFACE PROTOCOL

To date, the CLI has interfaced to tool Backends through three different protocols: a shared page protocol for intra-host initial checkout, the Distributed Programming System [6][7][8], and the MSG protocol [14]. Because of its internal organization, it is relatively easy to interface to different communication protocols.

At the time of this writing, the CLI has been interfaced to tool Backends within a single TENEX operating system. However, work is now in progress to extend the current protocol to inter-host capability. This can be done with either the Distributed Programming System protocol or the MSG protocol. The CLI is currently running on both a PDP-10 and a PDP-11.

Key threads of compatibility among the three protocols used to date are that all protocols either directly or indirectly allowed for a procedure call/return model for communication; that data structures were typed and therefore interpretable; and that call/return arguments and results are encoded into the same standard representation [7].

APPENDIX A: SAMPLE CML PROGRAM (taken from the NLS system)

```
FILE example

DECLARE FUNCTION

    PROCESS = "NLS-Backend", "PACKAGE" = "EDITOR":
        xreplace, xload, xdoit, setfield, showstatus;

DECLARE COMMAND WORD
    "CHARACTER" = 1 SELECTOR = CHARACTER,
    "WORD" = 2 SELECTOR = WORD,
    "VISIBLE" = 3 SELECTOR = VISIBLE,
    "INVISIBLE" = 4 SELECTOR = INVISIBLE,
    "TEXT" = 5 SELECTOR = TEXT,
    "LINK" = 6 SELECTOR = TEXT,
    "NUMBER" = 7 SELECTOR = INTEGER,
    "STATEMENT" = 8 SELECTOR = STRING,
    "GROUP" = 9 SELECTOR ADDRESS = getgroup,
    "BRANCH" = 10 SELECTOR ADDRESS = getbranch,
    "PLEX" = 11 SELECTOR ADDRESS = getplex,
    "OLDFILE" = 1 SELECTOR = OLDFILENAME,
    "IDENTLIST" = 1 SELECTOR = TEXT,
    "FILE" = 1,
    "PROGRAM" = 2;

SUBSYSTEM nlseditor KEYWORD "BASE"

% COMMON RULES %

    % PARAMETER TYPE DEFINITIONS %

        editentity = textent / structure;

    % TEXT PARAMETER TYPE DEFINITIONS %

        textent =
            "CHARACTER" / "WORD" / "VISIBLE" /
            "INVISIBLE" / "TEXT" / "LINK" / "NUMBER";

    % STRUCTURE PARAMETER TYPE DEFINITIONS %

        structure = "STATEMENT"/ "GROUP"/ "BRANCH"/ "PLEX";

replace COMMAND =

    REPLACE"

        type ← editentity

            % The rule EDITENTITY defined above is
            evaluated.  The one chosen (via user input) is
            stored in the variable TYPE. %

        <"at"> destination ← DSEL(type)

            % The user is presented the noise word "at" and
            requested to supply a destination of the type
            chosen from EDITENTITY.  The user must then
            identify the item to be replaced.  The
            representation of this item is stored in the
            variable DESTINATION. %

        <"by"> source ← LSEL(type)
        % The replacement is collected from the user
        and stored in the variable SOURCE. %

    CONFIRM

        % Have the user confirm that he wants the
        replacement to take place as specified. %

    xreplace( type, destination, source );

        % call the primitive in the application program
        that performs replacements.  Pass it the type
        of thing to replace, the specific instance of that
        type to be replaced, and the replacement. %

load COMMAND =

  "LOAD"
  type ← ("FILE"/"PROGRAM")

      % this command allows user's structured text files and
      programs to be loaded into NLS for user manipulation and
      execution, respectively. %

  filename ← LSEL(#"OLDFILE") CONFIRM

      % Collect the name of an existing file from the user.  The
      file may be the one to load or it may contain the program
      to be link-loaded. %

  xload(type, filename);

      % pass the application program's xload primitive the type
      of load and the file name. %

interrogate COMMAND = %interrogate user to help him send mail to other users%

  "INTERROGATE" CONFIRM

      % User wants to be interrogated for needed info to send
      mail to other users. %

  CLEAR <"distribute for action to:">

  content ← LSEL(#"IDENTLIST")

  setfield(#"ACTION", content)

      % CLEAR causes Carriage Return Line Feed on a
      typewriter-like terminal and causes the command area to
      be cleared on a display.  The application function
      setfield is called to set the "action" field in the current
      message header to the list of user recipients supplied by
      the user and stored in CONTENT. NOTE: this could have been
      stored in the Frontend until all parameters were collected and
      then passed to the Backend in one call.  This is up to the
      command-language and tool designer. %

  CLEAR <"distribute for information-only to:">

  content ← LSEL(#"IDENTLIST")

  setfield(#"INFORMATION", content)

  CLEAR <"title:"> content ← LSEL(#"TEXT")
```

```
    setfield(#"TITLE", content)
    CLEAR <"type of source:">
    (
    "MESSAGE" type ← #"STATEMENT"
        content ← LSEL(#"TEXT")
            % Message is the same as statement. %
    / type ← "FILE"
        content ← DSEL(#"CHARACTER")
            % The user may specify any character in the
            file. %
    / type ← structure
        <"at"> content ← SSEL(type)
            % Since this is an SSEL, the user may type it or
            specify its location in one of his files. %
    / type ← "OFFLINE" <"document">
        <"located at"> content ← LSEL(#"TEXT")
            % If it is an offline hardcopy document, simply
            have the user describe where it is stored.%
    )
    setfield(type, content)
    CLEAR <"show status?"> [ IF ANSWER showstatus() ]
        %If the user answers "YES", call SHOWSTATUS to present
        the current specification of the mail to the user. %
    CLEAR <"send the mail now?"> [ IF ANSWER xdoit() ] ;
        % If the user answers "YES", call XDOIT to send the mail
        as specified, otherwise simply let him use other commands
        to change the specifications and send it. %
END.
FINISH
```

APPENDIX B: COMPLETE FORMAL SYNTAX OF CML

```
file =  "FILE" .ID $dcls $rule #subsys "FINISH";

subsys =
     "SUBSYSTEM" .ID % subsystem handle %
     "KEYWORD" .SR  % recognition string %
     #(command / rule)
     "END.";

command = (cmdrule/
     ("COMMAND" / "INITIALIZATION" /
     "TERMINATION" / "REENTRY")
          rule);

rule       = .ID '=  exp '; ;
cmdrule        = .ID "COMMAND" '=  exp '; ;

dcls =
     ("CONSTANT" #<',> (.ID '= .NUMBER)/
     "DECLARE"
          ( ["VARIABLE"  /  "PARSEFUNCTION"] #<',> .ID /
          "FUNCTION" "PROCESS" '= .SR ', "PACKAGE" '= .SR ':
               #<',> (.ID ["OUT" "OF" "LINE"] ["PSEUDONYM" '= .ID]) /
          "COMMAND WORD" #<',> (.SR = .NUMBER [selector])));

selector  = "SELECTOR"
      ('= builtin /
      #("TYPEIN"/"ADDRESS"/"POINT") '=
          (builtin/.ID %selection function%));

builtin    =
      ("CHARACTER"/"WORD"/"TEXT"/"INTEGER"/"NUMBER"/
      "FILENAME"/"NEWFILENAME"/"OLDFILENAME"/"VISIBLE"/
      "INVISIBLE"/"STRING"/"WINDOW"/"OCTALINTEGER");

exp = #<'/>alternative;

alternative = #factor;

factor = terminal / '( exp ')/ '[ exp '];

terminal =
     subname/ control/ confirm/ answer/ option/ feedback/ recognition/ loop/
     conditional/ show/ abort/ resumehelp/ tparameter/ exit;

subname = .ID ['← parameter/ :← parameter];
confirm  = "CONFIRM";   % command confirmation %
answer   = "ANSWER";    % YES/NO answer to a question %
recognition = command-word/ selection;
command-word      = .SR [ '! #qualifier '! ]
qualifier        = "L2" / .NUMBER;
selection  = ("SSEL"/ "DSEL"/ "LSEL") '( type ');
feedback = clear-feedback / extra-feedback / replace-extra-feedback;
extra-feedback = '< .SR '>;
replace-extra-feedback = '<"..." .SR '>;
clear-feedback = "CLEAR";
control   = .ID % routine name % [ call-qualifier]
               '( $<',> parameter ["->" #<',> .ID] ');
call-qualifier = '[ .ID / "IN" "LINE" / "OUT" "OF" "LINE" '];
conditional = "IF" ["NOT"] parameter
          [('=/">="/"<="/'</'>) (.ID/.NUMBER/"NULL")];
show = "SHOW" ["CONFIRM"] '( parameter ');
abort = "ABORT" ['( parameter ')];
resume = "RESUME" ['( parameter ')];
```

```
parameter =
        factor/ % expression element %
        tparameter;
tparameter =
     / "VALUEOF" '( .SR )      % command-word value %
     / '# .SR                  % same as VALUEOF %
     / "TRUE"                  % boolean TRUE value (one) %
     / "FALSE"                 % boolean FALSE value (zero) %
     / .NUMBER                 % arbitrary non-negative integer%
     /"TYPEWRITER"             % boolean TRUE if user has a typewriter terminal%
     /"DISPLAY"                % boolean TRUE if user has a display%
     /"LINEATATIME"            % boolean TRUE if user has a
                               line-at-a-time terminal%
     /"HALFDUPLEX"             % boolean TRUE if user has a half-duplex
                               terminal%
     /"BREAKPOINT"             % boolean TRUE if tool is at a break point%
     /"HELPCODE"               % contains NULL or an integer help
                               code (see below)%
     /"RESULT1" ... /"RESULT8" % arbitray results from remote call%
     /"RESULT                  %= RESULT1; arbitray result from remote call%
     /"NODE"                   % logged in user name%
     /"PROJECT"                % logged in user account%
     /"CURRENTTOOL"            % name of tool to which user is
                               currently giving commands%
     /"ACTIVETOOLS"            % list of active tools (being run concurrently%
     / "WINDOW"                % identifier of current window ontaining cursor %
     / "NULL";                 % null pointer value (zero) %

loop = "PERFORM" .ID "UNTIL" '( exp ')/ "LOOP" '(exp');
exit = "EXIT";
```

ACKNOWLEDGEMENTS

The author wishes to acknowledge the significant contributions of other ARC staff members to the work reported in this paper. In particular, recognition should be given to Donald I. andrews, William H. Paxton, and J.F. Rulifson for their early work on Control Meta Languages[4][16] and to Charles F. Dornbush for implementing the CML compiler and interpreter now in use in the NLS-8 system. Ideas and implementation support were given by Donald C. Wallace, Keneth E. Victor, Andrew Poggio, and Donald I. Andrews. This work was conducted under the able management and supervision of Douglas C. Engelbart and Richard W. Watson.

This work was sponsored by the Defense Advanced Research Projects Agency, the Air Force Data Automation Agency, the Air Force Rome Air Development Center, and the National Aeronautics and Space Administration.

REFERENCES

[1] DOUGLAS C. ENGELBART.
Design Considerations for Knowledge Workshop Terminals.
In AFIPS Proceedings, Vol. 42, 1973 National Computer Conference, pp. 221-227, 1973. (ARC Journal # 14851)

[2] DOUGLAS C. ENGELBART, RICHARD W. WATSON, JAMES C. NORTON.
The Augmented Knowledge Workshop.
In AFIPS Proceedings, Vol. 42, National Computer Conference, pp. 9-21, 1973. (ARC Journal # 14724)

[3] STAFF OF THE AUGMENTATION RESEARCH CENTER
Online Team Environment / Network Information Center and Computer Augmented Team Interaction.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. 6-MAR-73. (ARC Journal # 13041)

[4] DOUGLAS C. ENGELBART, WILLIAM K. ENGLISH.
A Research Center for Augmenting Human Intellect.
In AFIPS Proceedings, 1968 Fall Joint Computer Conference, Vol. 33, Part I, p. 395-410, 1968 (ARC Journal # 3954)

[5] DOUGLAS C. ENGELBART.
Human Intellect Augmentation Techniques.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. JUL-68. (ARC Journal # 3562)

[6] JAMES E. WHITE.
Version 2 of the Procedure Call Protocol (PCP).
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. (ARC Journal # 24590)

[7] JAMES E. WHITE.
A High Level Framework for Network Based Resource Sharing.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. Being prepared for the 1976 NCC.

[8] JAMES E. WHITE.
Elements of a Distributed Operating System.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. Being prepared for publication in the Journal of Computer Languages.

[9] STAFF OF THE AUGMENTATION RESEARCH CENTER.
Tree Meta Report -- Preliminary Draft.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. 23-JAN-73. (ARC Journal # 14045)

[10] STAFF OF THE AUGMENTATION RESEARCH CENTER.
Tree Meta Report -- Preliminary Report, Formal Description.

Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. 23-JAN-73. (ARC Journal # 14046)

[11] CHARLES H. IRBY.
Display Techniques for Interactive Text Manipulation.
In Proceedings of the National Computer Conference, 1974, p. 247-255.
(ARC Journal # 20183)

[12] DOUGLAS C. ENGELBART.
Coordinated Information Services for a Discipline- Or Mission-Oriented Community.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. 12-DEC-72.
Paper given at Second Annual Computer Communications Conference, San Jose, California, 24 January, 1973. 13p. (ARC Journal # 12445,)

[13] DOUGLAS C. ENGELBART, WILLIAM K. ENGLISH, J. F. RULIFSON.
Development of a Multidisplay, Time-shared Computer Facility and Computer-Augmented Management-System Research.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. April, 1968. (ARC Journal # 9697)

[14] STAFF OF MASSACHUCETTES COMPUTER ASSOCIATES.
MSG Protocol Working Documentation.
Unpublished.

[15] STEPHEN D. CROCKER, WILLIAM A. CARLSON.
The Impact of Networks on the Software Marketplace.
Proceedings of the EASTCON, 1973

[16] WILLIAM H. PAXTON.
L10 -- A Programming Language for the Augmentation Research Center (System Programming Guide)
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. 1970 (ARC Journal # 7052)

[17] STAFF OF THE AUGMENTATION RESEARCH CENTER.
Knowledge Workshop Development.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. 1974

[18] STAFF OF THE AUGMENTATION RESEARCH CENTER.
National Software Works Development.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. 1975

[19] RICHARD W. WATSON.
User Interface Design Issues for a Large Interactive System.
Augmentation Research Center, Stanford Research Institute, Menlo Park, California 94025. being prepared for the 1976 NCC.

[20] R. M. BALZER, T. E. CHEATHAM, S. D. CROCKER, S. WARSHALL
National Software Works Design
USC/Information Sciences Institute RR-73-16 NOV-73

[21] R. M. BALZER, T. E. CHEATHAM, S. D. CROCKER, S. WARSHALL
National Software Works
USC/Information Sciences Institute RR-73-18 DEC-73