P-14

CONDITIONAL CONVERSATIONAL COMMAND PROCESSING

Charles A. Grant

University of California, Berkeley

Document No. P-14
May 1, 1969
Office of Secretary of Defense
Advanced Research Projects Agency
Washington 25, D. C.

Grant, Charles A.

Conditional Conversational Command Processing

ABSTRACT:

A general programming facility is proposed for communication with the interactive command languages of time-sharing systems in an attempt to overcome some of the current limitations of data exchange between man and machine. Commands may be constructed in an arbitrary way in a string processing language and then processed as if typed to a console by a user. The output resulting from the sent commands may be dissected and examined to determine subsequent action.

A set of functions to accomplish the above which could be embedded into any string processing language is suggested, and necessary information pertinent to implementation of the facility on existing time-sharing systems is given.

Key Words: Time-sharing, Command languages, Pseudo-teletype, Interaction, Conditional Job Control, Operating Systems

CR Categories: 3.80, 3.81, 4.29, 4.39

1.0 INTRODUCTION

Experience with time-sharing systems has shown some unsatisfactory conditions concerning the communication between user and system. This communication takes place via a teletype or display console in the command language of whatever program the user is operating. This paper will not concern itself with the design of command languages but rather with the solution of the following problems:

A) Users of time-sharing systems find that there may be sequences of commands that they frequently enter with little or no variation. Even more annoying than the repetition may be the time required for the physical console to accept the command and print the reply, or, more seriously, the possible loss of information due to the inevitable occasional typing error. For example, it is common that at the beginning of a session with the system there is a standard sequence of commands a user will give in order to retrieve his files from secondary storage, possibly assemble or compile them, and then initialize the particular subsystem he will use.

B) A user may wish to enter a long series of commands which, due to interspersed computation, requires him to be present at the console for a much longer period of time than is necessary to type the commands. For example, if a user wishes to assemble or compile several packages of a large program, rather than issuing the sequence of commands together, he may have to wait for each computation to complete before entering the next request, thereby partitioning his free time. Time-sharing systems have proved invaluable for speedy construction and debugging of programs but there are many programs which, when completed, will compute for a long period of time with no need of human interaction. Some systems provide an offline mode for running these programs but in general the command languages are different from the online command languages and less adequate.

C) There are many dialogues with the computer which require very little creative intervention by the user. His presence at the console may be necessary solely to chaperone the computation to check for errors or to supply as input to one program the output of a previously executed one.

What is needed in a time-sharing system is a continuum of capabilities ranging from pure non-interactive (read batch processing) on the one hand to highly interactive on the other.

2.0 HISTORY

Some work which has been done in this area will now be described.

In an early effort, since superseded, the SDS-940 time-sharing system at Berkeley provided a system which would operate on a character file in the following way: Characters would be taken from the file and delivered to a "pseudo-teletype". The system was deluded to believe that a pseudo-teletype was no different from a real teletype, and that characters sent from the file were actually typed at its keyboard. The pseudo-teletype would react to these characters in exactly the same way a real teletype would react if the same characters were typed to it by a user. The output response to this input at the pseudo-teletype was diverted to the console of the programmer using the facility. A file might contain breakpoints which would cause interruption of the program. Breakpoints could be placed wherever it was anticipated that creative or unpredictable intervention by a human was required. The user was allowed to interact arbitrarily with the pseudo-teletype, through his own console, at these points and then type a command to continue sending characters from the file.

CTSS, the time-sharing system for the IBM 7094 at MIT, made available to its users a program called RUNCOM. This is again a facility for sending a sequence of commands to a "pseudo-teletype". A macro facility (recursion and nesting allowed) which permits assigning a name to a series of commands is provided as well as a restricted conditional facility. The conditional facility is actually a special command in the time-sharing system which takes as an argument a symbolic name. If the name is not the name of an existing file then the RUNCOM program will ask the user whether or not to abort. Testing in this fashion proves to be cumbersome and very ad hoc but it is nevertheless found to be quite useful for detecting errors.

While JCL, the job control language for the IBM 360, is not a time-sharing command language it has attacked analogous problems for batch processing. JCL allows the definition of a sequence of commands as a macro with symbolic parameters. Calling a macro from an input card deck after using special commands to initialize the values of its parameters results in the series of parameterized commands being executed. JCL also has a conditional facility. Each "step" or statement of the command sequence has associated with it a "return code" which is a positive integer assigned to the step after it has been executed. The return code is an error indicator returned by each executed command upon termination. We can write a statement in the language which makes numeric tests on the return codes of specified previously executed steps. The result of a test statement is to decide whether the next sequential statement in the program is to be bypassed or executed.

3.0 THE LANGUAGE

The above three systems are all similar in that they essentially provide a facility for sending a linear sequential list of commands to be executed as if entered from a console. The macros were introduced as a labor-saving device and the conditionals allowed a small amount of control over the job being executed. We propose that a language with goto's, functions, conditionals, and generalized string pattern matching statements is more suited to the task of controlling interactive processes. The language should have the facility to send a command (or just a string of characters) to a pseudo-teletype and then wait for the complete response to this input. Subsequent action on the part of the program can then be based on the content of the response (i. e. the output of the pseudo-teletype). With reasonable conversational features in the language, a program can selectively choose the significant input and output required by the user and obtain the above mentioned continuum.

We will provide below a list of atomic functions which could be easily embedded into an existing implementation of any language with string-handling capabilities. These functions would provide complete, minimum capabilities for communication with the pseudo-teletype. That is, a function will be provided which takes as argument a string of characters. Calling the function will cause the characters to be sent to the pseudo-teletype. Other functions will be provided to collect the output characters. With this facility we can do arbitrary computation to generate commands and then do complex analysis of the response. We can imagine very grandiose applications of the facility. For example consider a program in a language containing the special functions which constructs programs in some other language. The constructed programs could be entered, compiled and executed on the pseudo-teletype and then evaluated on the basis of their output. Another application would be to construct an interface between user and system which is radically different from the standard command language provided. The string processing language could accept commands in the new syntax and then transform them into meaningful commands for the standard command language. More commonly, however, use of the language would be to consolidate lengthy command sequences into a single parameterized command. Mechanical error checking and other automatic operations would thereafter be removed from the user's responsibility.

4.0 THE COMMUNICATION FUNCTIONS

A complete set of atomic functions for communicating with the pseudo-teletype will now be listed. The functions could be added to a processor of any reasonable string manipulating language, like SNOBOL, TRAC, or COMIT, or even, in the form of system calls, to assembly language. The syntax of the presented functions will naturally depend on the language of their embedding.

LOGIN(<name> , <password>)

This function obtains a pseudo-teletype for the program and enters the named user on it if the password is acceptable. Null arguments will cause the name and password of the user running the program to be used. The now active pseudo-teletype is left in a state where it is awaiting its first command. The function will fail and do nothing if the arguments do not result in a legal entrance to the system.

LOGOUT( )

This function of no arguments causes the pseudo-teletype to be logged out, regardless of what state it is in. The pseudo-teletype is automatically logged out at the termination of the program even if this is not explicitly requested.

WAIT( )

This function of no arguments causes a pause in execution of the control program until the job on the pseudo-teletype is in a state where it can do nothing without recieving more input. That is, it waits until the pseudo-teletype is done with its current computations. This function has a null value and causes all output of the pseudo-teletype generated while waiting to be lost. This function is necessary to guarantee that all output from past commands to the pseudo-teletype has been generated. Without this facility we would be hard pressed to decide which output was associated with which command.

SEND(<string>)

SEND first does a WAIT and then delivers the characters in the string to the pseudo-teletype as it requests input. The SEND function returns a null value.

FORCED SEND(<string>)

FORCED SEND forces the psuedo-teletype into the highest level of the time-sharing system command language, interrupting any computation that may be executing. (A possible method of implementing this would be to send a special escape character to the input buffer of the pseudo-teletype which would be recognized by the time-sharing system as a request for this particular action.) Then the argument string is sent as with the SEND function.

RECVCHAR(<number>)

RECVCHAR takes as argument an expression that evaluates to a positive integer, N. It collects X output characters from the pseudo-teletype resulting from the last SEND or FORCED SEND function call, where X is less than or equal to N. If there are less than N characters (but at least one) then these characters are returned as the value of the function. If there are N or more characters of output then the first N characters are returned as the value. If there is no output then the function fails.

RECVLINE( )

The purpose of this function is to gather output from the pseudo-teletype line by line. The function takes no arguments and returns as its value the next line from the output of the pseudo-teletype. If there is no more output then the function fails.

ECHO(<number>)

For conversational applications, it may be desireable to occasionally allow the user to interact directly with the pseudo-teletype through his own console. A program to do this would

a) accept a character (or characters) from the user
b) SEND the characters to the pseudo-teletype
c) gather the reply, if any, with RECVCHAR or RECVLINE
d) print the reply on the user's console
e) go to step a.

However, input characters will appear twice in the output at the user's console - once for his typing of the character, and once as part of the output of the pseudo-teletype. The function ECHO, when called with negative argument, turns off the echoing of all characters input to the user's console. A subsequent call to ECHO with a non-negative argument will turn the echoing back on again. This function has other uses in an interactive language; for example, to accept secret passwords without having them printed.

The appendix contains a program which represents an example of the use of the above functions (imbedded in a SNOBOL3 language processor) operating on a hypothetical time-sharing system. The example demonstrates a straightforward application of the facility for consolidating a lengthy command sequence into a single command. It also shows how the conditional features can be used both to control and evaluate the execution of the job and to do it with concise, parameterized programs.

By providing a time-sharing system with a program to queue process control program file names and a supervisor program to sequentailly execute process control jobs from the queue, we can have a very reasonable background batch processing facility integrated into the system. These jobs would be able to communicate with the command languages of the time-sharing system and could execute with a great deal of conditional control over themselves. It would be very convenient to prepare, edit and submit background jobs from a time-sharing console.

5.0 IMPLEMENTATION

Implementation of the above functions will vary depending on the characteristics of the time-sharing system. The implementation at Berkeley required provision by the time-sharing system of four privileged routines which when called by a suitably authorized program would:

A) Simulate the input of a character at the keyboard of another teletype. (i.e. put a character into the input buffer of that teletype.)

B) Suppress the typing of characters put into the output buffer of another teletype.

C) Read characters out of the output buffer of another teletype.

D) Determine if another teletype is running a program which is dismissed waiting for teletype input.

Needless to say the physical existence of a teletype for this job is not required.

This method of implementation was influenced by the nature of the already existing time-sharing system, and is not completely satisfactory. When a process control job is running under this implementation, the time-sharing system actually sees two separate jobs - the controlling program and the job being controlled. This means that two entry ports to the system are absorbed although the two jobs are operated by only one user and seldom compute in parallel. Not only is this a waste of a valuable system resource, but it also results in a difficult accounting problem. During process control, twice as much LOGIN time is charged than is actually spent by a user at a physical console. Another complaint is that the indirect approach of using teletype buffers for the communication of commands between the two jobs seems inefficient and unclean.

In a more versatile operating system one would wish to cause the controlled job to execute as a subsidiary or parallel process of the controlling program. This introduces some problems however. It is important that the controlled job execute exactly as if it were entered from a standard console. In particular, it must have all the same capabilities (e.g. amount of memory, number of devices attachable, number of files it can simultaneously open, etc). Also, its universe of discourse must be restricted to only its own created environment, and not that of the controlling program. For example, if in the course of its computation, the controlled job executes the operation "CLOSE ALL FILES", the controlling program should not be effected.

As for the command communication between the two jobs, a parameterized input/output structure is needed. We should be able to specify as one of the initial parameters to a job, that its command input and output will take the form of communication with a particular process control program. This will cause each call to a teletype input/output routine to execute an appropriate process communication routine instead of a teletype operation. Notice that if the operating system permits any operations which make assumptions about the nature of command input/output (e.g. "CLEAR TELETYPE OUTPUT BUFFER") then these routines must perform an equivalent operation when executed by a controlled job.

These requirements for a clean implementation of the conditional conversational command processing facilities are in fact general problems of current operating system design.

6.0 CONCLUSION

In 1967 a special-purpose programming language devoted entirely to interactive process controlling called CCP (Conditional Command Processor) was implemented on the SDS 940 in the spirit of the above. The language had a profound impact on the use of the time-sharing system by people who construct and maintain large programs. The assembly and loading of these programs has been almost completely automated by the use of the language. The most notable example of this is the assembly and loading of the time-sharing system itself, which requires a CCP program six pages in length. The operations required are confusing enough that the chance of their being performed correctly by humans is less than fifty per cent: with CCP the entire process can be performed automatically in a relatively short time with no human intervention required.

Now completed is the embedding of the functions listed above in an implementation of SNOBOL4 at Berkeley. The much greater power in the SNOBOL language has enabled much more complex job control programs to be written, programs which can adjust their execution in a very flexible manner as they observe the course of the job being controlled. For example, one programmer interested in a new interactive text editing command language has built an interface with the standard editor on the system to allow experimentation with it.

Butler Lampson advised this research from the beginning, and with Larry Barnes helped specify the communication functions and their implementation on the SDS 940.

REFERENCES

The Compatible Time-Sharing System

P.A. Crisman

M.I.T. Press, 1966

IBM System/360 Operating System Job Control Language

IBM Doc. 28-6539-7, May 1968

Reference Manual - Time-sharing System

L.P. Deutsch, L. Durham, B.W. Lampson

Project Genie Doc. R-21, University of California,

Berkeley, October 1968

CCP - Conditional Command Processor

C.A. Grant

Project Genie Doc. R-29, University of California,

Berkeley, July 1967

Interactive SNOBOL4 System for the SDS 940

R. Sturgeon

Project Genie Doc. R-34, University of California,

Berkeley, December 1968

APPENDIX

```
*THIS SNOBOL3 PROGRAM ACCEPTS AS INPUT A LIST OF FILE NAMES
*WHICH ARE EXPECTED TO BE THE NAMES OF SYMBOLIC ASSEMBLY
*LANGUAGE PROGRAMS. THE NAMES ARE TO BE SEPARATED BY COMMAS.
*
*EACH FILE 'X' IS ASSEMBLED AND THE BINARY IS OUTPUT TO THE
*FILE 'BINX'. AFTER ALL ASSEMBLIES HAVE BEEN DONE, THE FILES
*ARE ALL LOADED AND THEN THE RESULTING CORE IMAGE IS DUMPED
*TO THE FILE CALLED 'DUMP'. IF THERE ARE ANY ERRORS DURING
*THE ASSEMBLY OF THE FILES, THE NAMES OF THE FILES
*AND THE ASSOCIATED ERROR MESSAGES ARE PRINTED, BUT NO
*LOADING IS DONE.
START        LOGIN(SMITH,PASSWORD)
*SEND MESSAGE TO USER TO INPUT FILE NAMES
             OUTPUT = 'ENTER FILE NAMES: '
*READ IN THE FILE NAME LIST
             FILELIST = INPUT ','
*THE COMMA IS USED TO ALLOW SIMPLE PATTERN MATCHING TO
*INDIVIDUALLY REMOVE ALL THE NAMES FROM THE LIST.
             FILECOPY = FILELIST
*THE COPY OF THE LIST WILL BE USED DURING LOADING.
ASSEMBLOOP   FILELIST *NAME* ',' =     /F(LOAD)
*NOW 'NAME' HAS AS VALUE THE NEXT FILE NAME TO ASSEMBLE
         SEND('ASSEMBLE FILE: ' NAME ' TO FILE BIN' NAME)
         ASSEMBLOUT = RECVCHAR(1000000)
*BY USING A LARGE NUMBER WE ARE SURE TO GET ALL THE
*MESSAGES GENERATED BY THE ASSEMBLER IN RESPONSE TO
*THE COMMAND SENT ABOVE.
             ASSEMBLOUT 'INVALID'   /S(ASSEMBLERROR)
             ASSEMBLOUT 'ERROR'   /S(ASSEMBLERROR)
             ASSEMBLOUT '?'  /S(ASSEMBLERROR)F(ASSEMBLOOP)
*IF WE FOUND ANY ERRORS WE WENT TO 'ASSEMBLERROR',
*OTHERWISE WE WENT BACK TO GET THE NEXT FILE NAME.
LOAD         EQUALS(ERRORFLAG,1)      /S(DONE)
             SEND('LOADER SYSTEM')
*NOW WE CAN LOAD EACH FILE
LOADLOOP     FILECOPY *NAME* ',' =    /F(DUMP)
             SEND('LOAD FILE BIN' NAME)     /(LOADLOOP)
DUMP         SEND('DUMP LOAD ON FILE: DUMP')     /(DONE)
ASSEMBLERROR   ERRORFLAG=1
               OUTPUT= 'FILE ' NAME ' HAD ERRORS: '
               OUTPUT = ASSEMBLOUT    /(ASSEMBLOOP)
*AFTER PRINTING THE ERROR MESSAGES WE WILL GO ASSEMBLE THE
*OTHER FILES IN CASE THEY ALSO HAVE ERRORS.
*
DONE         LOGOUT()      /(END)
*
*
```