FORTH LANGUAGE EXTENSION FOR CONTROLLING INTERACTIVE JOBS ON OTHER MACHINES

(Paper submitted to EuroFORML Conference, Stettenfels Castle, Federal Republic of Germany, October 25-27, 1985.

David K Walker
Virtual Walker & Walker
Postboks 68, N-5084 Tertnes, NORWAY.

ABSTRACT

A Forth application on an IBM PC/XT is described for (1) collecting, editing and generating input for a large model of the Norwegian economy used by the Norwegian Government, (2) transferring this information to a mainframe, and (3) running interactive jobs which check the input, process it further and send it on to another mainframe where the economic model equations are solved and result tables are written. The application emulates a person operating a computer terminal. Forth techniques illustrated include a finite state description language extension for controlling general interactive jobs on other machines. Anticipated further developments of this concept include controlling and logging the entire model solution process on several machines.

<u>The application</u>

The work reported on here, partly completed and partly still in development, suggests powerful Forth language extensions using the finite state machine concept (see for instance Landau, 1982) which seem likely to be particularly helpful in communications applications.

The Norwegian Government has for a quarter of a century used large computer models of the economy, particularly the MODIS model (for MOdel of DISaggregated type), as decision models in "what if" analyses to set market parameters such as taxes and subsidies, the exchange rate for the Norwegian currency, and total government expenditure. On many occasions the models have been used for centralized wage bargaining in which the Government as a third party has been active in using the models to negotiate packages involving joint taxation and wage rate concessions. The main characteristic of this decision process, compared with similar methods used in the USA (by the Congressional Office of the Budget), Canada, France, and Holland, is the maturity of the application and the extent to which it has been relied upon by the Norwegian Minister of Finance and the rest of the Cabinet in making quantitative economic decisions.

With changing machine types and the general move towards more interactive systems with improved turn-around times, and with the particular aim of picking up errors earlier in the model solution process, a microcomputer application in Forth has been installed for

- collecting, editing and generating input for MODIS
- transferring this information to a mainframe over a 9600 baud leased voice-grade link, and
- running interactive jobs which: check the input, process it further and send it on to another mainframe where the economic model equations are set up and solved as previously, and result tables are written.

This development is the first step in renewing the computer methods used for MODIS. The microcomputer program will be able to play a role in coordinating parallel use of new and old model implementations, to test the new implementations on new machines using identical input data.

The microcomputer application can be modified and recompiled to control interactive jobs on any machine to which it can be connected by RS-232C interface, using a special-purpose language which has been implemented in its first rudimentary but reliable version, as an extension of Forth. The sections following discuss the general goals, reasons why Forth is preferable for such applications, and the extensions which are appropriate. Plans for further development are discussed with special reference to guarantees of reliability.

Controlling interactive jobs on other machines

Conventional software for using microcomputers as workstations connected to mainframes supports two modes of interaction between machines,

- dumb terminal emulation, and
- downloading and uploading of files consisting of ASCII strings.

(In connection with the second, batch jobs can sometimes be submitted and their results viewed as files.) Usually coded in assembly language, software of this type is generally unable to provide machine-machine interactions related to specific final user applications. Such communications are conventionally regarded as a systems programming topic. The data objects dealt with are the kinds of data object that operating systems know about: terminals, files, job queues and so on.

This conclusion holds also for more advanced communications software used in local area networks and wide area networks: all communications is handled at the operating system level. Many would say that this is where it should be handled, that things should be factored out that way. The operating system, it is normally argued, should provide facilities of this kind to user processes. Why should the application have to include drivers for the communications ports? But we need to distinguish between responsibility for driving the ports and the high level protocols, as in the layered OSI model (see for instance Witt, 1983). As the layered OSI model recognizes, not only operating systems but also applications have to take account of things like deadlocks due to both parties waiting for the other to speak first. The main logical issues of communication have to be faced at all layers, including the applications layer. And so we need to ask: what software tools do we have to meet this challenge?

There is an almost total absence of interactive communications software at the applications level. Final users prefer interactive programs for data retrieval, spreadsheets, and similar applications. They would also like to be able to interact with communications software. It is useful to be able to start a program which will run an interactive program on another machine in an intelligent way and report back on selected events, as in the application described in this paper, or perhaps to run two interactive programs on two different machines and pipeline the output from one to the other. This sort of thing is rarely done because of the absence of adequate communications protocols at the application level in the OSI model. This absence leads to applications communicating by reading each other's files. This is the weakest and most error-prone kind of communications protocol that can be imagined, and it normally requires human intervention and checking to work, as we know from experience with MODIS.

Users want to do the things suggested above. Also, there exist interactive programs on mainframes which do not produce results on files suitable for downloading. Users would like interfaces between interactive programs on a variety of machines, which they are unable to modify themselves.

It is perhaps easy to imagine that a microcomputer emulating a smart terminal might simply take a disk copy of the dialog, which might be edited later by hand and uploaded to a different program on a different machine. There are many pitfalls here, however. The second machine may not want its input in that format. More fundamentally, a terminal emulating program needs to use a protocol such as XON/XOFF to slow down the mainframe while the microcomputer stores things on disk (or anywhere), and it will not be able to respond naturally at 9600 baud in a dialog if it tries to do this. Ideally, the microcomputer should act as an intermediary and buffer at the level of the interactive application.

Forth vs. other methods for communications applications

Assume for the moment that we are going to use assembly language, conventional high level languages such as C or BCPL, or new classical languages such as OCCAM (May & Taylor, 1983) -- which is specifically slanted towards communications. The absence of anything corresponding to Forth's text interpreter (Ting, 1980) makes life very difficult when implementing and testing the protocol in a communications application. This is perhaps not very important in small applications, but in large applications taking a variety of user needs and user-level exceptions into account it is an immense stumbling block.

Testing communications software using a top down testing approach with "stubs" instead of the complete code in the classical way (see for instance Myers, 1979) is not practical, as necessary input for the program to function is not generated. Furthermore, it does not help with the essential task of investigating the timing of messages between machines. Bottom-up testing is difficult with conventional methods because there is an excessive number of memory variables to be initialized correctly, by comparison with a Forth program. Forth programs run in a simpler context, which the system developer can initialize manually for bottom-up testing.

Forth's bottom-up interactive support for testing allows most of the necessary test input to be placed on the data stack before trying out selected words, and this allows experiments with timing. Correct hardware behaviour can also be verified by such testing. Individual modules can be constructed with a small number of possible program execution paths, and their correct function and stability can therefore be verified by experiment. Furthermore, at the crucial point when the whole code is first tried out, a function key breaking to Forth's text interpreter (Ting, 1980) allows us to investigate buffers and state variables, make alterations to both program and data, and re-run parts of the code.

While some of these facilities are available in machine code monitors and high level debugging monitors, it is normal for a monitor to interfere with i/o for keyboard and screen. Using Forth's text interpreter instead, such interference is well documented and can be modified if necessary.

Forth is apparently unique in providing the multi-level testing environment which is essential for implementing and testing communications applications in high level code. Most communications software has until now been written in assembly language. Regardless of whether or not a good macro-assembler has been used, a DDT-like debugging environment is not particularly helpful for testing large amounts of communications code. Single or multi-stepping ignores timing constraints in communications. It is in any case time-consuming in large programs. Setting breakpoints, registers and memory locations for longer test runs is less flexible and slower than using Forth's text interpreter, where for example buffers can be cleaned out and reset using tailormade commands.

Forth's multi-level structure also allows time-critical and machine-dependent modules to be written in assembly language. Forth's standardization and practical portability combine with the above features to make it the ideal communications language for developing applications.

Modifying Forth towards communications

In the application described in this paper, five different kinds of practical extension were made. Three of these are obvious practical tools rather than language extensions, and they are listed here without further comment:

- machine code primitives for sending bytes and strings (using the XON/XOFF protocol), and for receiving bytes

- machine code primitives for receiving strings in a buffer (using the XON/XOFF protocol)

- buffer housekeeping (clear, reset, inspect, copy text to buffer).

The two remaining developments are concerned with

- new definitions specifying data structures and actions for sending particular strings and monitoring answers received (regarding timing and appropriateness), and

- finite state machine concepts (defined later, alternatively known as the finite state description method, see Landau, 1982) for use as language flow control structures during machine-machine dialog.

Both the above were implemented using the <BUILDS DOES> structure of FIG-Forth. They can presumably be modified to the Forth-83 Standard.

As implemented, these two extensions are both within the existing possibilities of the Forth language/operating system. However, the last extension is of further interest because it was used largely instead of the usual control structures IF-ELSE-THEN and BEGIN-WHILE-REPEAT. (DO loops and BEGIN-UNTIL were generally not used.)

That is, the last extension represents a valuable alternative in (what would otherwise have been) large slabs of application code, compared with major control structures commonly used in Forth. It may be regarded as a generalization of both the CASE-structure and Forth's address interpreter (Ting, 1980).

The finite state machine concept is the theoretical basis for recent advanced work on reliability in communications. Gouda (1984) proves necessary and sufficient conditions for perfectly reliable communications between two communicating finite state machines. In connection with the new implementation of MODIS, an attempt is being made to use

Gouda's criteria for verifying reliable communications protocols between the processes arising in solving the model, at the Central Bureau of Statistics in Oslo.

Finite state machines may be defined in several ways. Here it is convenient to present them as alternatives to the usual block-oriented control structures such as IF-ELSE-THEN, and BEGIN-WHILE-REPEAT, functioning as described below. Forth's ability to address individual named modules to be executed later allows us to construct generalized switching and looping systems, which are driven by a small number of state variables and described by easily readable tables (see Landau, 1982). A **finite state machine** can then be defined as consisting of

a table of names or addresses of new modules which can be executed, pointed to by the elements of

another table selecting actions to be performed, based on the combined values of a limited number of state variables (usually two), and

a driving mechanism which loops repeatedly, using the two tables to execute a sequence of actions based on the state variable values as these change.

This scheme is very flexible, as any executed module (Forth word) may alter the values of the state variables and thus the flow of control. In general, reliability is obtained by strictly limiting the number of state variables and the number of modules which may alter their values. When this is done, the possible side effects which execution of a module can generate are in still principle dramatic (and potentially useful), but in practice nearly all possible events are neatly documented in the tables. Reliability can therefore be verified by exercising the limited number of possible program paths inside each separate module (Forth word).

While finite state machines are mentioned often in the computer science literature (particularly in connection with evaluation of parsed expressions), it is generally stated that such tables will be too large for the method to be of any practical use on a large scale, due to the number of possible states generated as elements of the Cartesian product of the sets of values which can be taken on by the state variables. This is wrong, however, as the writer has verified by constructing and selling a fully operational multi-user invoice, order and inventory system based to a large extent on finite state machine principles. Quite simple modifications solve the size problem usually referred to, while retaining almost all the benefits (particularly reliability). In a communicating finite state machine there is in any case only a small number of possible states, and the size problem can in any case be ignored.

There are several alternative ways of implementing finite state machines in Forth, reminiscent of (and partly derived from) the competition in the journal Forth Dimensions for the CASE-statement. So far in the MODIS input project described here, a simple method has been used, in which the first table is defined by a CASE-statement simply listing Forth words to be selected and executed according to the value of a single state variable. These are not however executed in natural sequence. The state variable is normally given a new value by each listed word in the CASE-statement when it is executed, fixed at the time the word is compiled.

This would give a fixed sequence, except for the fact that the single state variable is modified in some of the listed words, at run time, according to events.

In this simple implementation, the second table mentioned in the definition above of a finite state machine consists of data elements compiled into the individual Forth words named in the first table, so that it is spread between these words rather than collected in one Forth definition. Since these words are themselves defined on a couple of screens in single lines, the source code does have a readable table structure, as shown later in Figure 2.

In Figure 1, the Forth screen shown corresponds to the first table in the definition of a finite state machine.

The next two screens in Figures 2 and 3 correspond to the second table. (These screens refer to short words whose definitions are not shown here, related to the application.)

Figure 1

```
( branch for outer shell                          dkw-08/16/85 )

CASE: c-ACTION  ( n -- )              (   -1   exit, DON'T CALL)
                tty                   (    0   TTY, password )
                send-FI-ST            (    1   )
                send-ESC              (    2   login entry    )
                send-S-S-B            (    3   )
  send-N-N         (   4   )    send-FERD             (   5   )
  send-LOGOUT      (   6   )    INN                   (   7   )
  Name             (   8   )    TRE                   (   9   )
  Struct           (  10   )    Pred                  (  11   )
  Data             (  12   )    Open                  (  13   )
  Eval             (  14   )    Dxtest                (  15   )
  Cards            (  16   )    De-fi                 (  17   )
  Filename         (  18   )    Lesefil               (  19   ) ;
```

Figure 2

```
( sending and receiving strings 1                 dkw-08/16/85 )

   ( defining questions, answers to be tested for... )
   ( ...and new state variable settings afterwards )

 ( old new <-STATES)
   ( state sends   receives  max ?cr )

 ( 0 )  1   olav     prompt    100 cr SEND-STRING send-PASSWORD
 ( 1 ) 17   FI-ST    file:      20 cr SEND-STRING send-FI-ST
 ( 2 )  3   esc      login:    100 00 SEND-STRING send-ESC
 ( 3 )  0   s-s-b    password:  10 cr SEND-STRING send-S-S-B
 ( 4 ) 13   n-n      direktiv  150 cr SEND-STRING send-N-N
 ( 5 )  6   ferd     prompt    100 cr SEND-STRING send-FERD
 ( 6 ) -1   logout   ok-out:    50 cr SEND-STRING send-LOGOUT
```

Figure 4 contains elaborations of three words in Figure 3 which illustrate the flexibility of the method, to send a variable string for transferring data.

Figure 3

```
( sending and receiving strings 2                  dkw-08/16/85 )

 (  7 )  8 inn:    enavn?:    50 cr SEND-STRING INN
 (  8 )  9 COMBUF  valg?:    150 cr SEND-STRING send-Name
 (  9 ) 10 tre:    tperiode:  50 cr SEND-STRING TRE
 ( 10 ) 11 COMBUF  engere:   150 cr SEND-STRING send-Struct
 ( 12 ) 12 COMBUF  odevis:   150 cr SEND-STRING send-Pred
 ( 13 ) 18 open:   endres:   150 cr SEND-STRING Open
 ( 14 ) 15 eval:   enavn?:   150 cr SEND-STRING Eval
 ( 15 ) 16 dxtest: ltat:     150 cr SEND-STRING Dxtest
 ( 16 ) -1 ast:    direktiv  150 cr SEND-STRING Cards
 ( 17 ) 4  de-fi:  prompt     50 cr SEND-STRING De-fi
 ( 18 ) 19 k:      leses:    150 cr SEND-STRING Filename
 ( 19 )  5 null:   direktiv  150 cr SEND-STRING Lesefil
```

Figure 4

```
( Elaborations, sending and receiving states     dkw-08/16/85 )

: Name emptybuf N-FILE dx @ name 10 ->Bufftext send-Name ;

: Struct  emptybuf 1 S->D ->Buffnum   ( frequency yearly )
       Y-FILE  1 year @   S->D ->Buffnum  ( 1st year )
        0 year @ year @   S->D ->Buffnum  ( final year )
       send-Struct ;

: Pred  emptybuf
   P-FILE 0 pred @ 1+ 1 DO I pred @ S->D ->Buffnum LOOP
   send-Pred ;
```

The screen in Figure 5 contains a simple version of the driver referred to in the finite state machine definition.

The single line definitions in Figures 2 and 3 utilize the language extension SEND-STRING (implemented using the defining words <BUILDS and DOES> in FIG-Forth, not shown). SEND-STRING allows us to specify sending and receiving actions of a general nature (defined elsewhere), and a new value for the

Figure 5

```
( outer shell communications FD                         dkw-07/31/85 )
O VARIABLE o-state
: test?  o-state @ 18 = test @ 0= AND IF -1 state ! ENDIF ;

: com     BEGIN  state @ -1 >
          WHILE  state @ o-state !
                 state @ c-ACTION     ( ..branch to next action )
                 TTYk      ( ..look for F3 function key, -> TTY )
                 test?         ( ..switch testing or production )
          REPEAT please-send ; ( ..to AVOID LOCKING THE PORT ! )

: COM            INIT-COM1  2 state !  ( try to LOGIN ) com ;
```

state variable (i.e. the next action to be taken in the sequence if not over-ridden). While the SEND-STRING concept approaches the concept of a node in the graph of a communicating finite state machine as defined by Gouda (1984), it is not identical with it and more work remains to be done in this direction. SEND-STRING is relatively simple-minded. It sends a string several times until it either does or does not receive a specified appropriate string as a response. If it does, it sets the state variable to its usual new value, and returns to the finite state machine driver (Figure 5). If not, it sets a different state variable value leading the finite state machine to break to an exception strategy.

At present this consists of telling the operator to log out and restart the whole machine-machine sequence, but as noted in the next section, a major point of interest is to develop more subtle exception strategies which will allow the process to try other possibilities to get the dialog going again.

Further developments

It is planned to extend this scheme to use a 2-way table based on two state variables, one of which (as previously) indicates the current Forth word in the CASE: statement (Figure 1) being executed, while the other identifies a variety of responses from the other machine. To conform with Gouda's (1984) definitions, the elementary concept of action will either send or receive a message but not both (SEND-STRING sends and waits to receive a predetermined answer). This will enable the finite state machine to specify reactions to many alternative answers (one of which is a residual category). These plans are also based on the writer's experience using 2-way tables to handle user interactions, in the multi-user order, invoice and inventory system referred to earlier.

The benefit to be obtained from following Gouda's graph-theoretic definitions is that it is then possible in principle (and, we hope, in practice) to prove perfect reliability for some particular communications protocol between the various model solution processes, using his results, in the sense that the communication process cannot then deadlock or transfer data incorrectly without correction following. In practice, then, any communications failure will show itself as either an error report from the communications process, or a visible failure in the communications process itself. This is an essential step to allow decentralization of large applications.

References

GOUDA, M.G. (1984) Closed Covers: To verify progress for communicating finite state machines. IEEE Transactions on Software Engineering, Vol SE-10, No. 6, November 1984, pp.846-855.

LANDAU, J.V. (1982) Hardware-oriented state-description techniques. Ch. 16 in D.F. Stout, ed., Microprocessor Applications Handbook, McGraw-Hill, 1982.

MAY, D. & TAYLOR, R. (1983) OCCAM. INMOS Ltd, Whitefriars, Lewins Mead, Bristol, England.

MYERS, G.J. (1979) The art of software testing. Wiley, N.Y.

TING, C.H. (1980) Systems guide to fig-Forth. Offete Enterprises. Available from Mountain View Press, California.

WITT, M. (1983) An introduction to layered protocols. BYTE, September 1983, pp. 385-398.