Command Languages, C. Unger, Editor.
North-Holland Publishing Company (1975)

PROGRAMMING LANGUAGE EXTENSIONS WHICH RENDER JOB CONTROL LANGUAGES SUPERFLUOUS

Jørn Jensen and Søren Lauesen
A/S Nordisk Brown Boveri, Copenhagen.

Abstract

A job control language need not be a "new" language, but can be embedded in most existing programming languages. As an example, the necessary extensions for Algol are outlined and several applications of them are given.

Some aspects of the implementation and the requirements to the operating system are discussed. A complete list of all requests from job to operating system is given.

The approach gives several advantages: The Algol user will not have to learn a new syntax and nearly no new concepts. Still his job control language is much more powerful than any existing job control language. From a theoretical point of view, the approach clarifies the basic concepts and mechanisms of a job control language.

The main drawback seems to be a slightly more elaborate writing of the commands (you may also call it more readable). But this is completely independent of the job control purpose, and can be traced back to slight inconveniences in the syntax of Algol, for instance the strict matching of actual and formal parameter list.

All the existing system programs like compilers, editors, and linkers could relatively easy be made available from Algol, because the proposed call mechanism allows each system program to use its own core allocation method and its own strategy in communication with the operating system and the drivers.

CONTENTS

1. INTRODUCTION

This report discusses the interface between job and operating system. It also discusses which language facilities the user needs to make efficient use of the interface.

We conduct the discussion in the opposite sequence. First (in Chapter 2), we use Algol as a model language and extend it slightly to make it suitable for job control. Second (in Chapter 3), we discuss the requirements this makes on the operating system and we end up with a list of the requests a job can issue to the operating system. Finally (in Chapter 4), we show how the drivers can be simplified and the different file access methods put into procedures of the job.

1.1. Job execution

When a user logs in on the system, the operating system will ask for his user name and password (project number). After proper checking, the operating system creates a job process for the user and lets the job process execute an initial program. The operating system gives the initial program access to a primary input stream (lines typed in on the terminal) and a primary output stream (output to the terminal). The rest of the job execution is controlled by the initial program and the contents of primary input.

In the case of a card batch job, nearly the same thing happens, but primary input is the cards of the job, primary output a print file.

1.2. Program file and program parameter

We will assume that the system contains a set of files each identified by a file name. Some of these files are program files which by means of a program call can be loaded and executed. A Program call must specify a list of program parameters.

A particular program file is the initial program and it requires two parameters: the primary input stream and primary output stream. Other examples of program files are compilers. We will assume that they as a standard require 5 parameters:

1. A source stream containing the source text.
2. An output file name specifying the name of the object file.
3. A message stream to contain diagnostic messages and possible listings.
4. A mode integer which specifies whether listing should be produced, etc.
5. A result integer to contain information of how the compilation proceeded (has the output file been successfully created, were syntax errors detected, etc.).

Further we assume that the object file from the compilation has to pass through a linking (loading) phase where it is combined with precompiled parts from a library. We will discuss this a little more in section 2.3 and 4.1.

The program file containing the linker requires also 5 parameters like those above, except that the first parameter is the file name of the object file to be linked.

We define three types of program parameters: integer, file name, and stream. They are all exemplified above. You should notice the difference between a file name and a stream. A stream is an open file, with buffers allocated, and ready for sequential input or output. For instance, if two programs are called in succession with the same file name as an input parameter, they will get exactly the same data from that parameter. But if they are called with the same stream as an input parameter, they will get succeeding parts of the file.

For simplicity, we assume the following implicit type conversion for program parameters: If a program specifies a program parameter of type stream, the corresponding actual parameter may be of type stream or type file name. In the latter case, an implicit file opening and buffer allocation is assumed.

A system like OS/360 deviates from the principles above in the way access methods and buffers are specified. Above, we always assume that a program (or the implicit type conversion) can infer a proper buffering and access method from the file name alone. We will elaborate on that subject in Section 4.2.

2. AN ALGOL BASED COMMAND LANGUAGE

2.1. An Algol interpreter as the initial program

We will now discuss a system where the initial program is an Algol interpreter, which reads an Algol program from primary input, compiles it, executes it, reads the next program from primary input, and so on. The compiled program is executed (called) with two program parameters of type stream: "in" which actually is primary input and "out" which actually is primary output. A third parameter "mode" (of type integer) is supplied for reasons explained in Section 2.4.

As shown in example 1, such a system is well suited for simple Algol jobs.

We now postulate the existance of a standard procedure "execute" which can call a program file with a list of program parameters and execute it. The Algol statement for this is:

execute(<name of program file>,<list of program parameters>);

A program parameter of type file name appears simply as a string in the call, while streams are introduced as new Algol quantities.

This standard procedure is used in Example 2 to compile, link, and execute a Fortran program. The Fortran compiler has the file name `ftncomp`, the linker the file name `link`. All the job control statements" are statements of the Algol program from begin to end.

These examples are hardly surprising, but one advantage of the approach appears when the user decides to replace the job control statements by a single statement. That is, he wants to write his job as in Example 3. This requires the existence of a standard procedure "fortran" or a user defined external procedure as shown in the example. The procedure "fortran" has been improved relative to Example 2, so that it skips linking and execution if the compilation caused troubles. A further standard procedure "cancel" is invoked to cancel the intermediate object file from the compilation.

Example 4 shows another application where the Fortran program is executed ten more times with new sets of data.

The real advantages of the approach appear when we discuss high level job control commands like these:

A job control command which has a set of source file corrections as input and returns only the result of four test runs. It updates version numbers and takes safety copies automatically.

A job control command which executes a sequence of administrative programs like input conversion, sorting, merging, print out. It takes care of back-up versions, reporting of file contents, reruns, etc.

We hope that the reader can imagine that such commands are relatively simple to implement with the tool outlined. The key point is that streams, programs, and files can be handled by the programming language whith the same flexibility as integers and reals.

Example 1: Simple Algol job

This example shows the contents of primary input for a simple Algol job.

```
job <user identification>
begin real a,b; ...                    } The Algol program, which is
    read (in, a,b,...);                } read and compiled by the
    ...                                } initial program of the job.
    write(out, ...);
end;
<data>                                 } The data which is read
                                       } by the compiled program.
```

Example 2: Simple Fortran job

This example shows the contents of primary input for a simple Fortran job - still with an Algol interpreter as the initial program.

```
job <user identification>
begin integer result;
    execute('ftncomp', in, 'obj', out, 0, result);
    execute('link', 'obj', 'prog', out, 0, result);
    execute('prog', in, out, 0);
end;
subroutine ...
...
end
<data for Fortran program>
```

Example 3: Fortran job with compile-link-go procedure

This job uses a precompiled procedure "fortran" to achieve the effect of Example 2.

```
job <user identification>
begin                                  } could be abbreviated to:
fortran(in,'prog',out,0);              } fortran(in,'prog',out,0);
end;
subroutine ...
...
end
<data for Fortran program>
```

The procedure "fortran" may have this appearence (improved slightly relative to Example 2):

(Example 3, continued)

```
external procedure fortran(source,object,message,mode);
stream source, message; string object; integer mode;
begin integer result;
    execute('ftncomp',source,'work',message,mode,result);
    if result = 0 then
       execute('link','work',object,message,mode,result);
    cancel('work');
    if result = 0 then
       execute(object,source,message,mode);
    end;
```

Example 4: Fortran job with repeated execution

This job compiles and executes a Fortran program, and next executes it ten more times with new sets of data.

```
job <user identification>
begin
    fortran(in,'prog',out,0);
end;
subroutine ...
...
end
<data for Fortran program>
begin integer i;
    for i:=1 step 1 until 10 do
    execute('prog',in,out,0);
end;
<data 1>
<data 2>
...
<data 10>
```

Example 5: Use of templates for parameter specification

By allocating meaning to Algol's fat commas, the job of Example 3 could be written like this:

```
job ...
fortran(in)object:('prog')message:(out)mode:(0);
subroutine ...
```

or like this:

```
job ...
fortran(0)source:(in)message:(out)object:('prog');
subroutine ...
```

2.2. Algol shorthand, other interpreters

It is clear that the use of Algol introduces a slightly move elaborate writing of the "job control commands".

For instance, Algol 60 demands that a program must be a block or a compound statement, i.e. start with a begin and terminate with an end. It would be reasonable to allow any statement as a program, so that the begin-end could be omitted in Example 3.

More troublesome is Algol's strict matching of actual and formal parameter list. If by a mistake two parameters are reversed in the execute call, the strangest things may happen. A method used by many control languages is to specify for an actual parameter which formal parameter it should match. In an extended Algol language, this could be accomplished by giving meaning to the parameter comments, the "fat commas". The job of Example 3 could then be written as in Example 5. This also opens the possibility for default specifications, like those of Example 6.

It should be clear that most command languages are developed with facilities like these in mind. Normally the command languages lack the following facilities found in high level languages: general conditions and loops, variable declarations, procedure (or macro) declarations. But these differences have nothing to do with the interface to the operating system. They are rather a question of notational convenience.

In view of the principles of job execution explained in Section 1.1 above, the operating system need not care whether Algol is used for job control or whether an ordinary control language is used. It only depends on whether the initial program is an Algol interpreter or an interpreter for an ordinary control language. Of course, the initial program could also be an interpreter for Fortran, Cobol, or Basic.

2.3. String operators for file name specifications

Many control languages make implicit extensions of file names to distinguish between various representations of a program. For instance, when compiling the source file "pr", the object file may automatically be called "probj".

This may be a dubious facility, but if we insist to build it into our "fortran" procedure of Example 3, we will need some string operators in our extended Algol language. The "fortran" procedure now requires a file name for the source instead of a stream. It is shown in Example 7.

For a compilation or a linking it would also be convenient to specify a list of source files to be treated concatenated as a single stream. This facility is more troublesome, but one solution is to allow a file name parameter to be a list of file names, so that the following call is legal and compiles "source1" followed by "source2":

```
execute(`ftncomp`, `source1-source2 ,  obj , out, 0, result);
```

Again, the availability of string operators and string variables would make this approach suitable for use in procedures like "fortran".

We could of course abandon such automatic concatenation and insist that files are concatenated explicitly by means of a program (or a procedure) which takes two input streams and concatenates them into one output file.

2.4. The Algol interpreter, dynamic change of command language

In the examples above, we have assumed that the initial program is an Algol interpreter. If we have available an Algol compiler analogous to "ftncomp", we could compose the Algol interpreter as in Example 8.

We have specified the program parameters as we would have specified procedure parameters. This possibility is needed if we want to write general systems programs in our extended Algol. In the preceding examples we have assumed the following default specification of the program parameters:

```
program(in,out,mode); stream in,out; integer mode;
```

This corresponds to the way the Algol interpreter calls the final program. Notice, that the "program name" is the file name of the program file, and thus need not be specified in the source text.

With this Algol interpreter, consider the effect of Example 9. The first program "mode:=1" will change the integer "mode" in the Algol interpreter. When the following programs are compiled, the Algol compiler will get the mode-parameter 1, which is assumed to produce listing of the source text. Thus, we obtain the effect of a job log which contains a listing of all "job commands" executed.

Consider next Example 10. Here we call the program file "algolint" specifying the files "subjob" and "subout" for "in" and "out". Thus we will have the "commands" in "subjob" executed, and when "subjob" is exhausted, we return to executing the commands of primary input.

The new incarnation of the Algol interpreter works on other input and output streams, and we could call the technique a recursive change of the current input and output. If we had a Fortran interpreter, it could be called the same way, and we would then obtain a change of the current command language.

Example 6: Use of templates and defaults

If a parameter is not in the actual list, a default specification could be used, like this:

```
job ...
fortran( )object:('prog');
```

where "fortran" looks somewhat like this:

```
external procedure fortran(source,object,message,mode);
stream source default in, message default out;
string object; integer mode default 0;
begin integer result; ...
```

Example 7: Automatic file name extensions

The following "fortran" procedure requires a file name for the source. It delivers the object program in a file with a name derived from the source file name. An operator for string concatenation is used.

```
external procedure fortran(source,...);
string source; ...
begin integer result;
    execute('fntcomp', source, source concat 'obj',...);
    ...
```

Example 8: The Algol Interpreter

This example shows an Algol interpreter based on a traditional Algol compiler. We have adopted a convention of specifying the required program parameters.

```
program (in,out); stream in,out;
begin integer result, mode;
    mode:=0;
rep:  execute('algolcomp',in,'wrk',out,mode,result);
    if result =0 then
       execute('link','wrk','wrkl',out0,result);
    cancel('wrk');
    if result =0 then
       execute('wrkl',in,out,mode);
    if result ≠"end of file" then goto rep;
end;
```

3. INTERFACE BETWEEN JOB AND OPERATING SYSTEM

In this chapter we will discuss what happens on the interface between the job process and the operating system when jobs like those of Chapter 2 are executed.

3.1. Resource allocation

A job requests resources from the operating system when it calls a program and when it opens or creates a file. The basic operations for this should be available in the programming languages, although file opening and creation in the examples were done entirely inside the compilers and linkers.

The set of resources needed for a given operation may vary from system to system. For instance, file opening may involve special resources like file control blocks, buffers, exclusive writing permission, etc. In some systems the buffers are part of the job's core store, in others they are considered special resources by the operating system.

In the following we will just assume that a set of resources are necessary for each operation, without further specification.

When the job process is created, it has an initial set of resources which enables it to interpret at least some statements. Other statements may request further resources from the operating system.

The examples of Chapter 2 assume that we have a dynamic resource allocation. For instance, when the job has created a file, it may later create another file without cancelling the first. And no statement of the total set of files has been given initially or at the beginning of the "job-step".

Under operating systems with a static resource allocation, the job-process must state the resources (or even worse - the files) needed for a job-step at the beginning of that job-step. (We can define the beginning of the job-step as a point where the job holds only the initial resources). The operating system then allocates all the resources before the job is allowed to proceed.

If the resource specification has to be very detailed (including for instance file names), it could be simplified by procedures, just as we simplified compilation and linking in Example 3. This assumes only that the initial resources are sufficient to interpret calls of such procedures.

For an operating system with a dynamic resource allocation, the limit for some of the resources must be stated initially or between job-steps. Such limits correspond to the claims used in for instance the Banker's Algorithm (ref. 4), although it is not mentioned in the literature that the job could change its claims freely between job steps. Within the limit, the job may then request resources dynamically.

The main advantages over fixed resource allocation seem to be that the limits need not be stated so detailed as the resources and a certain overstatement has less influence on the turn-around time. As a result, the limits can often be stated in a standard fashion or the initial limits can be made sufficient.

3.2. The program call mechanism

The program call mechanism proposed above is recursive. Under operating systems with a limited core area for each job, this seems to cause trouble. However, the execute-procedure postulated above could proceed as follows: First it creates a working file and saves the proper core area parts here. Next, it relocates the parameter list to a standard form and position. Finally, it opens the program file, loads the program in the standard way and enters it. The program returns after execution to a resident part of the execute-procedure, which then restablishes the core area.

This implementation of the execute-procedure does not make any special requests to the operating system. It uses only the ordinary operating system requests to create a file, open a file, and perform input/output. But there might be two reasons to replace this implementation by special operating system requests.

One reason is file protection in case the system distinguishes between read, write, and execute protection. (Above, ordinary input was used to load the program file).

Another reason is self-protection, i.e. protecting the user's job against the user's own programs. When a program runs off the track, it may destroy the core area of the job. But in a properly protected system the operating system is always unharmed and may force a program return. Also, the operating system could make a copy of the erroneous core area on a file, thus allowing subsequent job statements to analyze or print out the core area. This gives much better possibilities than present day octal dumps.

The operating system could force such program returns when the job exceeds its resource limits or in connection with the time supervision explained in the next section.

Example 9: Job execution with listing of commands

The following job will list all "job commands" as they are executed.

```
job ...                  { strictly:
                         { begin mode:=1 end;
mode:=1
comment mode=1 is assumed to mean "list source text";
fortran(in,'prog',out,0);
...
```

Example 10: Change of current input and output

The following job will execute the "commands" found in the file "subjob" with output to the file "subout". Later job commands may then process the contents of "subout".

```
job ...
execute('algolint','subjob','subout');
execute('edit','subout',...);
...
```

3.3. List of operating system requests

We are now prepared to make a list of the requests a job can issue to the operating system. These requests should be directly available in the major programming languages. The internal form of the request may vary from system to system.

File creation: The request specifies the file name, the volume on which the file is to be created, the position within the volume (implicit for common volumes like drum and disc), the access method for the file data, the size of the file (where needed).

File cancel: The request specifies the file name.

Change of protection: The request specifies the file name and the new protection situation, permanency, etc.

File opening: The request specifies the file name and the stream. The stream comprises a set of buffers, control blocks, etc. It also holds data corresponding to a "file Lookup" (see below).

File closing: The request specifies the file name and the stream.

File lookup: The request specifies the file name. Data corresponding to the file creation and protection situation is returned.

Input/output: The request specifies the stream, an operation (input/output/upspace/rewind/...), a maximum physical block size, and - for random access files - a position within the file. These requests are rather to drivers than to the central operating system in order to save some of the overhead. Notice, that the driver is not concerned with the file access method, which is handled by procedures in the job process (see Section 4.2).

Program call: The request specifies the program file name and the program parameters.

Program return: No parameters needed. The return from the initial program corresponds to job termination or logout.

Time supervision: The request specifies a maximum period to program return. If the program return is not issued in due time, the operating system forces a program return as explained Section 3.2. This suffices for debugging based on post mortem analysis of the core area.

Resource statement: The request specifies a resource and an amount of it, either as static allocation or as a claim (see Section 3.1).

Core store allocation: The request specifies an extension or reduction of the core store area of the job.

4. FILES AND ACCESS METHODS

4.1. File description and directory

When a job makes a request to create a file, the operating system will enter a corresponding file description in a directory. The following information seems to be needed in the file description:

File name as used for identification.

Volume on which the file is kept. This includes a volume type allowing the operating system to select a proper device for mounting the volume, and a driver type allowing the operating system to select a proper addressing of the physical file blocks.

Position within the volume.

Size of the file if needed.

Protection situation describing which users have which access to the file, the lifetime of the file, etc.

Access method, which is only used by the job, not by the operating system (see below).

It is advantageous if the directory is kept in files which are readable for all users. In this way selective directory listing can be produced by ordinary programs. Note that even though the protection information is publically available, the protection can still be maintained. The method is to store the passwords (or the like) in another file only available to the operating system. Then a user cannot readily get the information needed to log in with another user's identification.

4.2. Drivers and file access methods

When a job has opened a file, it can make requests to a driver in order to read or write physical blocks. The physical blocks are transferred between the file and the buffers of the stream.

It is not necessary to let the drivers care about the contents of the file, whether it is a text, a sequence of logical records, a hash-ordered file, etc. The corresponding different access methods can be implemented as procedures in the job. The procedures will use the driver requests when new physical blocks are needed.

If a program is prepared to access files in various ways, it will need to know the proper access method for a given file. This it can obtain by means of the request "file lookup" or from the stream description. It could not reasonably obtain it from the physical file blocks, because the proper access method is not known yet. And it is too troublesome to let the user specify it.

System programs should of course always check their own requirements against the file's access method.

4.3. Self-protection of primary input/output

As explained in the preceding section, the job can exert some di-

rect control over the buffers of a stream. For instance, an erroneous program can destroy a buffer part containing output previously produced. Debugging under such circumstances is very difficult, but the trouble can be remedied by making primary input and output self-protected. This requires that the associated drivers handle only one line of text in an input/output request.

5. CONCLUSION

The report has demonstrated that it is possible to replace the command language by extensions of one or more normal programming languages. These extensions need only a rather simple interface to the operating system, and the interface is the same for all programming languages.

The user is relieved from learning a separate command language, and still he is able to compose his own job control statement with any degree of complexity or use system control statements composed in the same way.

The programming language takes care of the different file access methods - either explicitly or implicitly. Consequently, only simple access methods need to be built into the operating system and the drivers. Hopefully, the user will also benefit from the increased reliability of the operating system resulting from the simple interface.

The run time efficiency of the approach depends mainly on the speed of the compiler for the programming language. This is important if several small commands (programs) are compiled, but most jobs will perhaps contain a few composite commands like Example 3.

Another problem is that the initial resources of the job must be sufficient to compile the commands. This may cause troubles for many operating systems where the initial resources just are a set of tables in the operating system.

These problems seem to point to a general purpose, incremental language with few resources for compilation. A language like Basic might be a candidate in computers where only slow compilers for Algol, Fortran, PL/1 are available.

Perhaps the most important part of the report is the clarification of the interface to the operating system and the underlying concepts in the command language.

REFERENCES

1. D.W. Barron and I.R. Jackson: The evolution of job control languages. Software - Practice and experience, Vol. 2, p 143-164 (1972).

2. W.A. Clark, G.H. Mealy, and B.I. Witt: The functional structure of OS/360. IBM Systems Journal, no. 1 (1966).

3. G. Cuttle and P.B. Robinson (eds.): Executive programs and operating systems. Mac Donald, London 1970.

4. A.N. Habermann: Prevention of system deadlocks. Comm.ACM 12,7 (July 1969), p373-377,385.

5. S. Lauesen: Program control of operating systems. BIT 13 (1973), p323-337.