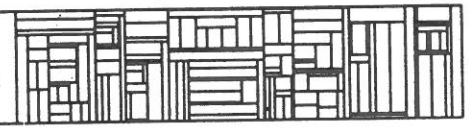
ISSN 0105-8517

KERNEL LANGUAGES

Brian H. Mayoh

DAIMI PB - 177
October 1984

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



KERNEL LANGUAGES

Brian H. Mayoh
Aarhus University

There are many programming languages; each have their advantages and disadvantages; none is completely adequate for all kinds of problems. In the ideal world a routine in any language should be allowed to call a routine in any other language. In practice most programs are written in a single language, but there is a trend towards more flexibility

- separate compilation;
- distinction between interface and body of modules (e.g. ADA packages, tasks, procedures and functions);
- use of library packages (e.g. PASCAL programs calling subroutines in a FORTRAN graphics module).

There is a need for a kernel language to link routines in different programming languages (see fig. 1). Because this kernel language should not be biased towards a particular kind of programming language, any parameter and type checking should be neutral and innocuous. At present the nearest approach to a kernel language is UNIX with its pipelines of software tools [KM], so its widespread popularity is not surprising.

Routine	Typical Call	Parameters
PASCAL program	from control language	files
PASCAL function	from program or another routine	variables,expressions, routines
ADA procedure	from control language or another routine	variables, expressions
ADA task entry	from another routine	variables, expressions
PROLOG predicate	from control language or another predicate	terms
Code subroutine	from control language or another subroutine	addresses, values

Figure 1. Examples of routines

In the ideal world programmers should not have to learn a complex and clumsy job control language before they can run their programs on a computer. Consider the current trend towards integrated software environments for developing programs at graphical work stations. Sometimes these environments are designed for a particular programming language and a modified version of the language is used as the control language (ADA for MAPSE,PROLOG for PSI). Sometimes these environments are designed to be language independent and the control language is simple [H]. There seems to be a case for replacing control languages with a kernel language that can distinguish between

- messages to and from the user (integers, strings, lists and the like);
- files of messages that can be kept in back up memory;
- routines that can be kept as code and executed on demand.

Such a kernel language should be powerful enough to serve as an "intermediate" language for portable compilers (P-code for PASCAL, A-code for ADA ....). In this paper we present the design of a kernel language KL that can link routines in other languages and replace both control and intermediate languages. The language KL is suitable for the casual user, who knows no other language, because it is both simple and powerful.

In section 1 we distinguish between KL system and library variables, and we describe how library variables are arranged in modules. In section 2 the KL way of calling routines is explained; in section 3 we describe how calls can be pipelined and how input and output can be redirected; in section 4 we focus on KL communications with the user. A general way of building new routines from old is described in section 5; since this way of combining routines captures recursive flow diagrams and there is a general value assignment command, KL is powerful enough for the casual user. One argument for asserting that KL is simple enough for the casual user is that the complete syntax of KL is given by the twenty rules in sections 1 to 5. The strongest argument for this assertion is the concise PROLOG implementation in section 6. This implementation is a straightforward modification of the formal denotational semantics of KL in the original version of this paper.

In section 7 ports are introduced so that KL can handle nondeterminism and multiple I/O. Since ports in concurrent processes can be linked, we can describe nets of communicating routines in section 8. The KL approach to failing, interleaving and backtracking routines is described in sections 9 and 10; The KL approach to local variables, declarations and scoping is described in section 11. The last section gives a PROLOG implementation of the KL features in sections 7-11.

#1 Variables, Modules and Libraries

In our kernel language KL we have an unlimited supply of system variables and the simplest KL command

(R1) variable := variable

has the meaning: assign the value of the right variable as the value of the left variable. This assignment command can also be used with library variables. At any time during a KL session there is a library of modules, and each module has a number of variables. The modules in a library form a tree, and the library

variables are the leaves of this tree (see fig. 1.1). As the paths in the tree identify the library variables, KL can have the syntax rule

(R2) variable ::= {identifier .} * identifier

where identifiers before "." refer to modules. Note that library variables have a "." and system variables have not. Note also that R1 gives a KL syntax rule

(R1') command ::= variable := variable

although it was presented as particular KL command. We will always confuse abstract and concrete syntax in this way, because particular KL implementations may disagree on concrete syntax and we do not want arbitrary conventions about escape characters, graphical presentation of commands and the like to interfere with our presentation of KL.

How can one create and destroy library variables in KL? The destruction command is

(R3) variable :=

and this also destroys modules which become empty when "variable" is removed (see fig. 1.1).

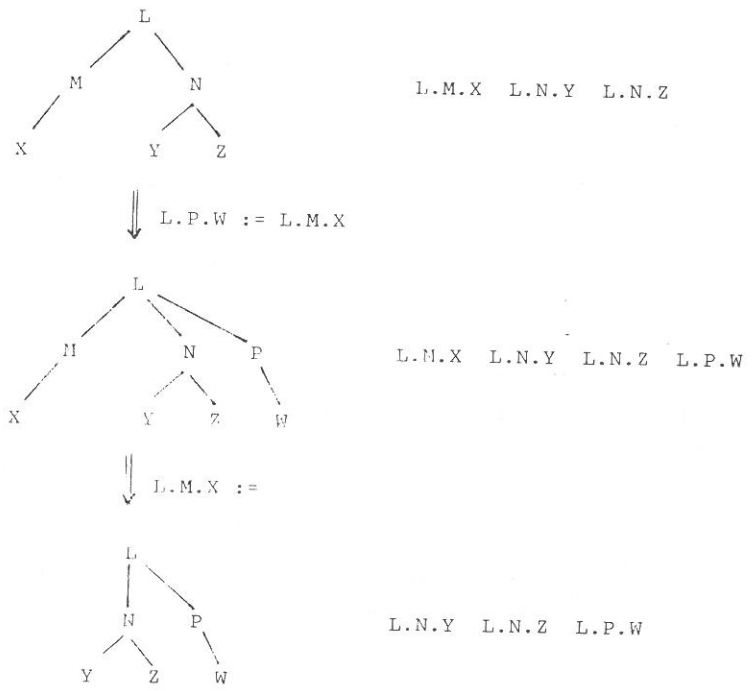


Figure 1.1. Trees of Modules and Library variables

The simplest way to create a variable is to use it in a KL command. When a new variable or module is mentioned in a command, the library is modified to accomodate the variable or module (see fig. 1.1). The library can also be modified by the compiler for ADA and other languages, which allow separate compilation of "packages" of variables and routines. The compilation of such a package P may give a new module P and new library variables for each package variable and routine. Note that a compilation, which creates variables, may also give initial values to the newly created variables. Note also the close analogy between messages and routines in module variables and the "abstract data types" so prominent in the theoretical literature.

#2 Routines

If a kernel language is to have any chance of success, it must have a simple but general way of calling routines. In KL routines are called by commands like

(R4) variable(message)

If the value of "variable" is not a routine, the command (R4) has no effect on the values of KL variables, but an error message may be sent to the user in some KL implementations. If the value of "variable" is a routine, then the value consists of a code and an introduction. KL uses the introduction to interpret "message" in the command (R4), then it executes the code. The way messages are interpreted depends on the programming language used to define the routine. Let us leave this language dependent issue to a later section on definition of routine values and introduce a particular routine value EVALUATE. The introduction of EVALUATE tells KL not to interpret its message, and the execution of the code of EVALUATE causes

- computation of message-value by substituting values for variables in "message";
- computation of result-value from message-value, interpreting & in "message" as string concatenation and +, ×, -, / and brackets in "message" as usual.

Fig. 2.1 shows the execution of EVALUATE for particular messages.

message	message-value	result-value	when variable has value	
17	17	17		
2+3	2+3	5		
2+a	2+7	9	a	7
2+a	2+a	2+cat	a	cat
2+a	2+3×7	23	a	3×7
2&a	2&a	161	a	3×7

Figure 2.1 Evaluating particular messages.

The KL command for evaluating messages is

(R5) variable := message

and this command causes the result_value given by the call of EVALUATE to be assigned to "variable". Because a kernel language should accommodate functions as routines, KL also has the command

(R6) variable := call

for assigning the results of routine calls to system and library variables. In KL all routine calls either loop or they have a result, but the result of a procedure, predicate or program might be an artificial message like: SUCCESS, FAILURE, ABORTION, TIMEOUT ... This convention allows us to rewrite (R4) as a syntax rule

(R4') call ::= variable(message)

where "message" might be interpreted as a list of actual parameters separated by commas. In the next section we introduce more complicated calls than those allowed by (R4'). All calls are KL commands, and particular KL implementations may tell the result of the call to the user when it is not assigned to a system or library variable.

NOTE: EVALUATE has been defined in such a way that the meaning of (R5) does not conflict with that of (R1). In section 4 we reveal how the user can learn the result of evaluating a message; the command EVALUATE (2+3×4) is allowed by (R4) but KL interprets EVALUATE as a system variable.

#3 Sources, Sinks and Pipelines

Usually programs are written on the assumption that there are standard input and output files. A kernel language should allow redirection of input and output; if a routine assumes input from a terminal keyboard and output to a terminal screen, then the kernel language should allow the routine to take input from any source file and/or send output to any sink file. In KL we have the commands

```
(R7)          variable >—— call
(R8)     variable := variable >—— call
(R9)                                call ——> variable
(R10)    variable :=                call ——> variable
(R11)              variable >—— call ——> variable
(R12)    variable := variable >—— call ——> variable
```

If there is a variable "source" before >——, then input to the called routine is taken from "source" instead of the keyboard; if there is a variable "sink" after ——>, then output from the called routine is sent to "sink" instead of the screen; if there is a variable "destination" before :=, then the result of the called routine is assigned to "destination". Intuitively information streams from the source through the code to the sink.

This intuition suggests the introduction of pipelining in a kernel language. In KL one can avoid temporary files by writing commands like

```
(R13)    variable(message) >—— call
(R14)                          call ——> variable(message)
```

where (R14) corresponds to the UNIX pipeline mechanism. The effect of (R13) is

- the call is executed;
- when the call is complete, its result is the result of (R13)
- when the call needs input, "variable(message)" is executed until it produces output.

This is subtly different from the effect of (R14):

- the call is executed;
- when the call is complete, its result is taken as the result of (R14);
- when the call produces output, "value(message)" is executed until the output is consumed.

The reason why KL has both (R13) and (R14) is that pipelines must have a driver, a routine which determines how active the other routines in the pipeline must be. From the syntax rules

```
(R15)    call ::= variable(message) >—— call
(R16)    call ::=                       call ——> variable(message)
```

we see that the driver of a pipeline can be identified as the unique "variable(message)" where >—— changes to ——> (see fig. 3.1). Note also that (R15) (R16) permit pipelines to have sources and sinks different from keyboards and screens.

```
A() >—— B() >—— C()   terminates  when driver C terminates
A() >—— B() ——> C()   terminates  when driver B terminates
A() ——> B() ——> C()   terminates  when driver A terminates

source >—— A() ——> B() ——> C() ——> sink   has driver A
```

Figure 3.1. Pipelines and their drivers

The execution of a pipeline can be interrupted when one of its components consumes a value produced by its predecessor. The termination of any KL command can be interrupted; in a multi-user environment any KL command can be interrupted by the timesharing mechanism. Some KL commands have other interrupt points:

- commands with no source (R4, R6, R9, R10) can be interrupted after requesting a keyboard input;
- commands with no sink (R4, R6, R7, R8) can be interrupted after sending a screen output.

Particular KL implementations may send particular messages to the user at interrupt points. As the user may have several active KL commands, an interrupt message for a traditional terminal should indicate which command has been interrupted (see fig. 3.2).

```
X >—— 16          -- active command X:=A(...) receives input 16
L.N.Y ——> "dog"   -- active command L.N.Y:=B(...) sends output "dog"
>—— 7             -- active command C(...) receives input 7
X:="cat"          -- active command X:=A(...) sends result "cat"
```

Figure 3.2. Typical KL dialog with a traditional terminal

The reason why we allow many KL commands to be active concurrently, is because "a window for each active command" is the natural modus vivendi at a graphical work station (see fig. 3.3).

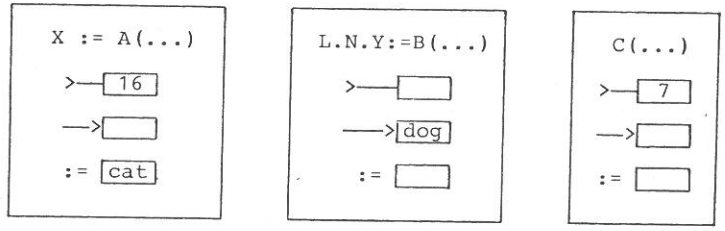


Figure 3.3. Typical KL dialog with a graphical work station

Usually a KL command becomes active as soon as it is entered, but we do impose the restriction: if the source, sink or destination of a command C is the source, sink or destination of an active command, then C is delayed. Particular KL implementations may impose stronger restrictions that further reduce indeterminacy and conflict, but it seems unwise to impose the strongest restriction: sequentiality.

#4 Communication with the user

There are four ways KL can send a value to the user

(1) Sending a file when a command has no explicit sink;

(2) sending the result of a call, when there is no explicit destination;

(3) sending a value when a command is interrupted;

(4) executing the command

(R17) variable ——>

What the user actually sees may vary from KL implementation to KL implementation. However we do recommend that the output form of a variable v reveals clearly if v contains a file, a routine or just a message value. Whatever the conventions of a particular KL implementation, the output form of a value is a message so (R17) coerces file and routine values into message values. Because KL has this coercion, it can also have the command

(R18) variable ——> variable

for delaying user output or coercing (the contents) of files and routines into call parameters.

Example 2

If the variable before>—— in (R20) has the value

```
PASCAL function f ..... end;
```

then (R20) uses the Pascal compiler to set the value of the variable after >—— to the compiled code for f with an introduction telling KL how to convert a functional value and an actual parameter message. The conversion rules depend on the particular Pascal compiler used by KL, but most Pascal compilers follow the rules in fig. 4.1. If the reader feels we have skated over the real complications of separate compilation like type checking, use of external variables and routines ..., she might consult [KMMN]

Concept	Cell	Contents	Code reference
local variable	top of stack	value	direct
local routine	-	-	direct jump
external variable	on stack	value	via display
external routine	-	-	direct jump
value parameter	top of stack	value from actual parameter	direct
variable parameter	on stack	address from actual parameter	indirect
routine parameter	on stack	address from actual parameter	indirect jump
functional value	top of stack	value to result	assignment

Figure 4.1 Code conventions of typical PASCAL compiler

Example 3

If the variable before>—— in (R20) has the value

```
PROLOG predicate member (X,S) has rules
member (X,[X|Y]).
member (X,[Y|Z]):- member (X,Z).
end.
```

then (R20) sets the value of the variable after >—— to code for member with an introduction telling KL how to convert an actual parameter message into two PROLOG expressions.

There are four ways KL can receive a value from the user

(5) receiving a file when a command has no explicit source;

(6) when a message is typed as parameter to a call;

(7) during a dialog after interruption of a call;

(8) executing the command

(R19) >—— variable

What the user actually sends may vary from KL implementation to KL implementation. However we do recommend that the answer to (R19) clearly indicates if a message, file or routine value is being sent. Whatever the conventions of a particular implementation, the input form of a value is a message so (R19) converts a message value into a file value, a routine value or another message value. Because KL has this conversion, it can also have the command

(R20) variable >—— variable

for delaying user input or compiling routines and programs.

Example 1

If the variable before >—— in (R20) has the value

PASCAL program P ..... end.

then (R20) uses the Pascal compiler to set the value of the variable after >—— to the compiled code for P with an appropriate introduction. In section 2 we described the role of "code" and "introduction" in routine values. If the program P has only INPUT and OUTPUT as external files, the appropriate "introduction" is empty because KL has other mechanisms for redirecting input and output; otherwise the "introduction" tells KL that actual parameters must be file names.

Example 4

If the variable before >—— in (R20) has the value

ADA package P is ..... end P;

then (R20) uses the ADA compiler to set the value of the variable after >—— to the message "ADA package P" and create a module named P. For each package procedure, function or task T there is a new library variable P.T containing an appropriate introduction. For each (initialized) package variable V there is an (initialized) library variable P.V.

Example 5

If the variable before >—— in (R20) has the value

ADA package body P is ..... end P;

then (R20) uses the ADA compiler to set the value of the variable after >—— to the message "ADA package body P" and add code to the procedure, function and task variables in the module P.

Every KL implementation should have some way for the user to stop the presentation of a command - e.g. by pressing a BREAK key on the terminal - and start a dialog with KL. During this dialog the user should be able to discover

- what modules exist
- what variables a particular module has
- the value of a particular variable
- the parameters of a particular routine
- the input conventions for a particular parameter of a particular routine.

The user should be allowed to end the dialog by naming a variable with a message value. If this value is added to the interrupted KL command before the user resumes its presentation, then KL will have a convenient macro mechanism. Whatever the dialog conventions of a KL implementation, they should not depend on whether the user started the dialog by interrupting the presentation of a command or the machine started the dialog by interrupting the execution of a command.

A powerful kernel language should not rely on other programming languages for defining routines, it should provide ways of constructing new routines from old. The simplest way of combining routines is the straight line sequencing we find in the command files of most operating systems. Suppose we can input the message

•—com1—>•—com2—>• ...•—comn—>•

to KL whenever com1,com2,...,comn are KL commands. In the same way that Pascal programs, Pascal functions, PROLOG predicates, ADA packages and ADA package bodies were converted to routines in the examples of section 4, this message can be converted to a routine value:

(*) DO com1, THEN com2, THEN ... THEN comn.

If the variable "straightline" has been given this value, then the effect of the KL command

destination := source >—— straightline() ——> sink

is (*) except

- default input in com1,com2,...,comn is taken from "source" instead of the keyboard;
- default output in com1,com2,...,comn is sent to "sink" instead of the screen;
- default results in com1,com2,...,comn are given to "destination" instead of the user.

A natural generalisation of this is to allow variables in com1,com2,...,comn to be parameters. If the variable "straightline" has been given the routine value for the message

parameters: f1,f2,...,fk •—com1—>•—com2—>•...•—comn—>•

then the effect of the KL command

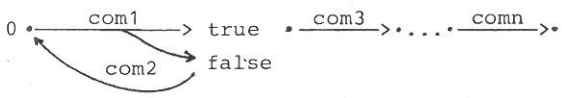
straightline(a1,a2,...,ak)

is (*) except a1,a2,...,ak are substituted for f1,f2,...,fk.

The introduction part of the value of "straightline" tells KL that messages must name precisely k variables that can be substituted for f1,f2,...,fk; the code part is just (*). Notice that the sequencing of (*) is different from the concurrency given by typing "com1" then "com2" ... then "comn"; notice also that the sequencing of (*) is different form any possible pipelining of com1,com2,...,comn.

As users of programmable pocket computers well know, "straightline" composition of routines is rather limited and "flow diagram" composition is to be preferred. The KL way of combining routines uses labelled graphs like

(+)



In this paper we want to make the semantics of KL as simple as possible, so we restrict the acceptable labelled graphs by: a single start node marked 0; if an edge has more than one sink, then its sinks have different KL values as labels. For a graph satisfying this restriction the corresponding routine value is deterministic; for the graph (+) the corresponding routine value is

(**) Step1: DO com1; GOTO CASE result OF true: step3□false; step2;
Step2: DO com2; GOTO Step1;
Step3: DO com3; GOTO Step4;

If the variable "graph" has been given this value, then the effect of the KL command

destination := source >—— graph() ——> sink

is (**) except default inputs, outputs and results in com1,com2, ..., comn are replaced by "source", "sink" and "destination" respectively. As before we allow variables in com1,com2,...,comn

to be parameters for which one can substitute when (**) is called. When a routine is called in KL, its parameters are new copies of system variables that die when the call ends, so RECURSION is possible. The PROLOG semantics in the next section will give the expected meaning to calls like: graph(graph).

Although our "recursive flowchart" way of defining new routines from old is extremely powerful, there is a case for other routine combinators. In a later paper the KL routine combinator will be extended by

- allowing non-deterministic graphs and accepting ANY successful path;
- allowing non-deterministic graphs and following ALL successful paths;
- allowing simultaneous recursion so a set of routines is given by a set of graphs;
- following the generalisation in [M] from graphs to to diagrams so routines can rendezvous and pass messages to one another.

#6 An implementation

In this section we give a precise definition of the meaning of every KL command by describing a PROLOG implementation. This implementation of KL would be very inefficient in practice, because we make various assumptions that make the meaning of KL commands as clear as possible. The most important of our simplifying assumptions is (see fig. 6.1):

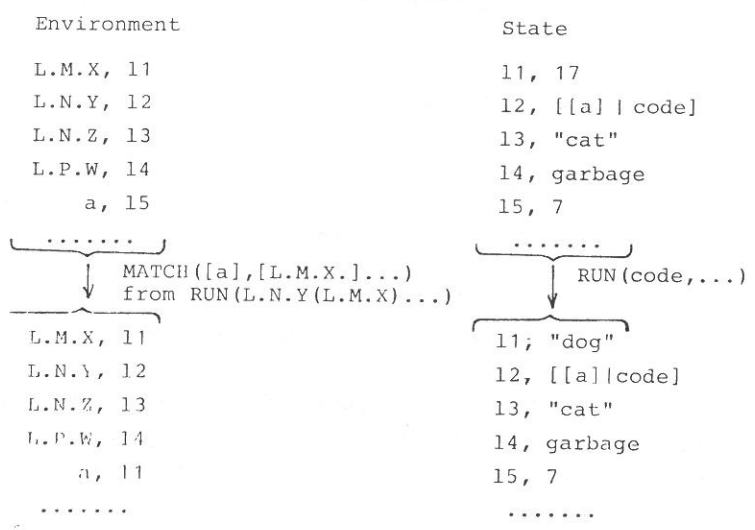


Figure 6.1 Changing environments and stores

- there is an infinite list "st" of location-value pairs;
- there is an infinite list "env" of variable-location pairs;
- there are functions GET and PUT for handling these lists.

The list "st" gives the current KL state, the list "env" gives the current KL environment (context, symbol table), and the effect of a KL command is to change the state. The meaning of the simplest KL command is given by the PROLOG rule:

```
(S1) RUN(left:=right,env,st,st')
      :- GET(env,left,l)GET(env,right,r)GET(st,r,v)PUT(st,l,v,st').
```

This rule defines the state st', which is given by executing the command (R1) in the state st with the environment env. We do not have to worry about creation of library variables, because we have assumed GET and PUT are total functions - all possible variables exist but their values may be garbage. Thus the meaning of the KL command for destroying variables is given by:

```
(S3) RUN(variable:=,env,st,st'):-GET(env,variable,l)
                                 PUT(st,l,garbage,st').
```

Let us postpone the general PROLOG rule for calling a routine, and give a particular case instead:

```
(S5) RUN(variable:=message,env,st,st')
      :- EVALUATE(message,env,st,v)GET(env,variable,l)PUT(st,l,v,st').
     EVALUATE(message,env,st,result_value)
      :- SUBSTITUTE(message,env,st,message_value)
         VAL(message_value,result_value).
```

The PROLOG implementation of routine calls in KL places the result of the call in a special state location, so the meaning of the command (R6) is given by

```
(S6) RUN(variable:=call,env,st,st')
      :- RUN(call,env,st,st")GET(st",result,v)GET(env,variable,l)
                                              PUT(st",l,v,st').
```

The implementation of routine calls has analogous conventions about special state locations, "input" and "output", so the meaning of the commands (R7-R12) is given by

```
(S7) RUN(variable  >—— call,env,st,st')
      :- GET(env.variable,l)GET(st,l,v)PUT(st,input,v,st")
         RUN(call,env,st",st').

(S8) RUN(destination:=source >—— call,env,st,st')
      :- RUN(source >—— call,env,st,st")GET(st",result,v)
         GET(destination,env,l)PUT(st",l,v,st').

(S9) RUN(call ——> variable,env,st,st')
      :- RUN(call,env,st,st")GET(st",output,v)GET(env,variable,l)
         PUT(st",l,v,st').

(S10)RUN(destination:=call ——> sink,env,st,st')
      :- RUN(call ——> sink,env,st,st")GET(st",result,v)
         GET(destination,env,l)PUT(st",l,v,st').

(S11)RUN(source >—— call ——> sink,env,st,st')
      :- RUN(source >—— call,env,st,st")GET(st",output,v)
         GET(env,sink,l)PUT(st",l,v,st').

(S12)RUN(destination:=source >—— call ——> sink,env,st,st')
      :- RUN(source >—— call ——> sink,env,st,st")
         GET(st",result,v)GET(destination,env,l)PUT(st",l,v,st').
```

The meaning of the KL pipeline commands is given by

```
(S13)RUN(variable(message) >—— call,env,st,st')
      :- RUN(variable(message),env,st,st")PIPE(st",st")
         RUN(call,env,st"',st).

(S14)RUN(call ——> variable(message),env,st,st')
      :-RUN(call,env,st,st")GET(st",result,v)PIPE(st",st"')
        RUN(variable(message),env,st"',st"")PUT(st"",result,v,st').
```

where `PIPE(st,st'):- GET(st,output,v)PUT(st,input,v,st').`

The PROLOG implementation of routine calls uses the predicate

```
(S15) APPLY([introduction|code],m,env,st,st')
       :- MATCH(introduction,m,env,st,env')RUN(code,env',st,st')
```

where MATCH creates the appropriate environment for the execution of the routine code. When the introduction indicates "call by reference used for parameters", the formal parameters are aliases for the actual parameters in this new environment (see fig. 6.1). This is how we implement "parameters are new system variables, created when the call is made and destroyed when the call is over". After this preamble we can give the meaning of a KL routine call by

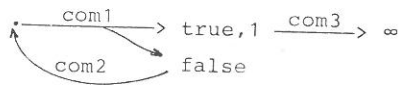
```
(S16)RUN(variable(message),env,st,st')
      :- GET(env,variable,l)GET(st,l,v)APPLY(v,message,env,st,st').
```

We can complete the description of the PROLOG implementation of KL by giving the meaning of the four commands for communication with the user:

```
(S17)RUN(variable ——>,env,st,st')
      :- GET(env,variable,l)GET(st,l,v)COERCE(v,w)PUT(st,OUTPUT,w,st').

(S18)RUN(source ——> sink,env,st,st')
      :- GET(env,source,l)GET(st,l,v)COERCE(v,w)
         GET(env,sink,l')PUT(st,l',w,st').

(S19)RUN(>—— variable,env,st,st')
      :- GET(env,variable,l)GET(st,INPUT,v)CONVERT(v,w)PUT(st,l,w,st').

(S20)RUN(source >—— destination,env,st,st')
      :- GET(env,source,l)GET(st,l,v)CONVERT(v,w)
         GET(env,dest,l')PUT(st,l',w,st').
```

Not only does our description of the PROLOG implementation use five undefined predicates - SUBSTITUTE, VAL, MATCH, COERCE and CONVERT - but it also reveals nothing about the possible codes in a routine value.

It is not unreasonable to expect a precise definition of the KL way of building a new routine from old routines (see #5). Suppose the new routine is given by the labelled graph



The paths in this graph are captured by the predicate R defined by:

```
R(0,1,env,st,st'):-RUN(com1,env,st,st')GET(st',result,true).
R(0,2,env,st,st'):-RUN(com1,env,st,st')GET(st',result,false).
R(0,∞,env,st,st'):-RUN(com1,env,st,st')
                   NOT GET(st',result,true)NOT GET(st',result,false).
R(2,0,env,st,st'):-RUN(com2,env,st,st').
R(1,∞,env,st,st'):-RUN(com3,env,st,st').
R(i,i',env,st,st'):-R(i,i",env,st,st")R(i",i',env,st",st').
```

From this example it should be clear how every graph G gives a predicate R which can be used to give the meaning of a routine which has G as its code

```
(*)  RUN(code,env,st,st'):-R(0,∞,env,st,st').
```

Parameters are also allowed in the KL way of building new routines from old, so we should also say more about the parameter passing predicate MATCH. If we assume actual and formal parameters are PROLOG lists, we can define MATCH by

```
MATCH([ ],[ ],env,st,env).
MATCH([fp|T],[ap|T'],env,st,env')
  :- GET(env,ap,l)PUT(env,fp,l,env")MATCH(T,T',env",st,env').
```


This definition is appropriate for a KL implementation that only supports programming languages in which parameters are called by reference. If the implementation supports languages with other parameter passing mechanisms, the definition of MATCH must be modified. Note that definition (*) - the meaning of "executing code" - must also be modified when a KL implementation supports other programming languages. More will be said of these modifications in a later paper.

#7 Non Determinism

Any kernel language should allow non-determinacy because routines with several possible results can be defined in PROLOG and many other programming languages. So far the only non-determinacy allowed in KL is that routines can be defined by graphs with more than one edge from a node. When such a routine is called, KL does not guarantee any particular result. Experience with Prolog implementations shows that it is unrealistic to insist on any of the desirable properties:

(CONVERGENCE) if the routine can terminate, then it will;

(REPRODUCIBILITY) repeated calls of the routine give the same result;

(PROGRESSIVE) repeated calls of the routine give successive results;

(FAIRNESS) all possible results are given, if the routine is called sufficiently often;

(EQUIPROBABILITY) all possible results are given equally often, if the routine is called infinitely often.

However it is realistic to suppose that all choices in a non-deterministic routine can be determined by the successive values in a "control" file in the same way that choices in a deterministic routine can depend on successive values in an "input" file. For the KL commands we have given so far, the user has to supply control values from the terminal to any non-deterministic routine calls. As this is frequently inconvenient, we introduce the KL commands:


```
(R21)                 variable >>----                    call
(R22)                 variable >>---- variable >---      call
(R23)  variable:=     variable >>----                    call
(R24)  variable:=     variable >>---- variable >---      call
(R25)                 variable >>----                    call ----> variable
(R26)                 variable >>---- variable >----     call ----> variable
(R27)  variable:=     variable >>----                    call ----> variable
(R28)  variable:=     variable >>---- variable >----     call ----> variable
```

These commands can be used to give computations of non-deterministic routines the desirable properties of convergence, reproducibility, progressiveness, fairness & equiprobability.

Example

Suppose R is a non-deterministic routine and f is a system variable. Consider the routine RR given by the graph

```
Set(f) -> f ----------> true   f >- Reset -> f ----------->.   f >>- R( ) ----------> false
```

where Set and Reset output a file of control values. A call of the routine RR causes a stream of R-results to be sent to the user. The particular set of R-results received by the user depends on the routines Set and Reset; since choice patterns in non-deterministic routines vary greatly, every KL implementation should have a variety of Set and Reset routines. In the special case when Set and Reset generate "random" numbers between 1 and n, and all computations of our non-deterministic routine R use a single such number, the routine RR will have the desirable properties of fairness and equiprobability.

We have not yet explained how :

- reading of input and control values
- writing of output values and results

can be expressed in the graph definition of a KL routine. Now we allow boxed edges in graphs for communication; and fix the KL meaning of some boxed edges:

quence of the fact that files are bounded and contain only finitely many values. Values are read from a KL file by moving a pointer and the contents of the file are not destroyed, whereas values received from a user or a pipeline are evanescent.

#8 Communication

Our Kernel language must have a process communication mechanism if it is to support Ada and other popular languages with parallelism. The KL mechanism is similar to that in CSP [H] and CCS [Mi]. The basic idea - synchronization of matched boxed edges - is seen at its simplest in a KL pipeline.
The command A( )—>B( )—>C( ) starts three processes that can run concurrently; the OUTPUT edges in the A-process are synchronized with the INPUT edges in the B-process; the OUTPUT edges in the B-process are synchronized with the INPUT edges in the C-process. This explanation also applies to the commands:

A( ) >— B( ) —> C( ) , A( ) >— B( ) >— C( ).

The difference between these 3 pipeline commands is that they give different driver processes. In KL several processes may be created simultaneously, but there is always one of the new processes that is the driver. The new processes may be run on separate processors, but they all die when the driver dies.

As we need a more general way of creating new processes than that given by pipelines, we introduce the net definition of routines with the examples in figure 8.1.

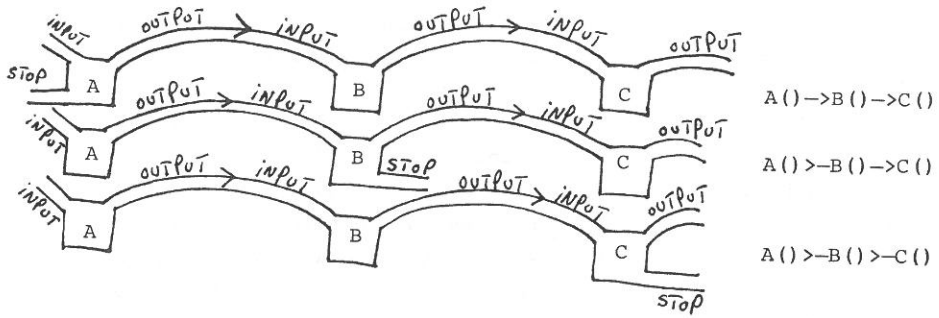
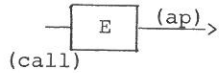


Figure 8.1 Routines defined by a net

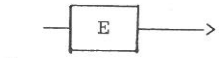
The net definition of a routine consists of directed links between process boxes. Every box in the net is labelled by a system variable, and the boxes may have several labelled ports. Each link in the net goes from a port in one process box to a port in another process box. One of the boxes in the net is distinguished as the driver - the box with the net port STOP in figure 8.1; (ports, which are in no link, are called net ports).

What is the effect of the KL command R(ap) when the routine R is given by a net definition? The command has no effect if some variable, labelling a process box in the net, does not contain a routine value. If there is a routine for each box, a process for executing that routine is created. Only one of these routines receives parameters, the driver routine is passed the message ap that was given to the routine R. From time to time the created processes will meet a boxed edge, which must be synchronised with another boxed edge, before the process can continue. The label in the box gives the port of the process to be used for synchronisation, and the net link gives the other process to be synchronised with.

Clearly processes should be allowed mort ports than INPUT, OUTPUT, START and STOP. In our Kernel language KL we allow any system variable to be the label of a boxed edge and we adopt the conventions:



sends the message ap over the link away from port E.



receives a message ap over the link towards port E and interprets it as the actual parameters for the routine value of variable E.

With these conventions KL graph definitions can easily capture ADA entry calls and rendezvous.

Example (Producers and Consumers).

Consider a buffer for values received from many producers and sent to many consumers.

Figure 8.2 gives the appropriate KL net definition. Note that KL allows the producer and consumer ports to be named P and C, where ADA would insist on Buffer. Receive and Buffer. Send respectively

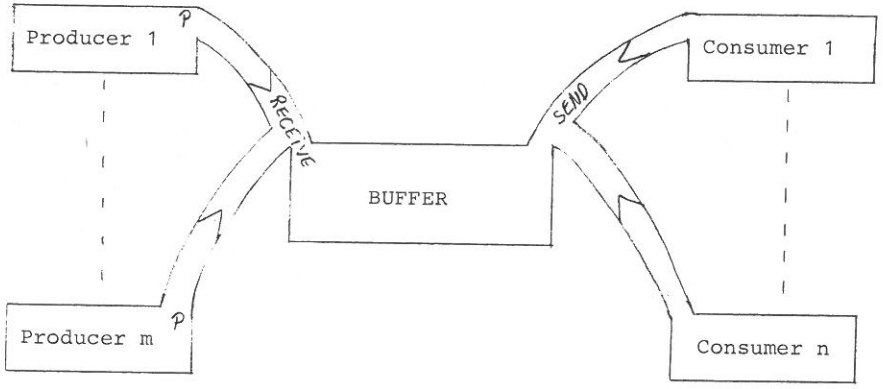
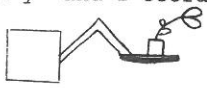


Figure 8.2 Producer and Consumer Net

Example (Robot Arm Controller)

Suppose we have a robot arm with:

- a platform on which objects can be placed and later removed.
- separate motors for adjusting the x-y- and z-coordinates of the platform position.



- an operator who gives a sequence of platform positions

(x1,y1,z1),(x2,y2,z2), --- , (0,0,0)

that ends with the home position (0,0,0).

A Petri net specification of a controller for this robot arm, and implementing programs in the languages - Pascal Plus, Occam, and Edison - are given in [KS]. The corresponding KL net definition is :

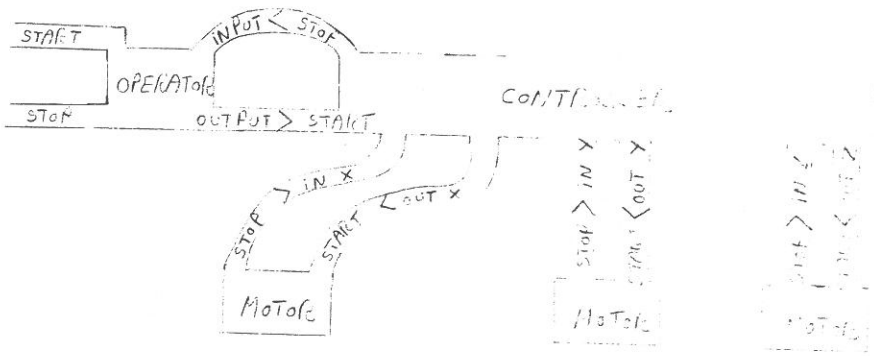
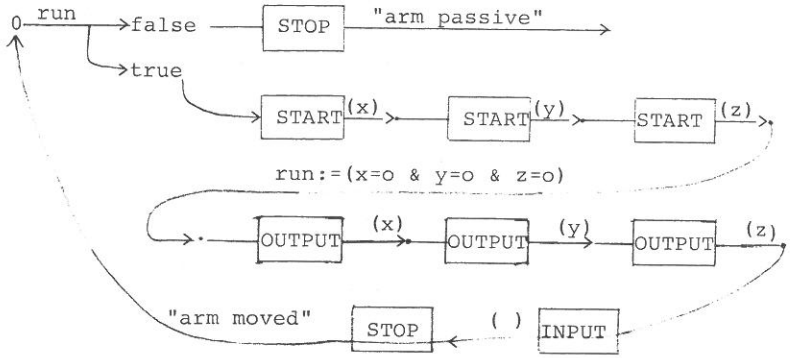
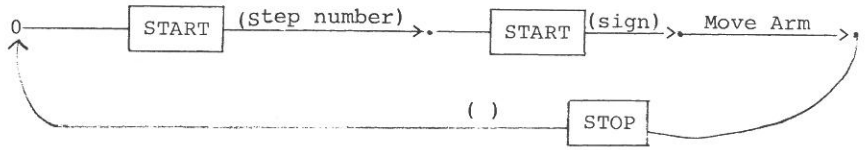


Figure 8.3 Net for Robot Arm Controller

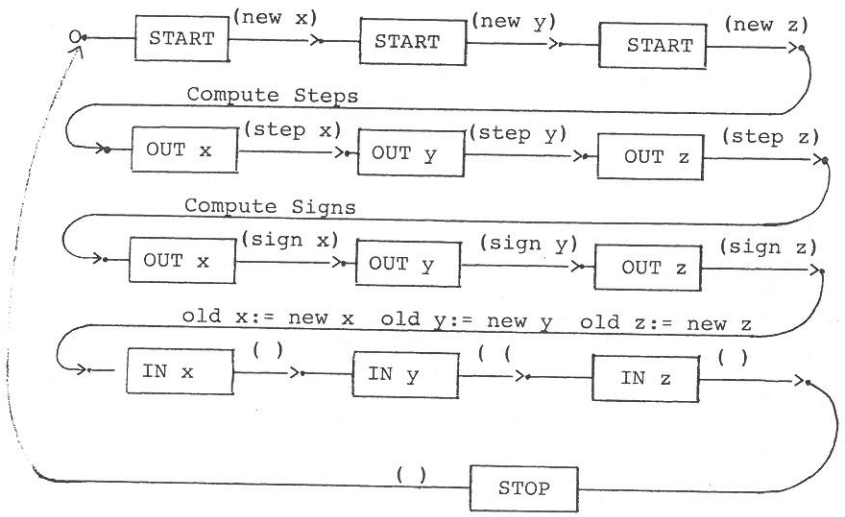
The graph definition for the operator routine is:



The graph definition for the motor routines is even simpler:



The graph definition for the controller routine is more complex:



Later we will explain how the local variables in these routines are initialised.

#9 Failure

Any kernel language should allow routines to fail in a well defined way because failing routines are an integral part of SNOBOL and other pattern matching languages. If a KL routine R is defined by a graph, then a call of R can fail in 6 ways:

(Divergence) endless looping in the graph.

(Starvation) waiting on a boxed edge for a communication that never comes;

(Death) reaching a sink vertex in the graph without producing a result;

(Jamming) following an edge gives a result that is different from all the labels on the edge tips;

(Internal) following an edge that calls a routine that fails.

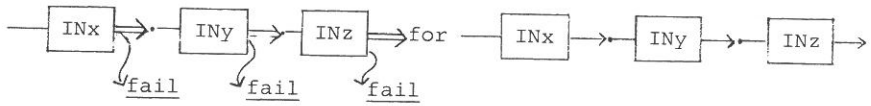
(Incoherent) following an edge that calls an undefined routine.

If a KL routine R is defined by a net, then a call of R can only fail because the driver of R fails.
When KL detects a failure, it gives the reserved value fail as the result of the routine call and it writes fail at the STOP port of the process. As the following example shows, this allows graceful recovery from failure.

Example

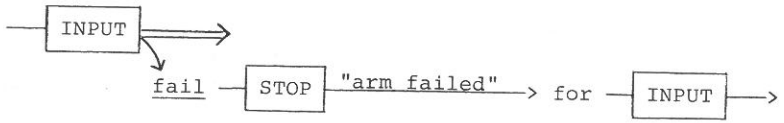
Reconsider our net definition of a robot arm controller. It contained graph definitions of five routines and all of them can fail. If any of the three motor routines fail, the value fail is written at the STOP port of the routine. Since these STOP ports are linked to the In ports of the controller routine, a handler for motor failure can be part of the graph definition of the controller routine.

If we substitute:



we get the simplest possible handler - motor failures are lumped together with controller failures.
If the controller routine fails, the value fail is written at its STOP port. Because this port is linked to the INPUT port of the operator routine, a handler for motor or controller failure can be part of the graph definition of the operator routine.

If we substitute:



we get a natural handler - user is told of the failure, then all net processes die.
The operator routine terminates properly when this handler is used, but it may fail for other reasons; if it does, fail will be the result of activating the 5 processes.

#10 Interleaving and Backtracking, Events and Activities

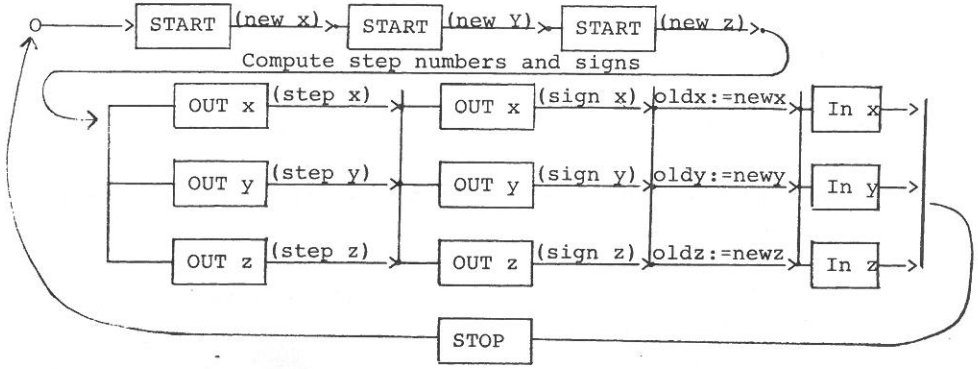
KL ought to support the common form of non-determinism, known as interleaving: obey each of the commands c_1, c_2, ...c_n but these commands can be obeyed in any order. Interleaving arises in many programming situations:

- programming languages with co begin and co end constructs
- systems with fork and join combinators
- specification languages with and and or combinators
- languages with path or flow exprissions

Should KL support interleaving by allowing commands like c_1, c_2, ... c_n ? Because such commands mesh badly with the more useful pipeline commands, the increased complexity of KL is too high a price to pay for the convenience of interleaving. Should KL support interleaving by allowing net definitions without a driver and interpreting them by: the net terminates when all of its components terminate"? This seems too high a price also. The natural way for KL to support interleaving is to allow more flexible graph definitions of routines. The argument for replacing graphs by diagrams is given in [M]; here a less radical extension will be represented: graphs can have "line"-vertices for forks and joins, not just point vertices. The way line-vertices are interpreted should be clear from the

Example

The graph definition for the robot controller routine could have been



Now that the KL ways of defining a routine have been defined completely, we can explain what happens when a routine R is called. This explanation must be done with care if the true parallelism of net definitions is not to degenerate to interleaving.
When a routine R is called, many events can happen:

- values may be read from the INPUT port
- values may be read from the START port
- values may be written to the OUTPUT port
- values may be written to the STOP port
- value passing may occur through other ports
- shared variables may be assigned new values

The events at each port are linearly ordered (in time), but the events for a particular all of the routine R as a whole are only partially ordered. If the particular call of R terminates, there is a last event in this partial order; if KL recognises that the call of R fails, it forces a last event - the value fail is written to the STOP port. A particular call of the routine R gives an activity, a partially ordered set of events. Repeating the call may give a different activity, and the meaning of the call is the set of possible activities. Non determinism and interleaving are reflected by "meanings are sets of activities"; true parallelism is captured by "events need not be linearly ordered in activities." However there is still a "shared variable" problem - we have not yet explained the effect of "shared variable assignment" events when they are incompatible with other events in an activity. ADA tasks are allowed to assign to non-local variables, but this is problematic when such tasks are run in parallel.

What happens when a pipeline A( ) >- B( ) -> C( ) is called? There are three kinds of events in this pipeline

- non-OUTPUT events of component A
- non-INPUT events of component C
- events of component B other than INPUT and OUTPUT events

The activities of the pipeline are given by

- an activity α of component A;
- an activity of component B such that OUTPUT events of α are identified with INPUT events of β;
- an activity β of component C such that OUTPUT events of β are identified with INPUT events of β.

If some call of the pipeline terminates, then the corresponding activity has a last event, which is a STOP event in the driver activity.

Pipelines give a useful form of backtracking; when a component needs an input value, it forces the proceeding routine to backtrack until it produces a new output value. Many programming languages seem to allow choice commands with the possibility of backtracking to the last point of choice if the computation fails. In KL such a choice command can be captured by a pipeline component which outputs a stream of "possible choice values". However there is still a "backtracking problem" - we have not yet explained how assignments to variables are undone when an unfortunate choice gives a computation that fails.

Example

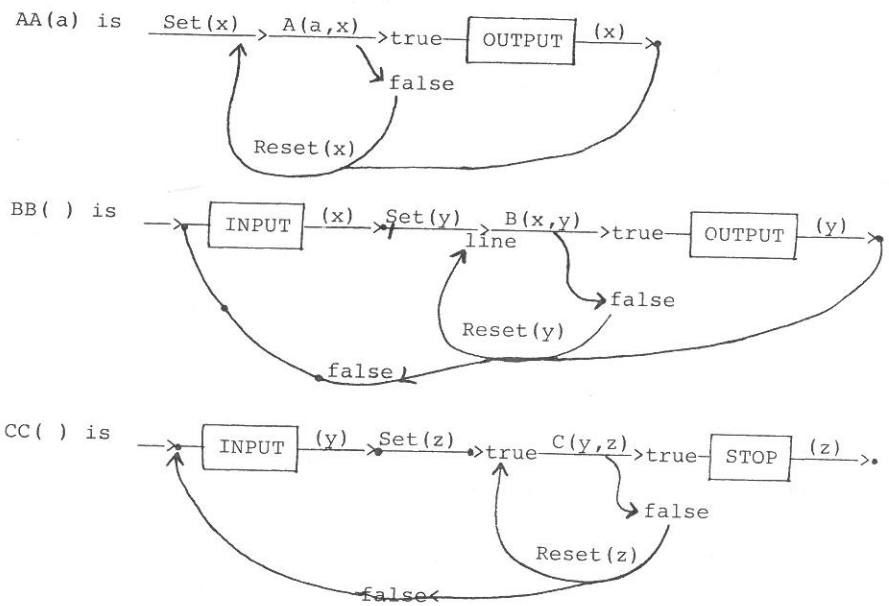
Suppose A,B,C are binary predicates and we want a program which computes z such that : $\exists x.\exists y[A(a,x)\&B(x,y)\&C(y,z)]$.
In a backtracking language one might write

```
choose x   such that A(a,x);
choose y   such that B(x,y);
choose z   such that C(y,z);   write (z);
```

In KL this is captured by the pipeline

AA(a) >——— BB( ) >——— CC( )

where the component routines have the graph definitions



The events in a call of this pipeline are

- on the link from AA to BB, writing values x_1, x_2 ..... such that A(a,xi) for all i;
- on the link from BB to CC, writing values $y_{11}, y_{12} \cdots y_{21} \cdots$ such that B(xi,yij) for all i,j;
- at the STOP port of CC, writing either fail or a value zi,j such that C(yij,zij).

□

#11 Local Variables

A realistic kernel language must allow local variables in the definition of routines. In KL a routine has a local variables for each parameter and each port. Since each routine has the standard ports - INPUT,OUTPUT,START,STOP - it also has the local varibbles - input,output,result,oracle (see fig. 7.1). For each parameter of a routine there is a local variable L and

- KL assigns the value of the actual parameter to L, when the routine is called;
- if the routine terminates without failing, then KL assigns the value of L to the actual parameter.

The KL solution to the "shared variable" problem is: shared variables are parameters of the driver component in a net; the KL solution to the backtracking problem is: system variables are parameters to backtracking components in a net.

Example

Consider a pipeline A( )>—B( )—>C( ) where the components share a variable, SV, and each component may backtrack. Suppose components A,B,C - make assignments to variables - av,bv,cv - and the components may wish to undo these assignments. The shared variable problem is solved by taking SV as a parameter of the driver component B; the backtracking problem is solved by taking - av,bv,cv - as parameters of - A,B,C - respectively.
□

Because local variables are useful for many purposes, KL allows frames of local declarations in graph definitions of routines. Each local declaration in a frame gives the name of a local variable and perhaps an initial value. The initial value of a local variable may be a message, a file or a routine. Before we delve into the mysteries of nesting, let us look at an

Example

Consider the Robot Arm Controller in section 8. The graph definition for the operator routine should have had the frame

x,y,z; run:=true;

where the only initialised local variable is "run". The graph definition for the motor routines should have had the frame

stepnumber,sign; Move Arm >— Δ;

where the local variable Move Arm is initialised to the routine with the graph definition Δ.
The graph definition for the controller routine should have had the frame

newx,newy,newz; oldx,oldy,oldz:=0;

Compute Steps >— Δ; Compute Signs>— Δ^1;

Inx,Iny,Inz >— INPUT; Outx,Outy,Outz >— OUTPUT;

where the local port variables are initialised appropriately.
□

Remember our discussion of library variables in section 1. Suppose R is a library variable with a routine as value and KL is given the call R(ap). While this call is executing, the local variables of R are new leaves of the library tree attached to the "new module" R. Figure 11.1 shows the changes in the library for a recursive call of a routine with three local variables. We would have a similar

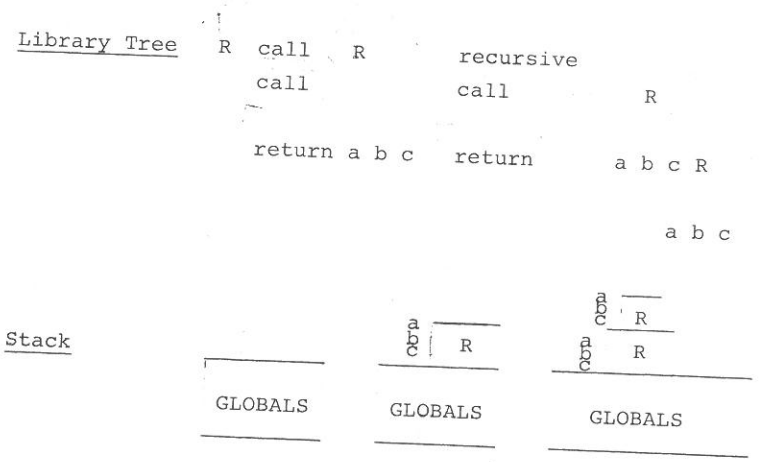


Fig. 11.1 Library changes when routines are called

figure if there was a call of a nested routine with three parameters during a call of R.
Suppose the routine R is given by a net definition. The new library tree has the local variables of components as leaves attached to the new component node C, and each component node is attached to the node for R. Figure 11.2 shows the change in the library tree when the routine for the Robot Arm Controller is called.
Note the use of a cactus stack to capture the independence of components in a net; the components may well be running on separate processors, each of which has the corresponding branch of the cactus stack in its local memory.

declarations, not just declarations of local variables which are initialised to a routine with a frame.

Example

The graph definition for the controller routine could have had the frame:

```
Compute Steps >— Δ;  Compute Signs >— Δ¹;
frame x is  newx;oldx:=0;Inx>—INPUT;Outx>—OUTPUT,
frame y is  newy;oldy:=0;Iny>—INPUT;Outy>—OUTPUT;
frame z is  newz;oldz:=0;Inz>—INPUT;Outz>—OUTPUT;
```

and the diagram representation in figure 11.4. Note how frame names become library module names when the controller routine is called.

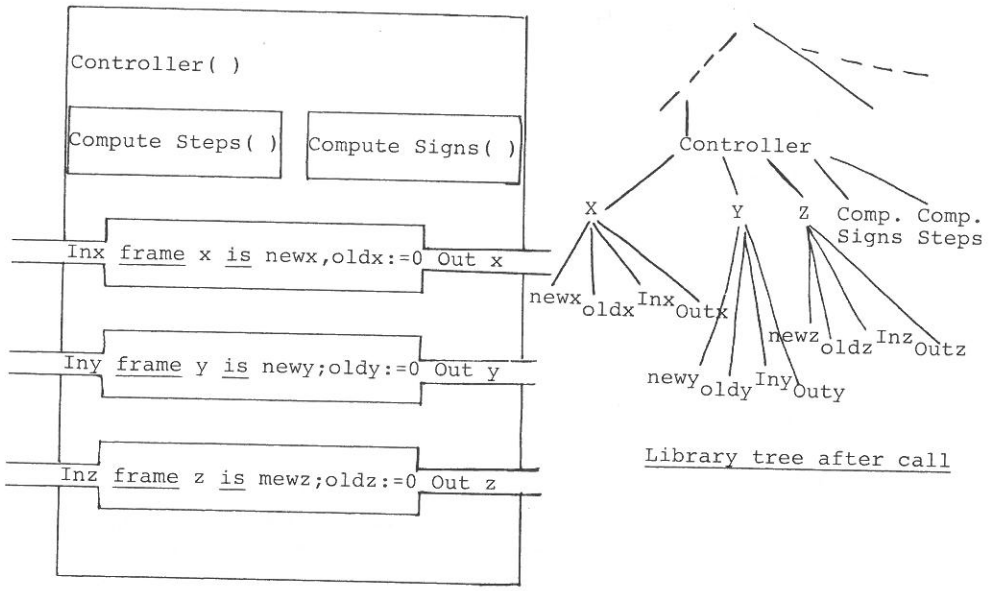


Fig. 11.4 Diagram for direct frame nesting

#12 More on implementation

In this section we extend the implementation in #6 so that the KL constructions in the later sections have a precise meaning. It is easy to give a PROLOG implementation of the new KL commands

```
(S21)  RUN(control>>—call,env,st,st')
          :-GET(env,control,1)GET(st,1,v)
            PUT(st,oracle,v,st")RUN(call,env,st",st').

(S22)  RUN(control>>—source>—call,env,st,st')
          :-GET(env,control,1)GET(st,1,v)
            PUT(st,oracle,v,st")RUN(source>—call,env,st",st').

(S23)  RUN(destination:=control>>—call,env,st,st')
          :-GET(env,variable,1)GET(st,1,v)
            PUT(st,oracle,v,st")RUN(destination:=call,env,st",st').

(S24)  RUN(destination:=control>>—source>—call,env,st,st')
          :-GET(env,control,1)GET(st,1,v)
            PUT(st,oracle,v,st")RUN(destination:=source>—call,env,
                                                        st",st').

(S25)  RUN(control>>—call—>sink,env,st,st')
          :-GET(env,control,1)GET(st,1,v)
            PUT(st,oracle,v,st")RUN(call—>sink,env,st",st').

(S26)  RUN(control>>—source>—call—>sink,env,st,st')
          :-GET(env,control,1)GET(st,1,v)
            PUT(st,oracle,v,st")RUN(source>—call—>sink,env,st",st').

(S27)  RUN(destination:=control>>—call—>sink,env,st,st')
          :-GET(env,control,1)GET(st,1,v)
            PUT(st,oracle,v,st")RUN(destination:=call—>sink,env,
                                                        st",st').

(S28)  RUN(destination:=control>>—source>—call—>sink,env,st,st')
          :-GET(env,control,1)GET(st,1,v)PUT(st,oracle,v,st")
            RUN(destination:=source>—call—>sink,env,st",st').
```

In section 6 we gave a precise definition of the behaviour of a routine given by a graph, whose edges were labelled by KL commands. In the same way that an edge, labelled C, gave a PROLOG statement like

```
R(i,j,env,st,st'):-RUN(c,env,st,st').....
```

a boxed edge in a graph gives a PROLOG statement like

```
R(i,j,env,st,st'):-RUN(—[ P ]—(m)—>,env,st,st').....
```

The implementation of boxes with labels - INPUT,START,OUTPUT,STOP - is given by

```
RUN(—[ INPUT ]—(a)—>,env,st,st'):-
   GET(st,input,[h,t])PUT(st,input,t,st")PUT(st",result,h,st"')
   GET(env,a,l)PUT(st"',l,h,st').

RUN(—[ INPUT ]—>,env,st,st'):-
   GET(st,input,[h|t])PUT(st,input,t,st")PUT(st",result,h,st').

RUN(—[ START ]—(b)—>,env,st,st'):-
   GET(st,control,[h|t])PUT(st,control,t,st")PUT(st",result,
                                                    h,st"').
   GET(env,b,e)PUT(st"',l,h,st').

RUN(—[ START ]—ll—>,env,st,st'):-
   GET(st,control,[h,t])PUT(st,control,t,st")PUT(st",result,
                                                    h,st').

RUN(—[ OUTPUT ]—(c)—>,env,st,st'):-
   GET(env,c,l)GET(st,l,h)PUT(st,result,h,st")
   GET(st",output,t)PUT(st",output,[t|h],st').

RUN(—[ OUTPUT ]—()—>,env,st,st'):-
   GET(st,result,h,st")
   GET(st",output,t)PUT(st",output,[t|h],st').

RUN(—[ STOP ]—(d)—>,env,st,st'):-
   GET(env,d,l)GET(st,l,v)PUT(st,result,v,st').

RUN(—[ STOP ]—()—>,env,st,st).
```

The implementation of local variables in routines, defined by graphs, is given by a small modification of (S 4')


```
(S 4")  APPLY( [ [params|locals]|code],m,env,st,st')
     :-MATCH(params,m,env,st,env')
       INITIALISE(locals,env',st,env",st")RUN(code,env",st",st').
```

The predicate INITIALISE builds the new environment by adding a location for each local variable, then it builds a new state by putting initial values in these locations.

Now let us look at net definitions of routines. Without loss of generality we can assume that each component routine in a net N has a graph definition. The graph definitions for n component routines can be combined into a graph G_n for the whole net as follows

- the vertices of G_N are n-tuples of component vertices;
- the initial vertex of G_N is the n-tuple of component initial vertices;
- the terminal vertices of G_N are the n-tuples of component vertices where the driver vertex is terminal;
- there is an G_N edge

  $$\langle i_1 \ldots i_n \rangle \xrightarrow{\alpha_1 x \ldots . x \alpha_n} \langle j_1 \ldots . . jn \rangle$$

  if and only if $\alpha_1 x \ldots x \alpha_n$ is a permissible synchronisation and for each component we have either the edge

  $$i_k \xrightarrow{\alpha_k} j_k \quad \text{or}$$

  $\alpha_k = 1$ and $i_k = j_k$.

The implementation of the net routine is given by the implementation of the graph G_N:

- if $i_k \xrightarrow{\alpha k} j_k$ is an unboxed edge,

then |x..x|xα_kx|x ..x| is a permissible synchronisation

and R(<$i_1 \ldots i_n$>,<$j_1 \ldots j_n$>,env,st,st'):-RUN(αk,env,st,st')...

- if $i_k \xrightarrow{\alpha k} j_k$ is a boxed edge and the label in the box i a net port, then |x..x|xα_kx|x x| is a permissable sychronisation

and R(<i_1..i_n>,<j_1..j_n>,env,st,st'):-RUN(α_k,env,st,st')...

- if there is a link from port P_k in the k-th component to a port P_l in the l-th process,

- if $i_k \xrightarrow{\alpha k} j_k$ is $i_k \longrightarrow [P_k] \longrightarrow j_k$

and $i_l \xrightarrow{\alpha l} j_l$ is $i_l \longrightarrow [P_l] \longrightarrow j_l$

then $|x..x|xak|..x|x\alpha_l x|x..x|$ is a permissible synchronisation

and $R(<i_1..i_n>,<j_1..j_{(n}>,env,st,st'):-RUN(P_l(ap),env,st,st')...$

This definition must be modified slightly to cover the cases when standard ports - INPUT,OUTPUT,STOP and START - are used in net links.
If P_l is a standard port, then

RUN(P_l(ap),env,st,st') in (*) is replaced by

- if $i_l \longrightarrow [Pi] \xrightarrow{(a)} j_l$, then RUN(a:=ap,env,st,st')

- if $i_l \longrightarrow [Pk] \longrightarrow j_l$, then RUN (result:=ap,env,st,st').

If P_k is a standard port, then

RUN(P_l(ap),env,st,st') in (*) is replaced by

- if $i_k \longrightarrow [Pk] \xrightarrow{(a)} j_k$

then RUN(ap:=a,env,st,st")RUN(P_l(ap),env,st",st')

- if $i_k \longrightarrow [Pk] \longrightarrow j_k$

then RUN(ap:=result,env,st,st")RUN(P_l(ap),env,st",st')

If both P_k and P_l are standard ports, then the substitution for P_k is made before that for P_l.

Example

In a pipeline output ports are linked to input ports and the normal situation is

$i_k \longrightarrow [OUTPUT] \xrightarrow{(c)} Jk$ $\qquad$ $i_l \longrightarrow [INPUT] \xrightarrow{(a)} J_l$

$(<i,i_n>,<j,..jn>,env,st,st')$

$:-RUN(ap:=c,env,st,st")RUN(a:=ap,env,st",st')...$

The PROLOG implementation can be rewritten

$R(<i_1 \ldots i_n>,<j_1 \ldots j_n>,env,st,st')$
(+) $:-RUN(a:=c,env,st,st').....$

Do we have the same implementation as that given earlier for pipeline commands (S 13-14 in section 6), with its predicate

PIPE((st,st'):-GET(st,output,v)PUT(st,input,v,st').

? Yes, because (+) shows that the events on a pipeline link are a sequence of assignments and these can be separated into "writing an output file" and "reading an input file". Notice that the difference between (S 13) and (S 14) is captured by "the terminal vertices of the pipeline graph are given by the terminal vertices of the driver graph".
□

The use of interleaving in our PROLOG implementation,instead of true parallelism, is justified by the argument: the behaviour of a routine does not depend on the speed of the underlying processor so the partial order of events in an activity can be embedded in a linear order without loss of generality.

The use of just one environment and one store in our implementation of net routines seems to conflict with our informal discussion of local variables and a cactus stack. However, there is no conflict if the PROLOG implementation gives distinct "internal" names to the component local variables and the local names in the component graphs are renamed accordingly.

References:

[HO] C.A.R. Hoare:

Communicating Sequential Processes, Comm. A.C.M. 21, (1978) 666-677.

[HU] J.M. Hullot:

A multi-formalism programming environment, Proc. IFIP 83, Paris.

[KM] B.W. Kernighan & J.R. Mashey:

The UNIX programming environment, IEEC computer 14 (1981) 12-24.

[KMMN] B.B. Kristensen. O.L. Madsen, B. Møller-Pedersen, K.Nygaard:

Syntax Directed Program Modularization, in "Interactive Computing Systems" ed. P. Degano, E. Sandewall, North Holland 1982.

[KS] J.M. Kerridge, D. Simpson:

Three solutions for a robot arm controller using PASCAL-plus, OCCAM and EDISON, Softw.pract.exp. 14 (1984) 3-16.

[M] B.H. Mayoh:

Models of programs and processes Inf. Proc. Lit. 17 (1983) 211-214.

[MI] R. Milner:

A calculus of communicating systems, LNCS v. 92, Springer 1980.