Computer Science 773

Robotics and Real-time Control

PFL

(NOTE : this text is taken, with some amendments, from my Working Note AC49 on PFL, and describes work in progress, not a completed system.)

PFL (Process Formulation Language) is still in the excruciatingly slow throes of development. A student's work on an early version[1] produced a compiler; more to the point, the exercise taught us a lot about what we were doing wrong, and here we describe a new version of the language, as yet unimplemented.

Even the unimplemented language can be useful. By writing programmes using the language, one achieves a disciplined and structured programming style which is hard to attain in a less structured environment. The programmes can later be translated by hand into some other language. This approach was found to be very successful in a trial by a class of engineering students, using the first version of PFL; programmes were written more easily, and worked correctly sooner, than similar programmes written directly in assembly languages[3].

The language is very much a procedural language with an imperative style. It is, after all, central to the idea of control that side effects should occur and that a sequence of operations should be precisely defined; alternative models such as the functional (Lisp) or assertive (Prolog) patterns are simply not appropriate. On the other hand, our system could be pushed fairly easily into the object-oriented (Smalltalk) class : we have not deliberately designed it that way, but our language has developed of its own accord features rather like those of Simula.

FEATURES OF THE LANGUAGE.

The language includes the common components of "scientific" languages : real, integer, logical, char, and string data types, and Pascal-like data and programme structures. The syntax and vocabulary are not always conventional, as we have emphasised ease and naturalness of use over ease of compiling : we have not felt constrained to produce a one-pass compiler. We shall not deal with these features here. Instead, we shall concentrate on the programme and data structures which PFL provides, and which we believe to be specially significant to control programming. The extensions provided for control purposes can conveniently be reviewed under four headings : processes, machines, timing, and interfaces.

Processes.

A process is an autonomous activity; everything that happens in a PFL programme must happen in a process. Each process is thought of as handling some identifiable task, which may be an independent task, or a task associated with a machine. The number of processes associated with a programme is determined by the PFL compiler, and is fixed : processes cannot be created or destroyed once the programme has started, though they need not all be active. A process which has not yet begun to execute, or which has finished, is said to *sleep.* A sleeping process consumes no system resources, except for one entry in the process table and enough memory to store its code. Any sleeping process may in principle be woken; a newly woken process always begins execution at the beginning. A sleeping process is quite distinct from a *suspended* process; a process may be suspended for one of a number of reasons while it is in execution, but when resumed it continues execution from the point at which it was suspended. This aspect of PFL's

design reflects its target field, which is the control of some predefined arrangement of plant or machinery. That is not to say that the configuration of the plant cannot change; but it *is* to say that any mode of change must be foreseen and catered for in the programme. We insist on this rule, because we want to ensure that every programme which runs must at least come, in its entirety, under the scrutiny of the compiler before it is executed, so that it can be checked. We do not believe that it is possible reliably to test a control system if the plant it controls is subject to unplanned changes.

Processes can communicate with each other by sending messages; processes which belong to the same machine may also share memory. Processes may suspend themselves for any specified time interval, or until some event occurs.

The language provides no explicit process declaration : a process is identified by the way it is executed, not by the static structure of the programme. Any statement may be used as the body of a process : it is executed as a process if it appears in one of the instructions **when**, **whenever**, or **run**. Consider this example of a procedure declaration :

```
vesselcontrol means
begin

… statements1 …
turnoff;        % A procedure invocation.
… statements2 …

. . .

turnon means   % A procedure declaration.
    begin
    if vessel's alliswell
    then switch on vessel's heater;
    when vessel's temp > maximum
        turnoff;
    end;

turnoff means  % A procedure declaration.
    begin
    switch off vessel's heater;

    when vessel's temp < minimum
        if not vessel's finished
        then turnon;
    end;

end of vesselcontrol;
```

This procedure illustrates a natural way to encode a thermostat controller. It could be executed as a procedure by invoking its name, just as in Algol; or it could be executed as a process with the instruction

```
run vesselcontrol
```

If run as a procedure, the procedure body is executed , then the procedure's activation record is discarded; when run as a process, the **run** instruction wakes up a process predefined by the compiler, and schedules it for execution. In either case, two auxiliary

processes are implied, one for each of the **when** instructions; these will be discussed further in what follows.

Machines.

A machine is a structure within the programme which is intended to represent a single item of plant. It typically contains declarations of local variables and structures, an **image** declaration which describes the plant's interface, and one or more procedures to handle various activities concerning the plant. A possible design would be to have a separate procedure, run as a process, to handle each input from and output to the plant, with communication by messages and variables local to the machine. The machine also contains code to handle starting up and shutting down; the final version of PFL is likely also to contain provision for describing emergency shutdown procedures, but we are not yet ready to fix the details of this part of the operation.

A **machine** declaration resembles a Simula class : it defines an abstract type, and several machines of the type may be declared. The name of a machine may appear in **start** or **stop** statements, which cause the corresponding code within the machine declaration to be executed. The variables within a machine are accessible outside the machine declaration, where they must be qualified by the machine's name, following the pattern "thismachine's internalvariable"; some examples appear in the procedure declaration above.

It is worth emphasising this significant difference between a machine and a procedure. When a procedure is not being executed, in PFL, as in Algol-like languages generally, its internal variables do not exist. A procedure describes an algorithm, which has no physical existence. The internal variables of a machine, on the other hand, do exist irrespective of the state of the programme's execution, because the machine declaration must reflect the properties of the real physical machine, which does not go away just because our programme happens not to be observing it.

Here is an example of a **machine** declaration :

```
vessel is thermostat; % Declare the object.

thermostat is         % Declare the "type".
machine

image :               % The plant, seen from the
                      % programme.
    temp         is number in;
    heater       is switch;
    red-light    is switch;
    mainswitch   is indicator;
    empty        is indicator;
    lid-open     is indicator;
    state-change is indicator pulse;
end image;

                      % Some local variables.
finished         is logical, initially false;
alliswell        is logical, initially false;
```

```
startup :           % How to turn it on.
    while lid-open
        begin
        send "The vessel's lid is open." to
                    displayconsole;
        suspend;
        end;
    while not mainswitch
        begin
        send "The heater is not switched on." to
                    displayconsole;
        suspend;
        end;
    alliswell := true;
    vesselcontrol;
end startup;

shutdown :          % How to turn it off.
    alliswell := false;
    finished := true;
    switch off heater;
    send "The heater is switched off." to
                    displayconsole;
    wait until temp < safelimit;
    send "Reaction vessel is now cool." to
                    displayconsole;
end shutdown;

whenever state-change  % What to do.
    begin
    if empty          % An emergency.
    then begin
        alliswell := false;
        shutdown;
        soundthealarm;
        end
    else
        if lid-open
        then switch on red-light
        else switch off red-light;
    end;

end machine;
```

Timing.

The importance of timing in control programming is seen in several features of PFL.

First, there are two *data types* which can be used for arithmetic with time : the **interval**, which ranges over time periods, and the **clocktime**, which represents a time of day (including the date). The language provides no constant of either of these types as such; constants are always constructed by applying appropriate operators to numeric variables. For the **interval** type, the operators are **seconds**, **minutes**, **days**, etc, which are postfixed to their operands; examples of constant **interval** values are :

```
6.67 weeks
15 days
6 minutes + 3.667 seconds
```

The "largest" such operator is **weeks**, as longer units in common usage are of varying length. All these operators accept operands of **real** or **integer** type. For the **clocktime** values, the 6-adic operator is : `: :: : :` ; an example of its use is :

```
1986:3:22::15:30:22.445
```

Only the last argument may be **real**; all others must be i**nteger**s. Any argument of the **clocktime** operator can be omitted, in which case the value of that argument at execution time will be understood – so, for example, `::::20:0:0` means "8 pm today". Because of this convention, the expression `::::::` is a valid **clocktime**, meaning the current time; for convenience (and comprehensibility !) , this can be replaced by the reserved function name **now**.

Three *statements* in the language are explicitly concerned with timing matters : **wait**, **when**, and **whenever**.

wait is used to suspend a process until a given condition is satisfied, or – as a special case – until a specified time. The **wait** statement looks like this :

```
wait until temp < maximum;
wait until ::::20:0:0;
wait for delay seconds;
```

If the condition is already satisfied, or the time is already past, then the **wait** statement has no effect; but if not the process is suspended until the condition is satisfied, and is then allowed to proceed. The first of these instructions is compiled as :

```
while temp >= maximum
    suspend;
```

The other instructions lead to the process's transfer to the system's calendar queue.

when is used to establish a process's response to a condition, should that condition arise, but the process itself does not wait. The form of the statement is

```
when condition statement;
```

In effect, "statement" is remembered as the process's response to the "condition", and execution continues. As PFL is a traditionally scoped language permitting nested blocks, there is clearly a question of how to ensure that the necessary context still exists when the "condition" is satisfied and the "statement" is executed. Activation records for contexts which still contain unsatisfied **when** statements must not be discarded. This forces a "cactus-stack" structure onto the implementation, which we shall discuss later.

The compiler must analyse the structure of a **when** statement with some care. Consider the simple thermostat controller presented above as the procedure `vesselcontrol`. When the procedure is executed, the thermostat is started by executing the procedure invocation `turnoff`. The procedure is entered, and whatever signal is needed to switch off the heater is sent to the vessel. The

appropriate "when-process" is then activated : `turnoff` has now finished, and the main process continues with "statements2". The when-process consists, in effect, of the statement

```
repeat
    if temp < maximum
        begin
        turnon;
        sleep;
        end;
    suspend;
forever;
```

together with the minimum context needed to execute that statement. It is the compiler's responsibility to identify the minimum context required; this will usually consist of those scopes statically enclosing any procedures or variables cited in the **when** statement, or used in the condition. Problems could possibly arise if such procedures were permitted to have any but value parameters; we have not yet worked through the implications of this question. Notice that, in conformity with our principle that processes cannot be created or destroyed, the when-proceses must be identified and set up by the compiler before the programme starts.

whenever, like **when**, activates a special process (the whenever-process). Unlike a when-process, though, a whenever-process, once activated, remains active until it is explicitly put to sleep, and is executed every time its condition is satisfied. It is intended as an interrupt-handling device, so the condition is usually an interrupt; other conditions are permitted for greater generality, but are interpreted as "whenever there is a change in the logical expression" – otherwise a condition's becoming true could precipitate continued execution of an associated whenever-process. An alternative would be to permit a general logical expression as the condition, but to insist that the programme should in some sense reset the condition before returning; it is not clear, though, how the compiler could check whether this requirement had been satisfied. An execution-time check is inappropriate : what action should be taken if it is found that the condition remains unsatisfied at the end of the statement ? It seemed better to define an interrupt-like interpretation of a condition, and the convention we adopt seems likely to conform to most requirements.

Interfaces.

High-level approaches to defining the connections between the control programme and the plant which it controls seem to have received rather little attention, and we have accordingly taken particular care with this part of our system. A very detailed analysis is offered in Pearl[2]; the facilities in the PFL system correspond roughly to Pearl's system-division. Our deliberations have been strongly guided by the principle that as far as practicable the control programme should be independent of the computer hardware on which it runs; it should rather be determined only by the control algorithms required and the communications characteristics of the plant.

We have therefore split the specification of the plant interface into two parts : the **image**, which describes the information and control lines between the plant and the computer in terms of the data types of the information carried; and the *connection block*. The connection block is not strictly regarded as a part of the PFL programme at all, but as information provided for the system loader. It includes a description of the electrical characteristics of the signals to and from the plant, and also defines the computer ports to

which the signals will be connected. The idea of this separation is, of course, not new : the principle appeared well over two decades ago in the post-processors used with numerical control languages for NC machine tools, and in the "environment division" of Cobol programmes.

The image is part of a **machine** declaration; it describes, at a logical level, how the programme sees the plant, and associates symbolic names with each of the input or output data items, and control and signal lines. As an example, consider the **image** declaration which appears in the example of the code for a machine which appears earlier. Once declared in the image, the identifiers which name the communications channels between the programme and the plant can be used in the PFL programme in contexts appropriate to their types : thus, `temp` may be used as a numeric quantity, `heater`, `red-light`, `mainswitch`, `empty` and `lid-open` are effectively logical quantities, and `state-change` is an interrupt. The "variables" are distinguished from ordinary variables declared within the programme by the conditions under which they may be used : `temp`, `mainswitch`, `empty`, and `lid-open` may be used as operands but cannot be assigned values, while `heater` and `red-light` may only be assigned values. The identifiers are accessible from outside the **machine** declaration, when they must be qualified by the machine's name : thus, "vessel's temp" in an earlier example.

The image provides all the information about the plant which is needed to write control programmes, but does not specify how the connections are to be made. The compiler converts references to the identifiers declared within the image into supervisor calls which take as one parameter an index which identifies the identifier in question; the compiler's output file includes a table carrying details of all the programme's **image** declarations. The linkage with actual input and output ports is made by the sytem loader – which must, of course, have information specific to the hardware system to which it pertains. In principle, a separate loader will be required for each computer type used, but in many cases differences between machines can be condensed into entries in a form of configuration table.

The loader generates linking code which forms part of the connection block; it is directed by a programme written in an auxiliary language, the *Connection Block Language* (CBL). Unlike PFL, CBL is purely descriptive; while its style is, for obvious reasons of compatibility and convenience, in many respects similar to that of PFL, it is best regarded as an assertive language.

The function of a CBL programme is to describe the disposition over the machine's ports of signals described in the image. For example, here is a possible CBL connection specification which corresponds to the **image** declaration seen above :

```
vessel's image :
    temp        is number in  port A[ 1 .. 8 ];
    heater      is switch  port A[ 9 ];
    red-light   is switch  port A[ 10 ];
    mainswitch  is indicator  port B[ 1 ];
    empty       is indicator  port B[ 2 ];
    lid-openis indicator  port B[ 3 ];
    state-changeis indicator pulse
                      port B[ 4 ];
end image;
```

The identifiers are the keys through which the PFL and CBL programmes match up their declarations. The type specification is deliberately repeated; CBL checks that the type declarations match, exploiting the redundancy to gain added security, but there is a more

important reason. The PFL and CBL programmes give information to different people : the PFL programme reflects the programmer's understanding of the control system, but the CBL programme is likely to be used as documentation by engineering staff who are in a position to exercise an independent check on the correctness of the information, and the repitition makes this check possible.

IMPLEMENTATION.

Two of our aims are : to be able to compile the complete control programme for a system, so that we reap the benefits of automatic consistency checking throughout, and for easier testing; and to make our programming language independent of any specific machine's hardware, even to the extent of distributing programmes over machines of different types. If these aims are to be reconciled, then we must find a way to "factor out" the machine dependencies. At the language level, we achieved this by moving all the machine-dependent parts of the the problem specification out of PFL into CBL; at the implementation level, we can achieve a similar result by making the target language of our compiler a standard *virtual machine* code, and providing virtual machines as programmes which interpret the generated code on each of the varieties of computer to be used. While this approach certainly entails a considerable programming effort, it need be done only once for each computer; and the effectiveness of the technique is demonstrated by the success of UCSD Pascal, running on the virtual P-machine.

NOTE : The object which we call a virtual machine *is sometimes referred to as an* abstract machine. *We prefer to use the term* virtual machine, *because we feel that "abstract" is a curious adjective to apply to an object which we expect to be very active indeed. We would prefer to reserve the term* abstract machine *for objects such as those discussed in automata theory, which really are intended as abstractions of actual machines; machines such as ours, which are at least potential candidates for implementation in hardware, seem more properly termed "virtual".*

An alternative approach, which would not incur the overhead associated with interpretive execution, would be to replace the virtual machines by a retargetable code generator, so that real machine code for each computer used could be generated. We see this as a possible line of evolution (rather than revolution : notice that it can be introduced machine by machine), but for the present at least have chosen to follow the virtual machine path as a conceptually simpler way to start. We have made one concession to efficiency : expecting that the target computers will almost always be microprocessors, we have provided rather different virtual machines to run on 8-bit and 16-bit processors. The PFL compiler therefore produces a form of assembly language rather than actual virtual machine code; this is translated into the correct sort of virtual machine code when the hardware type becomes known during the loading operations. (We hope to describe the dual virtual machine system in a separate publication.)

The virtual machine has a stack architecture, with space for local variables assigned in the stack for convenience. Several processes within a machine can be simultaneously active, and, as conventional scope rules apply, they may all require access to the machine's global variables. They may also need to share parts of their stacks in rather more intimate ways, as was suggested during the earlier description of when-processes and whenever-processes. At the moment of awakening of one of these types of process, it must share all the memory space of the parent process, because all references to variables are to the same variables, and a change to a variable made by the parent process must be visible to the child. As execution of the parent continues, though, it may leave the

procedures which supply the immediate scope of the child process. The child can still use variables defined in such procedures; but they are no longer accessible to the parent. What should happen now if the parent returns to one of the procedures which it has left in the care of the child ? Should the previous, still surviving, activation be reentered – or should a new activation be constructed ? It seems that the older activation should indeed be reentered; otherwise, changes to variables made by the child-process would never become accessible to the parent. Considerations of this sort lead us to an implementation design in which activation records are linked together in flexible ways, and only dismantled when their usage counters become zero.

It is also necessary to maske provision for processes which wait on any one of several interrupts. Perhaps the simplest example of such a process is illustrated by the code fragment :

```
when bellpush or :::::0:0
    ringthebell;
```

The when-process containing `ringthebell` must wait on two quite separate events, the bellpush signal and the turn of the hour, but, as the process can only "fire" once, its association with each signal must be made conditional on the nonoccurrence of the other. In effect, the process must be in two queues simultaneously. We have not yet resolved this problem, though several solutions suggest themselves. For example, a solution depending largely on software is possible, but requires that interrupts be suppressed while it executes; alternatively, we can provide for such requirements in our "virtualware", which would respond more quickly but extend the virtual machine with code that may rarely be useful.

The compiler produces a code file for each machine or independent process in the programme; the file includes the virtual machine "code", details of any **image** declarations, and details of requirements for message transmission to other processes, and for conventional input and output facilities. The loader accepts these intermediate files, and a CBL programme describing of the connections to the plant, and specifying which machines and processes are to be implemented on which actual computers. The loader produces a memory load for each computer, including a multiprogrammed virtual machine, incorporating essential operating system functions such as scheduling, the virtual machine code for the processes to be run in that processor, code for the connection block, and such input, output, and message-handling facilities as are required. The machines assembled can be tailored to the requirements of the processes they will run, so it is still possible to use quite modest processors, even though the maximum size of a virtual machine may be very considerable.

REFERENCES.

1 : P.A. Sergent-Shadbolt : *A new computer language for process control*, Ph.D. thesis, Auckland University (1986).

2 : *Basic PEARL*, DIN (Deutsches Institut für Normung) 66 253 Teil 1 (1981).

3 : See, for example, B.J. Cook, D.G. Jane, I.R. Sydenham, C.J. Thomsen, and C.I. Wallace : *Steering control for an automated guided vehicle* ("Miniproject report" to G.W. Blanchard, Department of Mechanical Engineering, Auckland University, 1984)

Alan Creak,
April, 1998.