Alan Creak
11 October 1993

THE PFL – CBL INTERFACE.

The interface between PFL and CBL has never been satisfactorily defined. Peter Sergent described a basic version in his thesis(1), I glossed over it in a working note(2), and Roger Kay has proposed an alternative implementation(3). Here I try to set out some issues which should be considered in a more complete analysis.

THE FUNCTION OF THE INTERFACE.

PFL is a language for writing computer programmes which control machines. It is intended to be a general language, incorporating features which are appropriate for control programming. These include both language constructs and data structures which describe the machines' interfaces in sufficient detail for the control algorithms' purposes.

These descriptions do not address the question of how the various components of the control interface are attached to the computer. This omission is deliberate. To build specific details of a computer into the control programme both obscures the nature of the control algorithm, which is the proper function of the programme, and ensures that the programme is not portable. It has always been our intention that a PFL programme should be transportable to any computer or distributed computer system that might be chosen to suit the implementation requirements, so any such constraint on portability was not acceptable.

At the same time, it is obviously necessary that code to drive each actual computer used in the implementation must be produced somehow. For the PFL programme itself, this requirement is satisfied by defining a PFL virtual machine which can be implemented on any chosen computer. This approach can cater satisfactorily for the pure PFL computations, which must always be performed in precisely the same way, but it is not suitable for the communications between the actual computer and the outside world, as each computer handles its communications differently. Different computers have different selections of communications ports, microprocessors use different conventions for memory-mapped and IO-mapped devices, there is a wide variety of interface adapter chips with diverse characteristics, and so on. Whatever machinery is used to effect the communication, it must clearly connect to the PFL virtual machine (or other implementation) and also to the real hardware of the specific processor used.

The *connection block* was introduced to help to describe this function. Its job is to provide the link through the computer hardware between the abstract variables which appear in a PFL programme as components of a machine's image and the real wires which convey the values of the variables between the computer and the controlled plant. Its nature is determined by the requirements of the programme and the configuration of the computer-plant interface. It may be as simple as a single instruction to read a byte from a machine port, or it could require complicated sequences of operations to set up and use interfaces, and to extract the required information from the data received.

Whatever the complexity, though, it is our contention that, given an adequately clear description of the PFL machine image on the one hand, and the hardware interface configuration on the other, it should be possible to infer the properties of the connection block, and from knowledge of the computer's instruction set to construct a programme which implements the required functions. It is in principle a routine operation, and should be effected automatically. The *connection block language*, or CBL, is intended to provide these descriptions; the CBL compiler should generate from the CBL description whatever is needed inside the connection block.

WHAT SORTS OF COMMUNICATION ?

A PFL programme may engage in two, rather different, sorts of communication. The details of both necessarily depend on the machine hardware, and therefore fall into the domain of the connection block, but they are likely to need handling in rather different ways. These are the communications between control programme and plant, which are likely to be fairly unstructured, to deal in bits and interrupts, and to be driven from the programme in terms of variables in the plant image; and communications with conventional communicating devices – other computers, terminals, more sophisticated plant such as robots, etc. – for which more or less standard communications protocols are defined, and which are driven from the programme by specific input-output instructions.

The two classes of communication are distinct; they should not be disjoint. It must certainly be possible to describe a machine as an image in a control programme, but nevertheless to communicate with it using a standard protocol. The conversion must still be performed by the connection block – and, therefore, it must be possible to specify the conversion in CBL.

In this discussion, I shall not discuss the internal functioning of the connection block, but only the interface between the connection block and the PFL programme. I shall concentrate on the communications which result from programme references to plant image variables. The more conventional communications pose less of a problem, as they can (I imagine) be handled by correspondingly conventional methods. But the world is full of surprises.

CHARACTERISTICS OF COMMUNICATION.

There are three properties of the signals transferred between PFL programme and machine which are useful for classification.

- The first, and perhaps most obvious, classification term is the **direction** of the communication : we classify operations into input and output actions. (I shall use those terms consistently from the viewpoint of the PFL programme.)

- The second classification is by **type**. So far as the interface is concerned, the important type distinction is between single-bit signals and signals which require many bits – numbers, characters, etc. For convenience, I shall refer to single-bit signals as logical, and the rest as numerical.

- The third classification is by **duration** : signals may be pulsed or continuous.

This is a nice tidy classification, and it is perhaps unfortunate that the language forms used to express communications actions in PFL programmes don't match it particularly well. Nevertheless, the classification is useful in categorising the communications which do occur. I shall discuss one by one the various syntactic forms in which communications can be specified in PFL.

Variable reference : A reference to a variable declared as part of a machine image is interpreted as an act of communication with the machine, and will be translated into actions which initiate appropriate input or output operations. The communication can be input or output, and the variable can be logical or numerical, but (except in the case of a WHEN(EVER) instruction) the signal must be continuous.

Whether every reference to an image variable results in an actual communication action is a matter of implementation. In fact, in some cases (such as commonly used values which change only slowly) it may be more convenient for the connection block to maintain a local record of the value, refreshed or transmitted by polling at appropriate intervals. (At present there is no provision for the connection block to handle such details, but as a routine and machine-dependent task it is certainly an appropriate candidate for inclusion in the connection block.)

Expressions which occur in the context of instructions discussed below may require somewhat different treatment.

WHEN(EVER) instructions : A WHEN or WHENEVER instruction introduces a procedure which is to be executed, once or repeatedly, in response to some event, or one of a set of events, in the environment. Each event is described as a syntactic item called an <indicator>(4), which is either a pulsed binary input variable (more commonly thought of as an interrupt) or a logical expression. Several events may be specified as potential triggers for a WHEN(EVER) instruction; in this case, it is reasonable to require that the multiplicity be handled within the virtual machine rather than the connection block, as only the virtual machine knows enough about the conditions to decide what has to be done. Notice that circumstances may be even more complicated if the same interrupt or variable appears in several different WHEN(EVER) instructions. I shall therefore suppose that only individual events need be considered here, and that the PFL virtual machine will sort out the mess. There are two cases :

- An interrupt must initiate execution of the appropriate procedure. It must therefore be conveyed from the connection block to the virtual machine, which must then schedule the appropriate WHEN(EVER) process.

- A logical expression must be evaluated from time to time. This can be effected either by polling, in which case the problem is equivalent to the time references discussed below, or the expression may be evaluated whenever one of its variables changes in value, which needs no reference to the connection block as it can be managed by a (cleverish) compiler.

Time references : IT'S, EVERY, etc. : Constructs for which the current time is important (as opposed to ordinary arithmetic on time variables) require access to a clock. Being dependent on facilities outside the virtual machine, this should reside either in the operating system or in the connection block. I have tentatively chosen the connection block, on the grounds that time signals may have to be acquired from another machine. If the time-of-day is explicitly required, it must be made available as some convenient form of variable. In many cases, though, the process only needs to know that some previously specified time has arrived, and at least two sorts of implementation are possible : either the time can be provided, as for the explicit time-of-day, and the polling left to the virtual machine, or the virtual machine can request the connection block to interrupt it when the desired time arrives.

SWITCH instructions : A SWITCH instruction is equivalent to an assignment to a logical output continuous image variable, but is deemed to be a much more natural way to talk about a switch. Early in the life of PFL there were suggestions that the same sort of instruction could be extended to multiple-position switches, by allowing a switch variable to be declared as a sort of enumerated type (and see Britton's description of a system specification technique(5) for an illustration of how it could be used). I don't think that lived long – I don't remember whether it survived into Peter's thesis. It didn't meet with unqualified approval because of the awkwardness of telling the connection block what to do with it. In hindsight, I'm not sure that that was a very good reason.

PULSE instructions : These seem to have got lost; I suspect that they should come back. They are needed to send a short signal – perhaps interpreted as an interrupt – to a machine, so they act on output binary pulsed variables.

I think that completes the survey. Even if I haven't covered the syntax exhaustively, I should have covered the communication types, so any omission should fit into one of the categories somewhere.

FITTING IT TOGETHER.

A summary of the process leading to the complete controller code, as I would expect it to be implemented, is shown as a lattice in Figure 1, where items lower down in the diagram are supposed to be derived from those higher items to which they are linked. The process can be seen as composed of several stages, which I have named on the right of the diagram; the names are only intended to be descriptive, so don't take them too literally, but they do indicate an order of development which I think is plausible. Figure 2 shows the same graph, rearranged to emphasise the boundary between PFL and CBL.

The links are intended to show logical dependencies, so if I have them right they are independent of details of the implementation. I am fairly confident of the links in the upper part of Figure 1, but some of those in the lower part may depend to some extent on implementation, and are therefore less certain. Observe in particular that in Figure 2, apart from the "specific to computer" links in the upper part of the diagram, only two links cross the boundary, both from the PFL programme. These represent transfers of information from PFL to CBL. One of them is necessary; the other I believe to be desirable, but in the next section I shall mention a possible implementation which would not use this link, thereby making the two parts more nearly independent. This direction of development is essential if CBL is to be used without PFL.

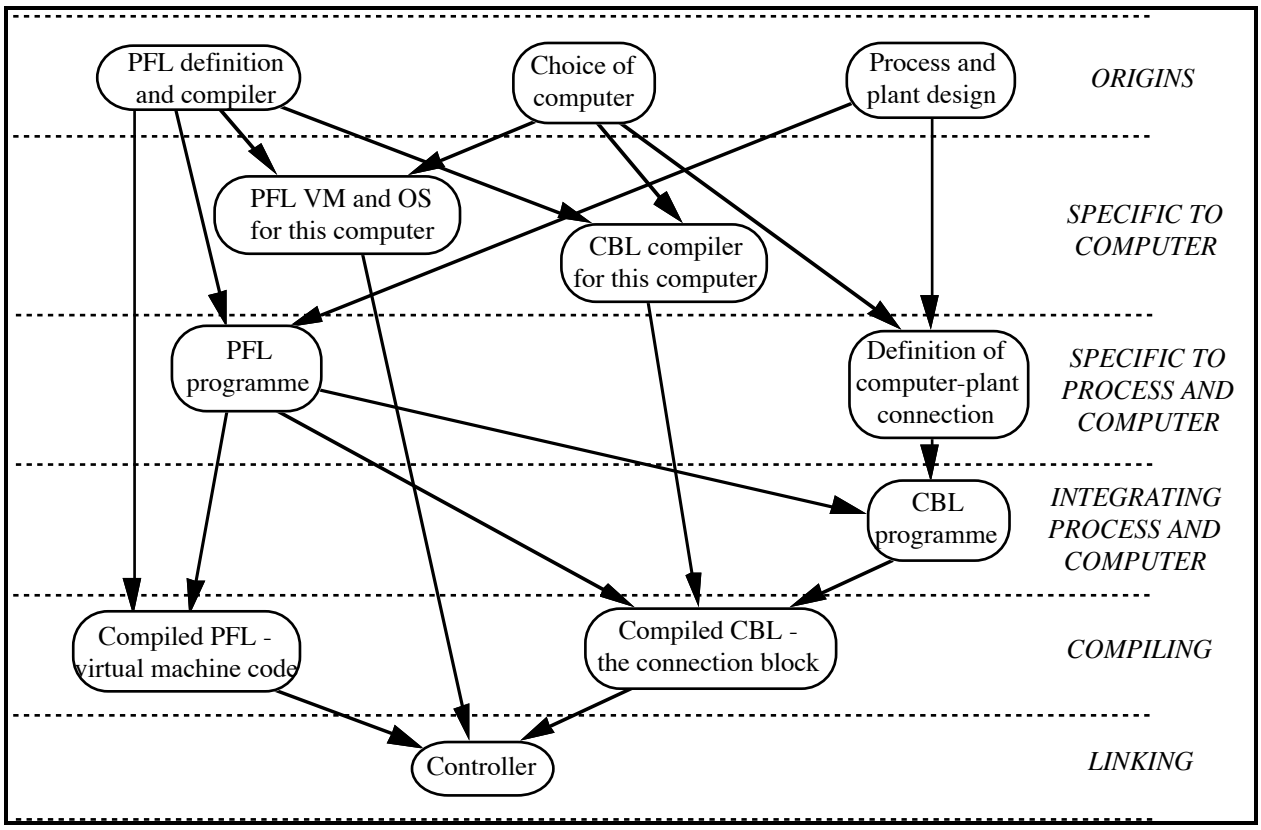

Figure 1

IMPLEMENTATION.

The ultimate aim of this discussion (which I shall not even approach in this note) is to develop an interface between virtual machine and connection block which will satisfactorily support the communication required – in effect, to specify a plug and socket. To do so, it will be necessary to identify a set of virtual machine "components" and corresponding things in the connection block which will perform the function required above.

The most important consideration in this design is perhaps the speed of the implementation. While PFL with its associated bits and pieces was specifically not intended in the first instance to address problems in which time was a critical factor, it was certainly our intention that the system should evolve towards a more satisfactory treatment of time and so towards a true real-time system. (On the subject of real-time operation, it is interesting to observe that it should now be possible to execute a PFL programme on a virtual machine *faster* than the same process could possibly have been executed as raw code on any readily available processor when the PFL work began ! The reality of real-time depends on where – and when – you stand.)

There are doubtless many different ways in which the link between virtual machine and connection block can be implemented, and if PFL-to-machine-code compilers are considered as well, there may be more. I shall discuss only two possibilities, one depending on procedure calls and one on integrating the two "machines".

An interface based on procedure calls is perhaps the most obvious candidate. It is familiar in its resemblance to the conventional operating system's supervisor interface, it is similarly easy to describe, and can in principle be used by programmes other than the virtual machine. An interface of this sort is proposed by Roger Kay(3). This sort of interface is clearly also easy to extend to include conventional communications.

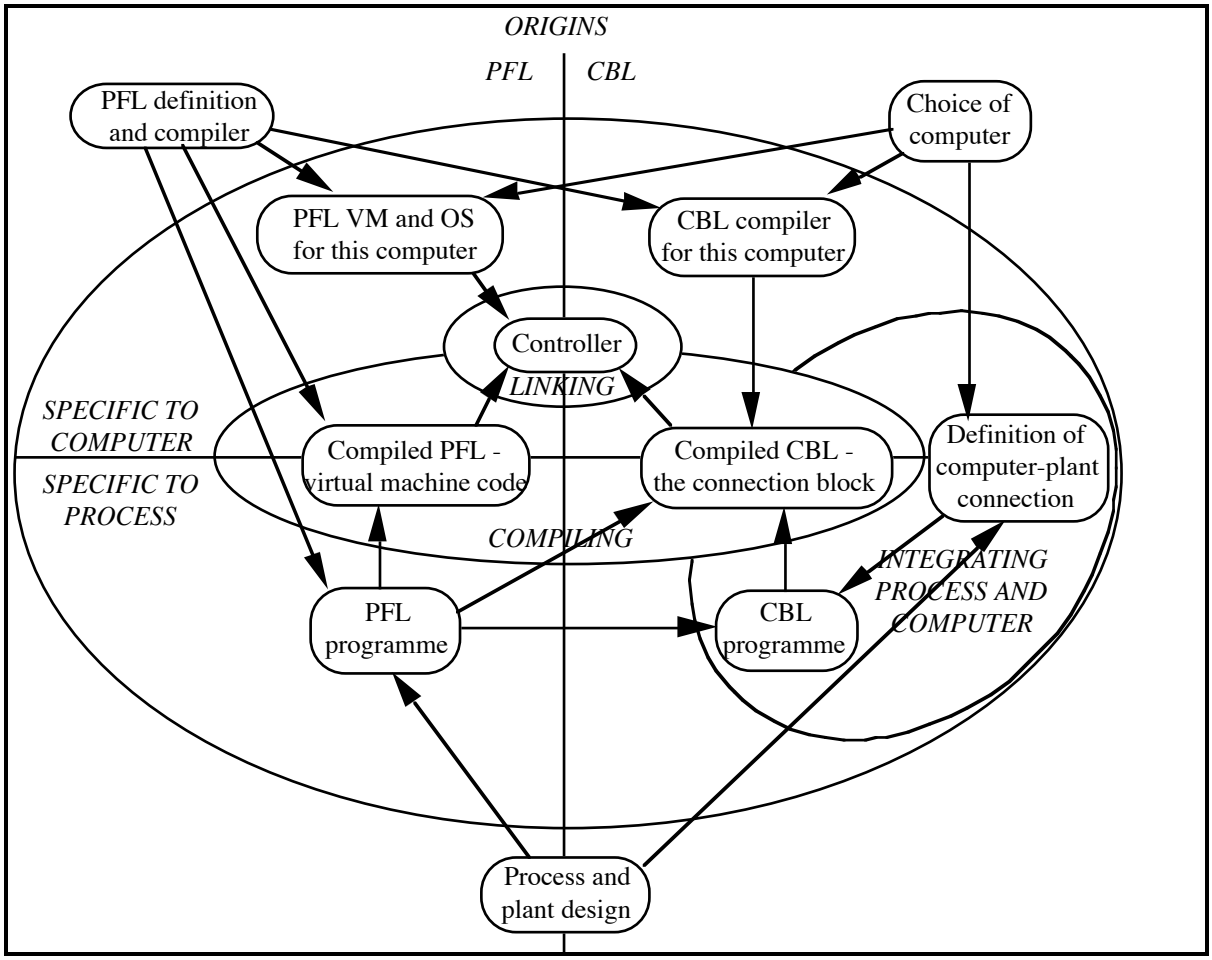

Figure 2

The difficulty with such an implementation for operations on the plant image is in knowing just how to define the transaction to be executed. A conventional input or output instruction has a clearly defined source and destination, so the supervisor calls used have arguments which define, directly or indirectly, an external source or destination stream (commonly a file) to be used and a buffer area to be filled or emptied. In a typical implementation, device drivers are supplied for all devices to be used. These use the basic input and output facilities of the computer to tailor the input and output transactions in ways appropriate to their devices. A programme which uses a device will usually first open it. This procedure sets up data areas both for transmission of data and for administrative operations, and also returns some identifying token which the programme will use in subsequent operations on the file. There are usually rather few supervisor calls available for communication between programme and file system (perhaps open, close, read, write, and control), but the system works well because that's what files do, and because, as everyone knows what files do, the supervisor calls are stable.

It is not clear that a similar system could work for the PFL image. There are at least four difficulties. (The first time I wrote this, there were two.) First, the granularity of PFL communications is much finer; transactions may involve any amount of data down to a single bit, so procedures to read, write, or respond to changes in words, bytes, parts of words, bits, or other units are necessary. The second is the variety of different sorts of protocol required for the communications – open, close, read, write, and control may not be sufficient. Some variables may be read or written on demand, some may require polling, some may signal interrupts, some may require quite elaborate sequences of operations to select different modes of the machines in the plant or of interface adaptors. The third is the fact that no device driver is available – the connection block *is* the device driver, and must be constructed by the CBL compiler. Fourth, it is not acceptable to expect the PFL programme to handle any of the processing. The PFL programme must be insulated from any peculiarities of the hardware, as it must run on any machine. Operations such as bit-field extraction, detection of interrupts by polling if necessary (or signalling that it's impossible if appropriate), and otherwise dealing with anything that might vary from computer to computer must be tackled within the connection block.

It may be that none of these difficulties is insuperable, but the question is how the required system calls are to be defined. To attempt to cater for all possible communication requests would produce a rather large connection block. Commonly, of course, only a few of the possible combinations of protocol and data granularity will be used by an individual control programme, but we don't know which before the system is defined. To determine which combinations are required, we must analyse the machine image in the PFL programme, which is presented as part of the CBL programme. It may also be necessary to analyse the PFL programme itself to distinguish between the ways in which the variables are used. The CBL compiler can then define a set of appropriate system calls, which must then be communicated to whatever wants to use them. If this is a PFL programme, that's easy enough, but either the compiler will have to know about them before it can run, or they will have to be built into the virtual machine. If it isn't a PFL programme, then they must be published in some accessible form, after which they can be written into the programme. This forces the sequence { CBL programme, compile CBL, write control programme }, which is survivable, but a nuisance.

Roger's recommendation(3) circumvents some of this by (apparently) passing both a pointer to a buffer area and pointers to the names of the image and of the variable within the image; further details are not given, but presumably the connection block will look up the names in its own version of the symbol table to identify the procedure to be followed. This is not conducive to rapid execution. Perhaps a faster implementation could be generated with some clever preprocessing during the linking phase, but again this is likely to compromise generality. Some acceleration may be achievable by providing a sequence of calls analogous to open-file calls to be used to associate identifying tokens with the various fields; this is perhaps the best compromise if a CBL compiler independent of PFL is required, though more careful evaluation would be highly desirable before embarking on an implementation.

The alternative is to recognise the important principle of early binding(6) which is conducive to efficient implementation. As applied in this case, it leads to the conclusion that, other things being equal, the speed is likely to be greater as the virtual machine and connection block are more closely integrated, and that integration and generality are antagonistic. For a fast implementation, we should therefore try to unite the connection block and virtual machine as closely as possible. Direct links between the required components can be made using information available from the PFL programme and found by the PFL compiler, perhaps communicated through the virtual machine code file. This can incorporate any information that might turn out to be handy, including a symbol table for the current programme. The linker can now identify corresponding image components in the connection block and virtual machine code files, and construct direct connections accordingly. (This is the significance of the link from PFL programme to connection block in the figures. In practice, as suggested by the reference to the symbol table, it might well be more convenient to use the virtual machine code, but logically the information is implicit in the PFL programme.) Of course, this does rather impair its potential as a system which could be used for programmes other than the virtual machine.

That paragraph leaves a lot of questions open, and I don't intend to close them here. Just as an example of a possible design, though, one might suppose that the virtual machine should have a set of specific "hardware" operators for communicating with the connection block. The PFL compiler could then generate tables which identify arguments passed by the operators to the connection block with certain actions, so that when the connection block and virtual machine are linked the proper operations can be identified and built into the generated code.

REFERENCES.

1 : P.A. Sergent-Shadbolt : A new computer language for process control, Ph.D. thesis, Auckland University, 1985, pages 6-2 to 6-8.
2 : G.A. Creak : *PFL : Progress Report ?* ...., unpublished Working Note AC49, April, 1986.
3 : R. Kay : *CBL : Discussion and implementation*, 07.473 Assignment 2 report, 1993.
4 : G.A. Creak : *PFL Syntax, yet again*, unpublished Working Note AC79, March, 1991.
5 : K.H. Britton : "Specifying software requirements for complex systems", *Proceedings of IEEE conference on specifications of reliable software, 1979*, reprinted in *Real-time Software* (R.L. Glass (ed), Prentice-Hall, 1983).
6 : C.U. Smith : "Independent general principles for constructing responsive software systems", *ACM Trans.Comp.Sys.* **4**, 1 (1986).