WORKING NOTE AC101

Alan Creak
20 August 1996

STATES IN PDL

An alarming fact has come to light : it is not clear that the state-based machinery previously described for programme control in PDL will work. Here I say why, try to do better, and end up reassured.

This sort of thing always happens when I'm forced to look closely at PDL. This time, I'm looking because I'm trying to work up a seminar on PDL, and I've come to a point which I can't explain.

THE PROBLEM.

The closest we have to an official definition of the use of states in programme control is a passage in an early report[1]. Thereafter, the matter is regarded as settled, with a number of references in passing, leading up to the confident assertion[2] that "Branching and iteration are dealt with by the state-based execution algorithm". Unfortunately, it seems that nobody has ever worked out just how that confident assertion can be put into practice.

I think that it *can* be put into practice, but some cases are clearer than others, and the (very informal, and probably restricted to me) model of PDL's execution on which the assertion was based is no longer current. In this note I hope to sort it out to my satisfaction, and make it available to whom it may concern for examination in more detail than has hitherto been possible.

I should add that someone has previously expressed strong scepticism about the practicability of the state approach as the sole means of programme control; I think it was Roger Kay, which accounts for my inserting the confident assertion into our joint report. In hindsight, he was right to be sceptical, and I should have taken more notice of his comments. Thank you, Roger. (Or whoever.)

MORE DETAIL.

The principle behind the state-based method is that any transition within the programme is characterised by information about the current state of the programme and some signal which identifies the reason for the transition. This is embodied in the execution algorithm in two ways. They were originally intended to be the same, but on analysis turn out to be slightly different in detail.

At the lowest level, every instruction is identified with a state which has two components, and is written as { *Normal*, *Fault* }. *Normal* is the current state of the execution; it identifies the instruction in the process currently being executed. Its value while executing instruction N is *Instruction-N*. *Fault* says something about the overall state. Its value is usually *NoFault*. Unless otherwise instructed, at the end of any instruction the normal state is changed to *Instruction-(N+1)*.

When an instruction is complete, the next instruction is chosen by a dispatcher. It searches the available instructions for one with the state identified by the current values of the two state components; if nothing untoward happens, this is Instruction (N+1). All that is so simple that all the details, state names, etc. can be omitted from the visible programme and defined automatically, leaving a programme which looks like a conventional sequential programme. That's sensible, because it's easy to read that way.

Any departure from this normal mode of execution requires that an instruction "out of sequence" be specified as the next to be executed. Following the same algorithm, that means that some other state must be defined as the next state. A state can therefore be given another name (a synonym, not an alternative) by prefixing the instruction in the programme with

```
State { <normalstate>, <faultstate> } :
```

Arbitrary (descriptive, one hopes) state names can thereby be attached to any instruction, so that an instruction can easily be addressed from anywhere else. So far, I've assumed that the scope of such names

(like the scope of the automatically generated names) is the programme for this machine. If large numbers of names are required, this could become difficult to sustain, but very long sequences of instructions which cannot be expressed as smaller machines did not seem to be very common when we thought about it. I can't remember the evidence, though, so this point must be regarded as not yet settled.

The theory then says that the code of any state can change the execution sequence by changing one or other of the two state components. That's sensible enough too. But what the theory doesn't say is how to do it. I can invent a reason for the omission, though I don't remember whether it was the real reason : it is that every machine's instruction set is in principle different, so there's no guarantee that an arbitrary machine can do whatever corresponds to changing its state arbitrarily. Generally in PDL, a machine has no predefined vocabulary, so if you want some verb to be available, it's up to you to make it work if you can. But the specification cannot *require* that any verb *must* be available because some systems might be too simple for that to be possible. In some cases, you might choose to implement vocabulary verbs such as "setstate" and "if", which would solve the problem, but that's your business. So my position was that I can't tell you how an arbitrary machine deals with loops etc., because I don't know.

It is now clear that I can't just walk away from the question like that. As the functioning of any but the simplest system is likely to depend fairly critically at some level on such abilities as branching and conditions, and they are likely to be required in almost all entities which can be called controllers, I must at least go some way to discuss possible means of implementation. I shall do so item by item. Please observe carefully that the descriptions I shall give are *not* in any sense specifications of what must be done; the implementation details remain undefined, as they must to preserve generality. Here I merely illustrate the general problem, and suggest one possible solution.

SETTING THE STATE.

This is the simplest requirement. It is necessary to implement an unconditional branch, and, once available, is likely to prove useful in the more complicated instructions too. A simple possibility is illustrated in this sequence :

```
     .....
     Instruction 1.
State { GoRound } :
     Instruction 2.
     Instruction 3.
     SetNormalState to GoRound.
     Instruction 5.
     .....
```

(I have omitted the "normal" part of the state; that would perhaps be a reasonable provision in an implementation in the interests of clarity.)

The `State` directive identifies *GoRound* as a synonym of *Instruction-2*. Notice that `SetNormalState` is counted as the fourth instruction, and that Instruction 5 is inaccessible except by an instruction `SetNormalState to Instruction-5`.

`SetNormalState` is an ordinary instruction; it must be implemented like any other instruction, and "SetNormalState" must be looked up in the appropriate database like any other verb. Its implementation must suppress the usual automatic state change which occurs after an instruction.

CONDITIONS.

I had tried to avoid explicit conditional constructs in the system, on the grounds that it was the thin end of a wedge which would open the way to arbitrarily complicated programmes which it would be beyond the capacity of a diagnostic system to comprehend. It also seemed to me that conditional branches would be caused by the arrival of messages of different types, so could be dealt with by linking the type of the received message with the next state desired.

I don't find either of these arguments persuasive now. It's still true that complicated programme structures are difficult to interpret, but some sort of conditional testing is necessary; it is perfectly reasonable that a control system should – for example – compare two temperatures to determine what to do next. The same example also illustrates why it is unreasonable to insist that conditional results should be associated with incoming messages, as there is no reason why temperatures should be continuously polled.

A conditional construct of some sort therefore seems to be inevitable. Given that, my preference is to evolve a variant of the `SetNormalState` instruction rather than to introduce a new operator. A possibility is :

```
SetNormalState to <state> if <condition>
```

I think that's about the simplest way to do it. That's important for comprehensibility. It's true that any other conditional construct can be reduced to something like that, so could in principle be used instead without increasing the complexity of the language, but recall that the language we're designing is intended to be interpreted by machines, understood by machines, and composed by machines in the grand plan. It seems only sensible to begin with the simplest instruction set.

For similar reasons of complexity, I'm not in favour of accepting the `if <condition>` suffix on any other instructions. It's easy for machines to interpret, but understanding and composing are perhaps just a bit harder, and the system becomes more than minimally complex.

FAULTS.

This case is interesting for historical reasons. In early attempts at implementation (particularly, I think, in Adrian's work, but perhaps in Mark's too – I haven't got the theses here, so can't check) the vocabulary was implemented as a set of function calls, with all the bits and pieces passed as strings. This was a long way from the interpreter model of PDL, but served as a sort of test of the language. (It sort of passed.) The idea then was that if anything went dreadfully wrong while executing one of the functions, it would return a corresponding fault state, which would set the overall state to { *Thisinstruction*, *SomethingWrong* }, and leave the dispatcher to direct execution to an appropriate fault handling instruction if one was implemented. This is the mechanism described in the report[1].

That went some way to demonstrating that the state mechanism could handle faults sensibly, and that it was easy to incorporate fault-handling procedures, but it opened up a brand new way to incorporate facts about the system in impenetrable code : just write them into the functions using – say – C. In particular, the association between a response from a machine and a corresponding fault state was hidden, and so were any difficulties in implementing the fault handling.

I conjecture that the consequent impression that fault handling was easy led me to be overconfident about the abilities of the state-based model, and to assume that it was all easy. The same idea can be extended directly to conditional branches, as a function can just as well perform a test and return a corresponding normal state, so that's easy too. This is the informal model I mentioned at the beginning, and which I'm trying to improve.

I therefore ask : what happens if a controller is waiting at Instruction N for a signal, and it receives an unexpected response ? (That can include no response, provided that the machine on which the controller is implemented is clever enough to manage time-outs.) The model answer, which can be elaborated to any required degree if certain responses can be foreseen and good reactions are known, is to set the fault state to some catch-all value – say, *UnexpectedInput* – and let the dispatcher handle it. The complete state would then be { *Instruction-N*, *UnexpectedInput* }, and an instruction identified with that state would be executed if one has been defined. If not, a general error-handling procedure must be executed. If the fault can be identified more precisely, a more precise fault state can be used; in either case, a branch from the normal sequence is executed.

A syntactic form remains to be determined. If the fault information is in variable parts of the message rather than represented by an immediately recognisable message format, the forms already introduced (extended to permit changes to the fault state) should be sufficient. For example :

```
Receive Expected-message from SomeMachine.
SetFaultState to SomeFieldValue if Some-field-in-
                Expected-message <> Expected-value.
```

If the error message is of quite a different structure from that of the expected message, the first instruction will fail, and a more appropriate syntactic form might be :

```
Receive Expected-message from SomeMachine else
                SetFaultState to UnexpectedInput.
```

In either case the desired effect is achieved.

(As an illustration, I have used that fault state *SomeFieldValue* in the first instruction, where a rather specific test has been applied and failed, but the more general *UnexpectedInput* in the second case where the fault is not specifically defined. In each case, it is the responsibility of the programmer, human or machine, to ensure that code is provided to deal with the states { *Instruction-N*, *SomeFieldValue* } and { *Instruction-N*, *UnexpectedInput* }.)

Ideally, the contents of the unexpected message will be available somewhere for analysis, but that's a different topic. Bear in mind that all this has to be implemented on some arbitrary machine which we don't know about beforehand. While it's likely that anything used as a controller will be able to manage such operations, there's no guarantee.

DISCUSSION.

- I think it, or something rather like it, will work. I think I've covered the significant cases, and that the answers I've suggested are at least plausible.

- I do not claim that my answers are simple or elegant, and am very willing to consider alternatives. In so doing, though, I shall bear in mind that the code is to be produced – or, at least, produceable – by a machine, so that aesthetic appeal alone isn't a strong criterion for acceptance.

- In her project report[3], Natalie suggested that basic PDL instructions could usefully be predefined. In general, that can't be done, because there's no guarantee that an arbitrary machine will be able to implement an arbitrary instruction. On the other hand, in practice it is clear that many of the machines for which PDL programmes might be written, and particularly those used as controllers, will be quite capable of supporting a selection of the instructions commonly required for controllers. In these cases, Natalie's suggestion might have a lot of merit. It would perhaps be useful to define a collection of standard facilities which need not be implemented in particular cases, but for which certain standard forms could be recommended. Examples are instructions such as `send`, `receive`, `calculate`, `set*state`; declaration of variables; facilities to save messages; and doubtless others. While the very nature of PDL guarantees that uniformity cannot be enforced, informal standards could save time.

- I haven't discussed what happens if an instruction arrives from a machine for which nothing is waiting. That's because there's nothing we can do about it, except raise a general alarm. If we were at all aware that it might happen, we'd have allowed for it; the very absence of an appropriate receiving instruction means that we don't know anything about it, and have no predefined strategy to handle it.

- There is another possibility which might be worth considering. It isn't quite the same as the state model I've been discussing, but shares many of its properties. In this implementation, the states are

preserved, but the transitions are initiated by the receipt of signals by the dispatcher, and it is the signal which specifies the new state.

In this view, the internal moves from instruction to instruction are unified with the receipt of signals from the external world. An internal instruction by default sends a signal containing the state of the next instruction when it finishes, and the language must contain instructions analogous to `set*state` to send other states explicitly or to send fault state values. The result of that is to reproduce the behaviour described above.

Now, though, introduce a general message handler. It must catch messages as they arrive from outside, classify them, and associate them with appropriate signal types. Some of these will correspond to normal states, when the signal is expected; others will correspond to fault states, when there is something amiss.

This is a more complicated system, and could not run at all on a very simple machine. It requires a message handler which must be given the information it needs to classify each input received. With the current programme structure, that would need a compiler, I think, and would seriously impair the uniformity of the PDL structure. Could the benefits (should there be any) be worth it ?

REFERENCES.

1 : G.A. Creak : *Information structures in manufacturing processes* (Auckland University Computer Science Department, Technical Report #52, February, 1991), pages 62ff.

2 : G.A. Creak, R. Kay : *PFL and PDL : two languages for process control* (Auckland University Computer Science Department, Technical Report #104, December, 1994), page 20.

3 : N.R. Spooner : *Design and implementation of a Process Description Language (PDL) interpreter* (Auckland University Computer Science Department, 07.485 Project Report, 1995), page 6.