WORKING NOTE AC116

Alan Creak
24 October 1997

STATES IN PDL REVISITED

My previous note on States in PDL[1] is probably far too pessimistic. Here I describe a means by which all the problems identified there can be overcome without introducing explicit conditionals, and also broaden the idea of state components.

This sort of thing always happens when I'm forced to look closely at PDL. This time, I'm looking because I'm building an abstract PDL machine to assemble the department handbooks[2], and I've come to a point which I can't explain. If that sounds familiar, check the previous note[1].

CONDITIONS.

My objection to including explicit conditional constructs in the PDL repertoire was that it could lead to a programme which would be less easy for a machine to understand. I still think that's true, but there remains the requirement answered by the conditional construct – that is, to alter the behaviour of the programme according to circumstances. My proposal here is to replace the conditional by what amounts to an assignment of state from a decision table, followed by selection of the next procedure for execution according to the usual rules.

The particular advantage of this approach is that all the possibilities are clearly identified, and it is easy to check – perhaps even automatically, if that should be desirable – that all contingencies have been covered. The state machine behind PDL is strongly emphasised, as the new feature corresponds closely to selecting the next state by taking account of the system's current state. The result is a method resembling tha tabular approach recommended by Britton[3] as a design technique for real-time systems. It also, perhaps not entirely accidentally, has something in common with an earlier suggestion for a generalised case statement in the Small language[4].

In the interests of simplifying inspection and interpretation of the tables, it is desirable that instructions which incorporate them should make all components readily visible, with a simple layout. I therefore propose syntax of a form something like this :

```
SetNormalState from
[
<condition> : <state>
<condition> : <state>
<condition> : <state>

......

]
```

Using this format, the example condition from the previous note can be written thus :

```
SetNormalState from
[
<condition>        :    <state>
<~ condition> :    <>
]
```

It is easy to check that the disjunction of all the conditions is true; the symbol <> is used to denote the identifier of the next procedure, normally assumed if no other instruction is given. An alternative, equally clear, is :

```
SetNormalState from
[
<condition>        :    <state>
else               :    <>
]
```

The advantage of "else" is that it is universal, and will work for any combination of conditions using quantities of any sort. The disadvantage is that it gives less information than an explicit condition, and does not force a thorough analysis of possibilities.

There is an obvious extension of this notation to decision tables of higher dimension :

```
SetNormalState from
[
<conditionA1> :
     [
     <conditionB1> :    <state11>
     <conditionB2> :    <state12>
     ]
<conditionA2> :
     [
     <conditionB1> :    <state21>
     <conditionB2> :    <state22>
     ]
<conditionA3> :
     [
     <conditionB1> :    <state31>
     <conditionB2> :    <state32>
     ]
]
```

And, of course, more if you want it. The table need not be rectangular, and `else`s or `<>`s can be introduced at any level.

A generally similar method can be used to branch on a value returned in a message, but in this case a new feature is more prominent. This is the possibility that the requirement for the exceptional case might be to set the fault state rather than the normal state. I have considered the possibility of incorporating this, potentially common, case into a convenient syntactic form, but on the whole I think it's better to avoid the complication and maintain the simplicity of the syntax – which is quite sufficiently complicated as it is. The example from the previous note becomes :

```
Receive Expected-message from SomeMachine.

SetNormalState from
[
<SfiEm = "one">    :    <StateOne>
<SfiEm = "two">    :    <StateTwo>
else               :    <>
]

SetFaultState to SomeFieldValue.
```

("SfiEm" stands for the "Some-field-in-Expected-message" in the previous note.) If it is desired to preserve the logical check without an else clause (which does it automatically), it would be necessary to require that a complete list of possible values of `SfiEm` be available – most appropriately, in the Machine Type Database[6].

STATE COMPONENTS.

This method extends my original idea that the text of the returned message could itself be used more or less directly as the label of a subsequent state. That the new proposal really is an extension is demonstrated by this generalisable formulation of the original idea using the new syntax :

```
Receive Expected-message from SomeMachine.

SetNormalState from
```

```
[
<SfiEm = "one">    :    <one>
<SfiEm = "two">    :    <two>
else               :    <>
]

SetFaultState to SfiEm.
```

Notice that the destination state labels are identical with the text of `SfiEm`.

There is no reason in principle why the original form could not be used, but a possible difficulty of doing so is that if the same machine is used in more than one place in the programme then there might be confusion in the new states defined; if the message text is R, then the only new state which will be specified when R is received is *{ R, NoFault }*, no matter which instance of using the machine is concerned. This is fairly unlikely to be satisfactory.

If we are content to suppose that the message identifies a fault state, we can use SetFaultState to escape from the trap. Setting the fault state gives a set of states with names like *{ Instruction-N, R }*, which certainly identify a unique set of states. To do so, though, is to blur the idea of the fault state by including some normal states in the same category. This is potentially misleading; a syntactic device that doesn't confuse different things would be preferable. This suggests an extension of the state naming principles, so that we can define (say) "branch states" as well as fault states.

There is no particular difficulty about doing so, and it does not appear to complicate the system in any significant respect. All we require is a new state-setting instruction of the form :

```
SetWholeState to { SfiEm, BranchState }
```

where `BranchState` is an identifier which can be chosen arbitrarily to distinguish the successors of this state from others which might be defined. As before, the state labels must appear in the programme, each in a `State` directive associated with the instruction belonging to the labelled state[1]; as these labels are regarded as synonyms for the automatically defined labels, there is no reason why some state should not have several equivalent labels if it proves convenient. Given this notation, the system can always check that the explicitly named instructions are named uniquely, so no harm can come of this instruction.

DISCUSSION.

The main proposal of this note is the notation for "decision tables". Is that a good idea ? It avoids explicit conditionals, and gives a very clear specification of what is expected and what alternatives are available. It seems to me to have much merit.

In contrast, setting the next state to a message from some external source can be completely inscrutable, because there need be no indication at the instruction which sets the state of what states might result, so the identity of the next action to be executed is completely unknown. This amounts to the introduction of a sort of computed or assigned **goto**[5], which is arguably to be deplored – though even the two sorts of **goto** instruction in Fortran include lists of possible destinations.

The comprehensibility argument is not very significant unless the artificial intelligence procedures are used to "understand" the system processes with a view to (perhaps) diagnosing faults – or, perhaps, unless I want to prove the programmes correct, which has hitherto not been a feature of PDL, but who knows ? Nevertheless, while those extensions of the simple PDL machinery might be remote prospects, we don't want to compromise them by building intrinsically incomprehensible systems. A possible way out is to follow the earlier suggestion, and to permit a list of possible labels to be associated with the `SetNormalState` or `SetWholeState` instructions. Such a list could be optional, so we can forget about it unless and until we need it.

As a final comment, it is important to bear in mind that we can't lay down fixed rules for any syntax in PDL, for there is still no way to prevent people from implementing any operation they might want on their own machines. I hope that by exploring some alternatives and suggesting sound syntactic

methods others (should there ever be any) will be encouraged to construct their programmes in ways which lend themselves to automatic understanding and, at least, verification.

REFERENCES.

1 : G.A. Creak : *States in PDL*, unpublished Working Note AC101 (August, 1996).

2 : G.A. Creak : *Background for a document generator*, unpublished Working Note AC115 (September, 1997).

3 : K.H. Britton : "Specifying software requirements for complex systems" (*Proceedings of IEEE conference on specifications of reliable software*, 1979), as reprinted in R.L. Glass : *Real-time Software* (Prentice-Hall, 1983).

4 : G.A. Creak : "Suggested modifications and likely linguistics", in G.A. Creak (ed) : *Thinking Small*, Auckland University Computer Centre Technical Report #15 (1979).

5 : for example, S. Lipschutz, A. Poe : *Programming with Fortran* (McGraw-Hill, 1978).

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