# Computer Science 773

# Robotics and Real-time Control

# PDL

## ORIGINS.

PDL ( Process Description Language ) grew from concern that software for control purposes – particularly those for manufacturing systems – was becoming very complex, and therefore very hard to change. At the same time, pressure for rapid change was increasing, so programming departments were becoming overloaded with consequent poor service and increased probability of errors.

A study of the problem suggested that a possible explanation was that the problems come from the interment of much of the information about the manufacturing system in the programme code. Though the programme is determined by information drawn from a very wide range of sources, from the nature of the product to the geography of the plant, very little of this appears explicitly in the programmes. Instead, much has been taken into account in decisions taken before the code is designed. Because of that, any change in such external conditions will require changes in the programme, but it is necessary to work through the decisions again before it becomes clear what changes are necessary.

That's a gross oversimplification of the argument ( or perhaps the argument is a gross oversimplification of the problem ), but there is some sense in the notion that more explicit representations in programmes of factors which determine their constitution make it easier to understand what's going on. Consider :

- If the engineers could see what the programme was about, they could check it;
- If the engineers could see what the programme was about, they could write little bits;
- If the engineers could see what the programme was about, they would be able to adapt it to new products and new machine tools;
- If the programmers could see what the programme was about, they could find faults more easily;
- If the programmers could see what the programme was about, they could adapt it more easily when new computers and software were acquired;
- If the system could see what the programme was about, there would be a chance of automatic checking;
- If the system could see what the programme was about, there would be a chance of automatic fault diagnosis;

PDL is therefore built on the notion of *information accessibility*.

It is argued that the information is important – that's what determines the programme. Therefore, for a robust system, the programme must change "automatically" as the information changes. So why don't we put the information in databases, and work out the programme from there automatically ?

In a manner of speaking, that's what happens now. The database is the specification and documentation ( should there be any – one of our difficulties ); "automatically" means having some programmer decide every time there's a change how it should be encoded in C. We're not trying to do anything dramatically new in principle – just to clarify, systematise, and automate the traditional process.

By the same token, we don't expect ( or need ) to do the whole automation job in one go; we're quite happy to leave a large proportion of the system implemented on people rather than computers. "All" we require then is some formalisation of existing things, probably amounting to writing down facts and procedures in simple forms – there are no revolutions in prospect. Yet.

That sounds a bit familiar. The information ALSO contributes to the diagnostic part of the system, where it's essential raw material for the attempts to interpret unexpected phenomena. In that context, we talk about knowledge-based systems, where the special feature is the separation of the knowledge base from the processing ( "expert system shell" ).

So perhaps we can use all the good arguments for expert systems ( should there be any ) in favour of our approach ?

## DEVELOPMENT.

We'd design a "source language" for the control programme, which – for the time being – would be written by an engineer, but could in principle be produced by an automatic implementation of the earlier parts of the overall design process.

We'd write a processor for the language which could take into account the various sorts of information we've discussed, assumed to be recorded in databases, and drive the system.

Notice that it has to produce code for a *distributed* computing system. The programme will describe the manufacturing process for one product, but that will be implemented by a wide variety of computers scattered around the plant. The "one product"-ness is important in several ways :

- In the manufacturing, it's important because it describes the complete process, and it's the entity we'd want to use for programme proofs, or other validation tests. In older systems, no such statement of the process exists; engineers know what has to happen, but by the time it got to the computers it is fragmented into bits for each separate computer.

- It can be of assistance as a knowledge base for fault diagnosis. If there's a fault in the product, then a view of the complete process is required to formulate hypotheses on where the fault could have originated; again, a set of separate programmes isn't very helpful.

- It emphasises that there's one more job to be done : coordinating the products which are moving along the production line at the same time. This emphasises the fact that a manufacturing system does at least two jobs – as well as making the product, it controls the behaviour of the plant, the movement of parts through it, and so on.

## CONSEQUENCES FOR THE LANGUAGE.

What are the constraints on the language ?

| | | | |
|---|---|---|---|
| • | Comfortable for engineers | - so that they can write it or check it. | ( Realistic, not patronising ! ) |
| • | Machine-writable | - so that it can plausibly be composed by ( non-existent ) "higher level" software. | Should be low level; "Assembly" ( literally ! ) language. |
| • | Machine-comprehensible | - so that it can be used as a database describing the manufacturing process. | Should include a description of what machines do, and how they do it. |

- and general, and portable, and sufficiently expressive, and ......

The first question is therefore "how do engineers like to write their programmes ?". Why not ask the engineers ? One example looked more or less like this :

```
Machine :     Detector.
Stimulus :    A bottle arrives.
Action :      Send "Found a bottle" to the Controller.

Machine :     Robot.
Stimulus :    Receive "Move bottle from Detector to Filler"
                from the Controller.
Action :      Move the bottle to the Filler,
              Send "Operation complete" to the Controller.

Machine :     Filler.
Stimulus :    Receive "Fill the bottle" from the Controller.
Action :      Fill the bottle,
              Send "Finished" to the Controller.

Machine :     Robot.
Stimulus :    Receive "Move bottle from Filler to Packer"
                from the Controller.
Action :      Move the bottle to the Packer,
              Send "Operation complete" to the Controller.

Machine :     Packer.
Stimulus :    Receive coordinates from the Controller.
Action :      Receive "Pack" from the Controller,
              Pack the bottle at the coordinates.
```

( That's a free interpretation of the style applied to a fictitious machine called a Gherkin Packer, but it preserves the main features. )

## REFINEMENT.

There's more to it than that, but the rest can be done in a fairly automatic way. The first step, which follows from the control structure of the plant, is to insert the controller activities :

```
Machine :     Detector.
Stimulus :    A bottle arrives.
Action :      Send "Found a bottle" to the Controller.

Machine :     Controller.
Stimulus :    Receive "Found a bottle" from the detector.
Action :      Send "Move bottle from Detector to Filler" to
                Robot.

Machine :     Robot.
Stimulus :    Receive "Move bottle from Detector to Filler"
                from the Controller.
Action :      Move the bottle to the Filler,
              Send "Operation complete" to the Controller.

Machine :     Controller.
Stimulus :    Receive "Operation complete" from the Robot.
Action :      Send "Fill the bottle" to the Filler.

Machine :     Filler.
Stimulus :    Receive "Fill the bottle" from the Controller.
Action :      Fill the bottle,
              Send "Finished" to the Controller.

Machine :     Controller.
Stimulus :    Receive "Finished" from the Filler.
Action :      Send "Move bottle from Filler to Packer" to
                Robot.

Machine :     Robot.
Stimulus :    Receive "Move bottle from Filler to Packer"
                from the Controller.
Action :      Move the bottle to the Packer,
              Send "Operation complete" to the Controller.

Machine :     Controller.
Stimulus :    Receive "Operation complete" from the Robot.
Action :      Send coordinates to the Packer
              Send "Pack" to the Packer,
              Calculate next coordinates,
              Restart.

Machine :     Packer.
Stimulus :    Receive coordinates from the Controller.
Action :      Receive "Pack" from the Controller,
              Pack the bottle at the coordinates.
```

What more do we need ? Not an exhaustive list, but it'll do :

| *The programming language – what it needs for its machine :* | | |
|---|---|---|
| • | Standard vocabulary. | "English", associated with the machine. |
| • | Means of associating actions with the vocabulary. | The machine database. |
| • | How to make a machine do things. | "Machine language", and how to get it there. |
| *Programme structure – what it needs for the process :* | | |
| • | Programme structure – conditions, loops. | - but keep the basic simplicity. |
| • | Manage faults. | - particularly external faults. |

So a machine database must include :

- What the machine can do, as English descriptions. ( Not essential at this level, but documentation. )
- The corresponding instruction which must be got to the machine, described in "English".
- The corresponding instruction which must be got to the machine, described in networkese.

( The last two define the programming language to be used when programming for the machine, and the translation to be used by the parser which handles it. THESE CAN'T BE PREDEFINED – so the language can't have a defined vocabulary. Better called a framework ? )

- and a network database must include :

- How the components are connected together.

- and the controller's database must include :

- How to send a message into the network.

The overall programme must be split into separate programmes for the individual machines of the manufacturing system. The programme for the controller ( which is the machine of greatest interest from the computing side ) is something like this :

```
Trigger

-- Await a signal from the detector;
              Receive "Found a bottle"
                : from Detector.

Procedure

-- Tell the robot to move the bottle to the filler's
                workbench;
              Send move instruction
                : to Robot;
                : carry bottle;
                : from detector station;
                : to workbench.

-- Await the "finished" signal from the robot;
              Receive "operation complete"
                : from Robot.

-- Tell the filler to fill the bottle.
              Send fill instruction
                : to Filler.

-- Await the "finished" signal from the filler;
              Receive finished
                : from Filler.

-- Tell the robot to move the bottle to the packer;
              Send move instruction
                : to Robot;
                : carry bottle;
                : from workbench.
                : to packer station.

-- Await the "finished" signal from the robot;
              Receive "operation complete"
                : from Robot.

-- Send the coordinates to the packer;
              Send message
                : to packer
                : contents box coordinates.

-- Tell the packer to pack the bottle into the box;
              Send pack instruction
                : to Packer

-- Calculate the coordinates for the next bottle.
              Calculate next
                : box coordinates.
```

That describes the controller's actions for the manufacture of a single product. Other programmes can be derived for the other machines involved, but if they're simple machines the programmes simply describe their behaviour, and are not executed by any sort of computing machinery. For more elaborate machines ( robots, NC tools ) the machine programmes must be conveyed to the machines after transformation into the machines' own languages.

## PROCESSING.

These programmes can then be compiled or interpreted or otherwise executed to control the machinery of the manufacturing system. Here I use "processed" to mean any of these, because essentially the same operations must be performed in all cases.

The fundamental problem is to convert one of those instructions into a sequence of actions which has the desired effect. In the case of a controller, this is usually to receive or send some message, but the same principles apply anywhere. Consider the instruction :

```
Send move instruction
   : to Robot;
   : carry bottle;
   : from detector station;
   : to workbench.
```

Practically all the vocabulary in that instruction is peculiar to some machine in the system. We can't just build it into a language processor, as with a conventional programming language, because we might well want to use the processor with machines which haven't yet been invented. That gets us back to the database idea; all the vocabulary known by a machine must be in its database, so that it's clearly documented, and easy to change if necessary.

The result is a processor which works something like this. It knows ( please forgive anthropomorphism – we all know that I don't really mean it ) that it's working on a controller programme, so it looks for thing in the controller database. It first finds `Send`, which it finds in the controller's database identified as an instruction, with some indication of the expected syntax. Using this syntax, it identifies the sort of instruction it has to send ( `move` ), which might define some variant of the syntax. It also finds the destination of the instruction ( `Robot` ), and now it must inspect the robot's database to find what sort of instruction it can give, and the corresponding syntax, and how to encode the instruction in a message which the robot can understand. In working on this part of the message, it might have to refer to the databases for the detector station and workbench to find their coordinates, and to that of the bottle to find out how to pick it up.

All this sounds rather complicated, and in some ways it is, but in fact it doesn't involve anything new. All these data – geographic, encoding, etc. – must be used by human programmers constructing the same code, but they are usually not easily identifiable in the code production sequence. The most novel idea in PDL is to write everything down more or less formally in accessible form, and to use an automatic method to draw it together. The hope is that with this system it will be possible to change the plant by making changes to an easily recognisable set of databases, whereafter the automatic system will produce the appropriately modified code.

## DOES IT WORK ?

Yes. Well, so far it does. We're working on it.

Alan Creak,
April, 1997.