WORKING NOTE AC82

Alan Creak
10 January 1992

CLARIFICATION

This note was originally produced on 6-v-1991, not as one of my Working Notes. I'm dressing it up, almost unchanged, as a Working Note now to make sure I don't lose it.

This is a discussion on the language(s) we're talking about. I suspect some of it isn't as obvious as it should be in my report[1], though I think it's all there, if only by implication. I want to emphasise the differences between the stages at which we can describe the process to be executed.

PRINCIPLES.

In my report[1], I present a view of the development of a control programme from the time of the decision to produce the product concerned to the eventual production of a set of programmes for the production line machines which do the job. My proposal is structured as a stepwise development, where the specification is transformed in stages from an initial description of the product to a programme for controlling each machine's behaviour.

I considered four stages of transformation, connecting five states. The states are :

- A specification of the product.
- A specification of the process seen by the product.
- A specification of the process seen by the production line.
- A specification of the process and communications seen by the production line.
- A specification of the process seen by each machine.

To each of the states there corresponds a way of describing the task to be performed. I haven't been very specific in presenting details of these descriptive approaches, if anything implying that they exist as data structures lurking in the innards of some programme. In fact, though, each of them is, at least potentially, a language.

I think we've only really looked at two of the languages, and one of those only in passing : those corresponding to the process and communications seen by the production line (the language Al of my report[1] (page 48)), and the process seen by the machine (the Cl language (page 56)). I've confused the issue by not mentioning that two languages are involved - partly because I hadn't thoroughly realised it, as the difference was obscured by talking about the separate machines of the production line in the production line case, but about the LIS as a "machine" in the machine case, so that the example programmes for both cases described the same process. That's the bit I want to clarify.

It is important to bear in mind that the language must also function as the knowledge source for an expert system. The expert system needs to know about both what happens to the product in the production line (for things centred on the workpiece) and about what the machines do (for things centred on machines). We therefore need two (perhaps more) sorts of language : one to describe the process and communications from the production line's point of view (which is both a natural way to describe a process, and also important knowledge in its own right); and another to describe the process from machine's point of view (for reasons of practical necessity). The expert system needs access to both sorts of knowledge, so must understand both languages: it is obviously in our interests to make the two as similar as possible.

WHAT F&P DO.

They write all their programmes at the final level - what the machines do. All the preceding parts of the analysis are done by people in their heads.

That works well - but it is quite inappropriate as a basis for a flexible system that has knowledge about itself which it can use to address faults or cope with new developments. The knowledge must be accessible to the system to achieve any advance, and there are two ways in which this can be done.

- First, we can clamp the F&P programmers and engineers in appropriate constraints fitted with computer screens and electroencephalographic headsets. (Other facilities for

biological maintenance and housekeeping may be necessary; I shall not discuss those in this paper.) Problems will be described in natural language on the computer screens, and the subjects' responses monitored by electroencephalography. Techniques of natural language construction[2] and interpretation of electroencephalographic signals[3] are both under active development, though it may take some time to bring either to the level of sophistication which might be needed for this project. This is an exciting line of development which would bring the name of F&P into the worldwide vocabulary; a start could be made in a few years time for little more than $100,000,000.

- Alternatively, we might try to represent the knowledge of the programmers and engineers in terms comprehensible to fairly conventional computer programmes, so that it can be stored in the computer memory. This is pretty boring, limited in its publicity value to a few learned journals which nobody of any importance reads anyway, and is within the compass of Adrian Krzyzewski, who costs next to nothing.

DESCRIBING THE PROCESS AND COMMUNICATIONS SEEN BY THE PRODUCTION LINE.

Here is an example of a programme written in the Al-like language. It describes a process which uses a production line (fictitious, so far as we know) called a Gherkin Packer. It is a rather more detailed version of the programme which appears on page 49 of my report[1].

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

Machine :  LIS.
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 LIS.
Action :   Move the bottle to the Filler,
           Send "Operation complete" to the LIS.

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

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

Machine :  LIS.
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 LIS.
Action :   Move the bottle to the Packer,
           Send "Operation complete" to the LIS.

Machine :  LIS.
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 LIS.
Action :   Receive "Pack" from the LIS,
           Pack the bottle at the coordinates.
```

DESCRIBING THE PROCESS SEEN BY THE MACHINE.

From that programme, we want to derive programmes to drive the separate machines which constitute the production line. In this instance, the only interesting machine is the LIS. This requires a Gherkin Packer programme encoded more or less as below. (There's rather more here than there was in the report, because this time I've worked it out more carefully.)

```
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.
```

Programmes for the other machines will look broadly similar, but will be written using different vocabularies. For example, here's a programme for the filler :

```
Trigger
     Detect bottle
          : on workbench.

Procedure
     Fill bottle
          : with gherkins.
     Fill up bottle
          : with vinegar.
     Send signal
          : to controller;
          : using "finished" button.
```

We have to derive these programmes from the first one. How ?

Take the LIS programme as a reasonably ambitious example. First we select the LIS (or whatever) pieces from the programme :

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

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

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

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

Though these steps are separated in the initial Al programme, from the point of view of the LIS they form a single sequential programme, initiated by the appearance of the bottle at the detector station. We therefore want to combine them into a single sequence of instructions.

But we want to do this by some means which will satisfy our criteria of flexibility and expandability. That means, for one thing, that we can't build into the compiler any vocabulary peculiar to any individual machine. As both the programmes are almost entirely composed of instructions concerning peculiar properties of individual machines, this could pose some difficulties - but not many. We still have at our disposal the syntax of the languages, and it will be clear that this is conspicuously simple. It is therefore easy to analyse, and provided that the compiler can retrieve all other information it needs from descriptions of the devices used in the production line, all should be well. This is how it works.

The basic syntax of the two languages is perhaps apparent. (Note that I'm making this up as I go along, so it will perhaps work for this example but not much more. To do it properly will need a bit more work, but this should demonstrate feasibility.) The Al language can be defined in terms of frames :

```
<Al Programme>  ::=  { <action frame> } *
<action frame>  ::=  <machine line> <stimulus line> <action line>
<machine line>  ::=  Machine : <machine text>
<stimulus line> ::=  Stimulus : <stimulus text>
<action line>   ::=  Action : <action text>
```

The <machine text> must identify a machine described in the system's database; the syntax of the <stimulus text> and <action text> are defined in the machine's database entry.

The Cl language is equally simple :

```
<Cl programme>  ::=  <starter> <procedure part>
<starter>       ::=  <trigger>
                |    <whenever> -- See the report - not elaborated here.
<trigger>       ::=  Trigger <instruction>
<procedure part>     ::=  Procedure { <instruction> } *
```

As with Al, there are components - here `<instruction>` - which must be defined in the Machine Type Database. Now down to the example. To begin with, the compiler knows that it's dealing with a programme for the LIS, so it can retrieve the vocabulary proper to the LIS. This must include instructions for compiling (at least) `Receive`, `Send`, and `Calculate` instructions. So there must be a database which looks, in part, something like this :

```
LIS
     Al Syntax
          <stimulus text>       ::=  ?
          <action text>         ::=  ?
     Cl syntax
          Receive
          Send
          Calculate
```

The first item in the Al programme is a `Stimulus`, which we can reasonably compile into a `Trigger`. Next, though, we have to attend to the detail of the instruction, for which we need, first, the syntax for the `<stimulus text>` and `<action text>`. These are plausible entries :

```
<stimulus text>           ::=   <receive instruction>
<action text>             ::=   <any instruction>
<any instruction>         ::=   <receive instruction>
                          |     <send instruction>
                          |     <calculate instruction>
<receive instruction>           ::=  Receive <machine>.text from <machine>
<send instruction>        ::=   Send <machine>.text to <machine>
<calculate instruction>   ::=   ....
```

(The form `<machine>.text` is meant to denote a text string as defined by the `<machine>` mentioned in the statement.)

We can now proceed to analyse the first instruction, `Receive "Found a bottle" from the Detector`. Matching this against the `<stimulus text>`, which can only be a `<receive instruction>`, we identify `Detector` as a `<machine>` and `"Found a bottle"` as a text string proper to that machine. To determine what it means, therefore, we must look at the database entry for `Detector`, where we shall expect to find a description of a signal called `"Found a bottle"`. Here's a bit of the `Detector`'s entry in the database :

```
Detector
     Messages sent
          Found a bottle
               Description :   "Found a bottle"
```

We find the message as expected (if we didn't, we'd report an error), and we find that the message in fact consists of the string `"Found a bottle"`. In practice, it could equally well be of a quite different form, but I'll stick to this simple example for convenience. Now we can compile the Cl language equivalent. Once again, we refer to the LIS database entry to determine the required syntax. We find :

```
Receive <machine>.text.description : from <machine>
```

which by an obvious process becomes :

```
Receive "Found a bottle" : from Detector.
```

The next instruction is a little more complex :

```
Send "Move bottle from Detector to Filler" to Robot.
```

The first stages must be quite like those for the `Detector` message, but now we have a message which contains variable parts. Here's a bit of the `Robot` database entry :

```
Messages received
     Move <thing> from <A> to <B>.
          Description :
               <thing> : type object, attribute movable.
               <A> :type place, attribute reachable.
               <B> :type place, attribute reachable.
```

Matching identifies the components of the instruction, and the variable parts are further described as required. Types such as `object`, `place`, and so on should perhaps be regarded as world knowledge and built in to the basic language definition. Attributes, which can be used for checking semantics, may be associated with various things at various levels - so `movable` is here associated with the `object`, and is sufficiently general to be defined globally, while `reachable` is only applicable to a small class of machines, and should perhaps be locally defined. Notice too that to evaluate `reachable` we must know the identities of the specific robot, object, and places concerned; here we must use information from the Machine Database.

Associated with each type there must be instructions for handling variables of the type. So an `object` will be sought in the Component Database and, for specific details, the Active Products Database, while a `place` may require reference to any material component of the system, and must somehow be identified in the appropriate database. `Detector`, for example, must denote a `place` : so there is a component in the entry for `Detector` in the Machine Type Database which defines the meaning of the word used as a location. These will, obviously, normally be expressed relative to some defined base coordinate of the actual machine concerned.

Now having identified the parts of the instruction with information from the machines concerned, we can synthesise the next instruction in the Cl programme, again referring to the databases for specific information on how to construct the required text - or, as it may be, code at some later stage of the enterprise.

REMARKS.

I don't for a moment imagine that this will work as it stands - I remarked that I'm working it out as I go along - but I think the idea is sound. Neither do I think that the mechanism described is necessarily the only way, the best way, or even a particularly desirable way : it is, perhaps, the fundamental way. It is almost certainly an extremely slow way, and one obvious improvement once much more is known about the forms of the messages required at different levels is to build something much closer to a conventional compiler. This would still be quite a simple programme, as the basic structure of the language is simple, but would probably run at an acceptable speed.

But the important point is this :

> *It is possible in principle to devise a set of languages which deal with machines in a production line, and can be adapted to include new sorts of machine through changes to a database, and without any arduous reprogramming.*

REFERENCES.

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

2 : N. Ivanov : M.Sc. Thesis, Auckland University, 1990

3 : Z.Z. Keirn, J.I. Aunon : "Man-machine communications through brain-wave processing", *IEEE Eng. in Med. Biol.* **9#1**, 55 (March 1990).

SUMMARISED DATABASE FRAGMENTS.

```
Detector
      Al syntax
            Messages sent
                  Found a bottle
                        Description :      "Found a bottle"
            Positions
                  . :   coordinates : ....
                        name : "detector station"



LIS
      Al Syntax
            <stimulus text>          ::=   <receive instruction>
            <action text>            ::=   <any instruction>
            <any instruction>        ::=   <receive instruction>
                                     |     <send instruction>
                                     |     <calculate instruction>
            <receive instruction>    ::=   Receive <machine>.text from <machine>
            <send instruction>       ::=   Send <machine>.text to <machine>
            <calculate instruction>  ::=   ....
      Cl syntax
            Receive
                  syntax
                        Receive <machine>.text.description
                        : from <machine>
            Send
                  syntax
                        Send <machine>.messagetype
                        : to <machine>
                        : <machine>.messagetype.format
            Calculate



Robot
      Al syntax
            Messages sent
                  Operation complete.
                        Description :      ....
            Messages received
                  Move <thing> from <A> to <B>.
                        Description :
                              <thing> :   type object, attribute movable.
                              <A> :       type place, attribute reachable.
                              <B> :       type place, attribute reachable.
      Cl syntax
            Move
                  Messagetype
                        move instruction
                  Syntax
                        : carry <thing>
                        : from <A>.name
                        : to <B>.name
```