WORKING NOTE AC81

Alan Creak
10 January 1992

THEME AND VARIATIONS.

This note is about programmes for constructing products with optional components which can be chosen during production. The theme is that the overall process must still conform to the description I presented in my report[1]; the variations come in the timing of the operations rather than in their nature. Much of this note can be seen as an expansion of the section entitled BRANCHES found on pages 46 and 47 of the report, where most of the ideas on which this discussion is based are presented.

The note is also about *variability*, by which I mean changes in the nature of the product. If our aim is to produce several different types of the same basic product, then, if the numbers of units of the different types do not warrant the establishment of separate production lines, it may be convenient to set up a single production line which is capable of producing units of any of the required types. With such an organisation, it is clearly necessary explicitly to define the programme which the line is to follow for each unit produced, and it is here that our concern with variations begins.

The point I make in my report is that in order to manufacture a product certain information has to be brought together. The information required is of a number of types and comes from several different sources. There is a partial order in which the various items of information are required, but, so long as we are only concerned with the manufacture of a single type of product, precise timing is of little concern - provided, of course, that the control programme is complete by the time we wish to begin manufacture !

This simplicity is lost if we wish to manufacture different types of item at will using the same production line. Now the production line must behave in different ways according to the type of product being manufactured; in effect, according to the product type's own control programme. (Notice that I am here using *programme* to denote the set of instructions executed; it is not synonymous with "code file". In this sense, a code file may contain one programme, or several, selected by conditional instructions.) We must now worry about getting the right programme ready for each product as it moves through the production line, so preparing the programme is no longer an activity which we can think of as completed before the production line operates; instead, it becomes an integral part of the production process itself.

I therefore rephrase the first sentence of the note as "This note is about programmes for constructing programmes for constructing products ...". A supervisory programme must receive the initial request to make a whatnot, and then construct the specific programme required for making a whatnot and set it running. It is the high-level programme which is the focus of discussion.

To take a specific example, suppose we expand our gherkin business by taking over a sardine company, which packs sardines in tomato sauce in tins. Immediately after the takeover, we have two quite separate and rather rigid production lines which produce our two quite separate products. Later, though, we come to believe that we can benefit by combining the two operations, and that it is possible that products with all conceivable combinations of attributes might find a market niche. I am bound to say that the idea of tins of gherkins in tomato sauce does not have a strong appeal to me, but there are a lot of odd people about.

We therefore decide to construct a new production line to make { bottles, tins } of { gherkins, sardines } in { vinegar, tomato sauce }. (I invented this gharkine production line on page 11 of my report, but didn't expand it there.) How do we do it ?

VARIABILITY IS NOTHING NEW, BUT THIS SORT IS SPECIAL.

Variability in some sense is not uncommon; any system which exhibits different behaviour under different circumstances is showing a form of variability. (To reconcile this idea with my previous definition, it is only necessary to think of the "product" as the sequence of actions performed by the system - or, indeed, as the programme produced by the supervisory programme.) A robot which locates an object then adjusts its behaviour to pick up the object shows variability; a machine tool which continues until some condition is satisfied rather than merely repeating an operation for a preset number of times shows variability.

I think that the sort of variability under discussion here can be distinguished from these other types in a fairly precise way. The distinction is based on the idea of different programmes for the different product types, which I introduced in the previous section, and the essentially arbitrary nature of the differences between them. Once we have decided to manufacture a bottle of sardines in vinegar, then we know that there are certain sequences of actions which will be required, and others which won't. With the robot and machine tool mentioned as examples above, this is never so; while there may be occasions on which either uses less than the whole of its control programme, we can't say beforehand that any specific part can safely be omitted.

Further, if we now decide to manufacture a tin of sardines in vinegar, we must replace the bottling sequence by a tinning sequence; and there is no reason whatever to expect that these two sequences will be in any way related. This distinguishes the variability of group manufacturing systems from systems such as the original gherkin packer, in which the final component - the packer - executed a different action each time in order to pack the bottles into boxes. In that case, there was no arbitrariness in the sequence of different actions; it was always known which action to perform next, and the sequence could easily be generated by a simple algorithm.

We can therefore identify the species of variability of interest in this paper by describing the behaviour of the system in some such terms as these :

- The system is capable of several distinct modes of behaviour, each of which can be described by a programme.
- Different programmes are likely to have some parts in common and some parts different.
- Parts which differ between programmes do so arbitrarily; no relationship between them, either in form or in the sequence of the different programmes, is expected. (In special cases, there may be *accidental* relationships; it may be convenient always to construct a left-hand bookend immediately after a right-hand bookend. If you always want this to happen, though, you would combine the two programmes into one, and you may not wish to do so, as there is a significant demand for odd bookends. Not that I've ever been able to find one.)

I shall take that as my working definition. At the moment, I think it's right, though it could do with tidying up a bit.

One other characteristic of variability is noteworthy : the more remote we are from the machinery which does the work, the less the variability of the programmes we use. In this, variability resembles "hardness" (as in "hard real-time"). At some level in the system, a supervisory entity, computer or human, always executes the same programme; perhaps this accepts instructions from an operator and in response constructs lower-level programmes with more variability, which are then distributed to lower-level systems and executed there. In this note, I am trying to explore the various ways in which this programme synthesis can be managed by the high-level supervisory programme.

BUT FIRST - A MORE PRECISE DESCRIPTION.

In the report, and in the preceding discussion, I assumed that the aim of the exercise was to produce a programme that could be executed by something to make the object we wanted. That's still true, but my use of the word "programme" can be misconstrued. Had I been more careful, I would have used some term such as "set of instructions, and anything else needed, which when executed produces the right result" - or perhaps "black box which does the right thing when started". Now we begin to look at how it works, rather than just what happens, we need to be more choosy.

In F&P terms, the main reason for choosiness is that the signals emitted by the supervisory processor (that which executes the LIS) need not come from the actual executed programme at all : they may simply be copied from the database system. My "programme" therefore includes both the supervisory programme proper, and the relevant parts of the database. The distinction is general, though : Kusiak[2] distinguishes between the *variant* and the *generative* approaches to process planning. In the variant approach, which he regards as well suited to group technology manufacturing, manufacturing process instructions for different parts are preassembled and retrieved from a database as required; in the generative approach, the instructions are computed when they are needed.

To take account of this new and more detailed view, therefore, I shall distinguish between two parts of the programme. These are :

- The **active part**, which is the executable code and anything else which causes an action at the current level of discussion; and
- The **passive part**, typically a collection of data distinct from the code file, containing information essential for the proper behaviour of the active part, but not itself active.

The qualification regarding the current level is necessary. For a machine operating at the supervisory level, the data manipulated and sent to subsidiary machines may well be code for those machines, and will be regarded as the active parts in their operations.

WHEN TO MAKE CHOICES.

The gharkine manufactory turns out to be a good deal more complicated than I want for an initial analysis, so I'll just suppose that we have a process with a single choice, which makes either As or Bs. Obviously, we have to begin by knowing how to make both As and Bs; and if we have a production line available it has to have machinery with which both As and Bs can be made. What happens as we work through the process discussed in the report ?

We begin with a *product specification*, which in this instance describes two products, A and B. We can think of it as a united specification for making A-or-B. We end with a manufactured product, which must be unambiguously either an A or a B - so at some stage of the process we have to decide what we're actually doing. If the decision is delayed until the moment of manufacture, then we certainly know which of the two we want, so we can pick the correct action. To make this possible, though, at all earlier stages of the development, we must have prepared for either eventuality, either by continuing with a united specification for A-or-B, or by providing separately for both A and B. We can switch from the united to the separated track at any point in the development; the three cases follow.

- The first step is to convert the product specification into a *fabrication programme* - and, perhaps, a *production line design*. If we choose to separate the A and B developments at this stage, we shall generate separate fabrication programmes for the two cases, and perhaps separate production lines. If we do that, of course, there isn't much point in aiming for A-or-B in the first place; we would only begin from there if we were already convinced that both products could profitably be manufactured on the same machinery. If instead we defer the separation, there must be some instructions in the fabrication programme of the form "IF you are making A, THEN do something, ELSE do something else".

  There is an intermediate case : we could have started the A-or-B analysis because we wanted a combined production line, but were quite happy to have separate programmes. One can imagine circumstances in which this approach might be useful, but I shall follow it no further here.

- In the second step, we convert the fabrication programme into the *controller programme*, composed of specific instructions to the machines of the production line. Similar remarks apply : if we separate the development at this point, we must generate two controller programmes, while if we continue the joint development, we produce a single controller programme with conditional instructions.
- In the third step, we execute the controller programme to make the *product*. At this point, we must know what to do, so we do it. Either we begin by executing the correct controller programme of the two that are available, or we supply whatever information the single conditional controller programme needs to determine which path to follow.

The difference between the approaches is purely a matter of *binding time*, and similar decisions are common in computing. A very familiar example is the choice between compiling and interpreting : the initial programme must eventually lead to the same behaviour whichever course is taken, but there are differences in detail along the way. A general property seems to be that early binding can lead to greater efficiency, while late binding is conducive to greater flexibility. Kusiak's variant and generative methods[2] correspond to comparatively early and late binding. In a field not too distant from ours, it has been suggested[3] that progress in intelligent robot assembly research was greatly hampered for some time by attempts to implement early binding, which resulted in rigid systems which could not adapt to changes in the environment. Because of this limitation, all possible changes had to be foreseen and explicitly provided for in the programme, and this task proved too onerous to be practicable in real environments.

If the same is true in our systems, we shall avoid precompiled programmes unless we are quite sure that the early binding can do no harm. In this respect, we are in a favourable position; because of the particular species of variability in our systems, we know that our different programmes are assembled by tying together what amount to macros - significant sequences of instructions which relate to different aspects of the manufacturing process. These sequences are predetermined, so we can lose nothing by precompiling them, provided that we retain the ability to exercise late binding in connecting the fragments together.

The result, curiously enough, is that by taking account of the need for late binding we (almost) do away with it; for both the low-level fragments and the high-level programme which sticks them together can be fully specified and precompiled before production begins.

The current F&P system is just such a two-stage early binding process, I think. The two stages are needed because the system incorporates two levels of processor : a supervisory (LIS) level, and an executive (PLC) level. The operations possible at the PLC level can be imagined as a set of Meccano pieces from which you can assemble a complete assembly process. As with Meccano, there is a sense in which one could in principle devise a "new" machine by sticking together a random set of components. (In practice, the possible combinations are constrained by the geography of the factory; the Meccano of a traditional production line is highly constrained, that of a factory based on workcells with flexible materials transport between them is less so, and that of a job shop is comparatively free.) As well as sticking together the components you have, you can do new things by providing more Meccano - which, translated into terms of the control system, is extending the programming language by changing the passive part of its "compiler". Because of the way the system is designed, it's comparatively easy to make the change - but that doesn't affect the principle. It follows that everything's built in right at the beginning, with no real choice during manufacture except at a comparatively few decision points where the piece of precompiled code to be executed next is chosen; in other words, the system is based on early binding.

We can avoid late binding because we know just what is supposed to happen at each stage of the process. What do we do when it doesn't happen ? - which is to say, how can we build intelligent error recovery into the system ? If the system is to respond intelligently to faults, there must be some possibility of late binding somewhere. I shall not discuss this question further here, but it should be borne in mind that if a system is designed too rigidly it may not be easily extended to include intelligent recovery from unforeseen errors.

SEQUENCE OF OPERATIONS.

I discussed the development of the production control programmes in my earlier report[1]; here I comment on interactions between that development scheme and the development of programmes for systems with variability. (Page numbers in this section refer to the report.) It is still true that there is generally no rigid specification of the order in which things will happen. While it is reasonable to suppose that the sequence described there (page 29) will hold for any individual item of the process - we must know how to build an object before we can write instructions for a machine to do it, and the machine won't work until we've written the instructions - there is no reason why the development steps for different items need be synchronised in any way.

Fabrication programme : We now have two or more objects to make, but, as I pointed out earlier, we are ex hypothesi committed to making them with a single piece of equipment (which is in fact an assembly line, or something like it), so we want to write a single fabrication programme (this time more like a code file !) for the whole process. We must therefore be able to distinguish two parts to the programme, which I shall call the *gross* and *fine* levels.

The *gross level* is the instruction sequence which is common to all the manufacturing processes. The size of this component is some sort of measure of the degree to which the processes can share the same equipment, though it also depends on the degree of variability offered. For example, returning to the gharkine factory, there is no reason to suppose (apart from consumer surveys) that each of the possible features of a product should not appear in just about half of the manufactured goods, but because every component is subject to choice the gross level fabrication programme is something like this :

```
Get container
Put things in
Add stuff
Put lid on
Pack
```

The gross level part of the process is always needed, and will not vary; it can be converted into a controller programme in the usual way, whatever that is.

The *fine level* deals with the differences between the individual processes. It behaves like a set of subroutines called as appropriate by the instructions of the gross level programme - so `Get container` may call one of the two fine level routines `Get tin` and `Get bottle`.

Is there another level ? This description works well enough provided that the separate processes don't interact in any way, so that the machinery can switch instantly from one task to another. But suppose there are parts of the system which must be washed out if the "stuff" changes from vinegar to tomato sauce, or the conveyor belts need different jigs to move bottles or tins ? Then there are additional instructions at what I shall call an *interaction level* which are not properties of the separate running processes at all. So where do they come from ?

Well, they may not be properties of the "separate running processes", a form of words chosen with some care, but they follow from the processes' required initial conditions. This is a topic to which I didn't give sufficient attention in my report; though I mention initial conditions in a few places, I don't think I analysed the requirements in detail at all, and I haven't time to do it now. Generally, though, it is clear than any process may require that a certain state be established before it can run, and there should be provision for a starting-up phase to establish this state. The interaction level amounts to a check that the system is in the state required by the next process to run, followed by institution of appropriate action if not. The appropriate action may or may not be the initial setting-up procedure; either way, the information must be provided in the fabrication programme, and taken into account in successive stages of processing, and the system state must be available.

Line design : The principles for designing the production line are essentially unchanged from the version laid out in the report, with the added complication that there must now be provision for producing several different components in the same manufacturing facility. The *gross level* programme now acts as a grouping mechanism; it defines the order of operations, and shows where alternatives must be catered for. If the variability is high, so that there is no significant gross level programme, then no general order is specified, and the factory must be organised as a job shop.

In favourable cases, the variability may make no difference at all. If the differences are minor, so that all varieties can be manufactured by the same sequence of machines, then a simple production line will suffice, leaving all the variability to be handled by the programme.

It is interesting to remark that any gross level programme can in principle be reduced to this simple case, but possibly at a cost. If at some point in the manufacture we must use machine M_a for product type A or machine M_b for product type B, then we can always imagine them as arranged in series along a single production line and both operating on all products - but with M_a programmed to do nothing to products B, and vice versa. This may in fact be the easy way to do it, and certainly saves a branch in the production line; but the line becomes more rigid in that production schedules for the two product types become tightly linked. If machine M_a takes much longer to do its job than machine M_b , the series organisation makes it impossible to save time by using the two in parallel.

The *fine level* programme carries the information which determines whether or not different machines will be needed for the various product types. If all the operations required by all the corresponding fine level programmes can be handled by a single machine, then such a machine can be used. Alternatively, the separation into fine level programmes can be seen as a source of information which permits the choice of several simpler machines instead of a single complex, and possibly more expensive or less reliable, piece of equipment.

Controller programme : It now becomes necessary to include branches in the controller programme, but as before these need not necessarily be made explicit, as the type of product being made can sensibly be included as a part of the state of the virtual machine looking after the process. (Pages 46 - 47. Further reflection leaves me less dogmatic about IF; maybe it's all right so long as it's restricted to controlling computation (of values or of states), and can't get out into programme structure. I hint on page 47 that we'll need a conditional branch to deal with things we really can't predict.)

System operation : The controller programme determines how the system will operate, as before. In general terms, it is unlikely to look much different from the sort of programme discussed in the report; each component of the production line has its own part of the programme, and the additional complication consequent on the variability is evidenced more in the interactions between the components than in any overt feature of the code.

One possible difference may be visible if binding is left very late : the controller programme may be caused to retrieve variable components from some form of database (typically the Product Type database - or, for F&P, the BOM) as they are required.

DURING THE SYSTEM OPERATION.

Another way of looking at the question of variability is to concentrate on the mechanism whereby the computer system comes to emit the correct stream of instructions to cause the manufacture of the required product. This has to fit in with the stuff above somehow, but just at this moment I don't see how. Never mind - press on regardless.

We stick with the A-or-B product, and suppose that it is constructed in three steps, only one of which differs :

To make A :

```
do P;          ( Get container. )
do Q( A );     ( Fill with gherkins etc. )
do R.          ( Pack. )
```

To make B :

```
do P;          ( Get container. )
do Q( B );     ( Fill with sardines etc. )
do R.          ( Pack. )
```

Then the stream of instructions which must be produced somehow or other by the programme is :

```
I( do P ); I( do Q( which ) ); I( do R ).
```

where *I* is a function which miraculously converts the raw specification to a set of instructions in whatever encoded form is appropriate for the system. It is obvious that there are very many ways in which this desirable outcome can be managed, but I think the classification I propose below will cover the major ones. I'd like it to cover them all; any improvements are welcome.

First, we define a computer. A computer is a function $\mathcal{C}$ which produces output. The output we get depends on certain conditions :

$$\text{output} = \mathcal{C}(\text{ input, programme, environment }).$$

The three arguments of the function together cover all components of the computer system which are variable - that is, all available means of affecting its behaviour.

- We expect that the **input** will be something to the effect "Make an A" or "Make a B". Generally, it includes any information bound during execution of the process (in the computing sense) which controls the manufacturing operation.

- The **programme** is at our disposal; it is constrained by the nature of the processor on which it runs, but as we are concerned with a supervisory programme I shall assume that we have all normal computer facilities - which includes access to the environment. The programme is supposed to be bound early, before the process begins. Anything which affects its behaviour while running is deemed to be either input or environment material, even if it is presented as executable code. (If that needs an excuse, we may always suppose that the "real" programme contains all possible execution sequences, and the later information is merely used to select the sequence which will actually be executed.)

- The **environment** includes anything bound early, except the programme, which can affect the computer's performance; I shall use it only as a way of including the database material in the system.

> The definitions of these categories took a little while to develop. While the need for three different components was clear from the first, it was less obvious where to draw the boundaries. The "environment" category was particularly troublesome. I originally thought of it as including any resource on which the process could draw while in execution, which includes any database material, but also includes the operator. As I also had a mental picture of input as a file of instructions presented at the start of the process, this led to a consistent set of categories, but also to a confusion of real-time decisions by an operator with archival material from a database. This only sorted itself out when I realised that, once again, the binding time was a critical factor.
>
> I still wonder whether there should be some distinction between *initial input*, presented when the process begins, and *real-time input*, presented while the process is in progress. Practically, the difference is that the real-time input can be chosen after considering information on the progress of the manufacturing operation so far; this is interactive computing in comparison to the initial input's batch processing. Logically, the difference is again a matter of binding time : initial input is bound late, while real-time input is bound very late indeed. In this discussion, I have kept to the single input category; perhaps the usefulness of splitting the two subcategories will become clear as we proceed.

Consider the behaviour of the system when presented with the input "Make an A", expressed in some acceptable notation. We know that the output must be the string "`I( do P ); I( do Q( A ) ); I( do R ).`"; how can this output be derived from the input ? We consider the possible combinations of sources of the components of the programme.

There are at least three components. One is the gross level programme containing the basic instruction sequence "P, Q, R"; the second is the fine level programme which gives the details of the two variants of Q; and the third is the operation of selecting one of the Q variants. If we assume that each of the three can be associated with any of the arguments of $\mathcal{C}$ - input, programme, or environment - that gives us 27 possible combinations, each a potential source of the required behaviour. Not all of these combinations are particularly sensible, but they are all possible; so long as the system has all the information it needs when it is needed, it is of no importance whence it comes.

I shall ignore perfectly possible, but hybrid, combinations, such as schemes in which the first half of each instruction sequence comes from the input and the rest from the environment, or where the instructions for building A are explicitly included in the programme, while instructions for building B must be read from the input. Some such combinations seem a bit silly, but others are plausible enough. (A is the standard model, but B is a variant chosen by the customer for a large additional fee.) I'll stick to the simple case; I'm looking for basic principles, not a magic lamp.

There is one (perhaps only one ?) element of consistency which pertains equally to all the possibilities I discuss : that whatever happens must be determined, at some level, by the programme which controls the process. That doesn't mean that the programme as loaded into the machine contains explicit instructions to cover every detail of the operation; but it does mean that if anything remains unspecified in the programme then the programme must contain some instructions which say how the thing in question is to be specified. Each of the bits and pieces below therefore says something about what must be in the programme, and I have tried to make this as explicit as possible in each case.

Here goes. First, some remarks on all the individual variants. In each case I concentrate on how the variant will be encoded in the controller programme.

Source of the instruction sequence.

Gross level programme from input : the instructions are presented to the programme from "outside". The source could be another programme, or, if real-time input is acceptable, an operator taking the system through its paces step by step, as with a teleoperated manipulator. This would also be plausible for a job shop.

The programme initially loaded into the computer includes an instruction to the effect :

```
Read a description of the complete process from the
     input.
```

The instruction need not look quite like that. If we choose real-time input with very-late-indeed input binding, it will be more like :

```
Do forever
     Read a description of the next process step from
          the input;
     Execute the next process step.
```

In either case, though, the programme contains no explicit details of the process to be executed.

This idea of interactive execution sounds attractive in some ways, but if it is to fit into our overall scheme the freedom permitted must be very greatly restricted. This is because the whole system as described here and in my report is based on the principle that we know what we want to do right from the start - ideally, before we even start to build the production line. This precise knowledge is used throughout the development of the system. How can it be reconciled with a scheme for guiding the process interactively ?

Well, it can't, unless the gross level programme has been reduced to the trivial form :

```
Do forever
     Anything.
```

The only alternative is that the gross programme determines at each stage a set of operations from which one can be selected; but in that case, the selection is not a matter of providing the gross level programme, but of selecting a variant.

We conclude that real-time input can, in this case at least, be ignored.

But we can raise similar questions about initial input too. If we must know what the gross level programme is in order to define the production line at all, how can we make sense of supplying it again at the time of execution ? The quick answer is that we can't, except possibly as a purely trivial operation which could just as well be done from the environment.

We conclude further that the whole idea of reading the gross level programme from the input is silly. I shall ignore it henceforth.

Gross level programme from programme : the sequence "`do P; do Q( which ); do R`" itself is built into the programme as the standard response to a request to make A or B. The code for P and R is also included, as it doesn't change with the type of product made.

Gross level programme from environment : on receipt of a request to make A or B, the programme retrieves the sequence of operations from a database.

This variant is in some ways very like the "Gross level programme from input" case; in both cases, the programme has to fetch information from some external source, and it is in the best traditions of device independence that it should not be necessary to know what that source is. Therefore, the programme includes an instruction to the effect :

```
Read a description of the complete process from the
     environment.
```

Comparison of this descriptions with the corresponding version for the input case makes two points. First, they are indeed, superficially at least, very similar, which contributed to my confusion when trying to sort out categories. Second, though, the environment is by definition bound early, so in this case there is certainly no point in an equivalent of the second form of instruction suggested for the input case; that is only useful even in principle if there is some reason to defer the decision on the course of events until the process has already started, and I have already shown that it is useless in practice.

The similarity to the input case does not extend far enough to validate the argument presented there for the complete abolition of the case; the critical difference is the early binding of the environment. It is just as sensible to save the details of the programme in the environment as in the code file itself, and it would certainly be sensible to use the environment if we split the development process before making the controller programme, and therefore have two independent programmes, one for each product type.

Indeed, this is such an attractive arrangement that we may sensibly ask if any other case need be considered. Given that we begin with a request to make a product of a specific type, why can't we simply retrieve the whole programme we want, and run it ? I have no good answer to this question; but I can't demonstrate that the alternative is necessarily silly, so I'll go on thinking about it.

Fine level programme from input : this is rather less unreasonable than the suggestion that the gross level programme should be read from the input. It means we're making something in which one characteristic cannot be predefined. "Knit a pullover using this pattern : ...." ? "Make a pizza with these ingredients : ...." ? Provided that the form of the fine level programme can be constrained sufficiently to ensure that it will run in a machine, or sequence of machines, which can be precisely defined to the development system, all should be well.

Once again, though, all may be well, but all is not thereby guaranteed to be sensible. Who is going to check that the fine level programme will run ? We have no provision for this sort of analysis in our controller programme. We must at least provide a programme to parse and check the input - and this then becomes our fine level programme. To extend the two examples above, we would have fine level programmes called "Knit a pullover" or "Make a pizza", and, provided that the resources they needed could be clearly defined, they could interact with the operator as much as they wanted.

I think that's unavoidable. The development system is not designed to be recursive, which it would have to be if we want to add what amounts to new product specifications during production, and I don't think it would be straightforward to make it recursive, as there are too many constraints on what can be done once production starts. I'm sure it would be complicated.

We therefore conclude that the whole idea of reading the fine level programme from the input is silly. I shall ignore it henceforth.

Fine level programme from programme : the two variants of the "Do Q" instruction are coded into the programme. The programme includes an instruction to the effect :

```
case which of
A :  do Q( A );
B :  do Q( B ).
```

The code for Q(A) and Q(B) is, of course, also included.

Fine level programme from environment : the two variants of the Q procedure are kept in a database, and fetched by the programme as required.

The programme includes an instruction to the effect :

```
Read from the environment the details of Q( which ).
```

Notice that we don't need a case statement here, as only one version of Q is available for execution. That's not to say that we won't need a fair bit of other machinery to tie the programme together, but this can be anything from a linking loader if we want to end up with a single programme in memory to nothing at all if the Q procedure is simply a set of instructions to be sent to another machine.

variant selected by input : the input explicitly specifies Q(A) or Q(B) somehow. This is the normal case; if there is no arbitrariness in the production requirements, then we have a fully defined system which we can treat as though there was no variability. It may be, of course, that the techniques considered here could be useful in such cases - in the alternate manufacture of left-hand and right-hand bookends, there may well be much common ground. In such circumstances, the next category is more appropriate.

In this case, nothing special appears in the programme. We assume that the initiating instruction, "Make an A", passes the parameter A to the programme as the value of an argument `which`.

variant selected by programme : the programme determines for itself which variant to make. The bookends example is a trivial, and not very interesting, example. Such a procedure is also appropriate if there are characteristics of the process which are variable from product to product, but which don't depend directly on the information which defines the product concerned.

I think that my "`do P; do Q( which ); do R`" example is a little too simple to illustrate this case adequately, because it contains only one variable. Suppose, though, that there were two parameters for a product - say, { gherkins vs. sardines } and { bottles vs. tins }. Suppose further that because of the peculiar chemical composition of sardines they cannot be packed in bottles with the usual plastic lids. Something somewhere has to say :

```
if bottles
then if sardines
     then metal lids
     else plastic lids.
```

Who should do it ? We could require that the lid type be specified as a third input parameter; but then we would have to include a safety check somewhere to ensure that sardines didn't get plastic lids by accident - so we may as well work out the lid type in the programme, and stick to the two original input parameters. The programme still knows about the "lid type" attribute, but the variant is decided by the programme itself.

I think that this is where the *interaction level* (see page 5 of this note) comes in, but here the need for special treatment comes not from peculiar characteristics of individual products, but from special requirements attached to the sequence of products. The code for the change from vinegar to tomato sauce would be something like this :

```
if vinegar then
     if last was tomato sauce
     then wash
     else
else

if tomato sauce then
     if last was vinegar
     then wash
     else
else
```

Except for the reference to the previous product, this is clearly analogous to the plastic lids example.

variant selected by environment : the database somehow knows which is the correct instruction to return. This is rather like the previous case, but avoids building manufacturing details into the programme. As before, we are concerned with interactions between multiple specifications, but now we use an instruction like

```
Read from the environment whichlid given whichcontainer
     and whichfood.
```

Now, this gets a bit tricky. On the face of it, there are at least two ways to make it work. One I've just described, but it requires an odd little database which doesn't fit in nicely with the rest of the system. The other goes more like this :

```
Read from the environment CONTROLLER code for
     whichcontainer;
Execute the controller code.
```

The controller code for the tins does nothing in particular; but the controller code for the bottles contains the instructions to select the lid type. This is quite neat; but it goes against my principle that there's no real controller code anywhere but in the programme, so I think I want to reduce it to the previous case.

I'm not quite as sure about this as I'd like to be. It's brought to light one flaw in my argument : I haven't distinguished as clearly as I should have between code executed by the controller and code sent to the machines doing the fabrication. For the moment (because I haven't time to do it all again) I'll stick to the odd little database, but I should come back to this argument and tidy up the loose ends.

THE COMBINATIONS ENUMERATED.

That's all about the individual possibilities. Now the 12 combinations. In each case, I give a label for reference, a summary of the combination, and a sketch of a controller programme which fits the combination, and a comment. The process proper is always initiated by an input instruction of the form "Make an A", which sets the initial value of the variable `which`.

Bear in mind that the programmes given are only sketches, intended to illustrate, not to prescribe, what might be found in the actual programmes. This is most obvious in cases where a variant is not completely prescribed by the input, so is calculated in the programme or retrieved from a database in the environment. I remarked on this case above; it is sometimes marked in the examples by the appearance of a variable `whichlid` which is never used.

PPI

Gross level programme from programme;
Fine level programme from programme;
variant selected by input.

In the programme :

```
do P;
case which of
A : do Q( A );
B : do Q( B ).
do R.

procedure P : ..... ;
procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
procedure R : ..... ;
```

In the environment :

nothing in particular.

Very plausible. You choose which you want; the programme knows how to do it.

PPP

Gross level programme from programme;
Fine level programme from programme;
variant selected by programme.

In the programme :

```
do P;
if bottles
then if sardines
     then whichlid = metal lids
     else whichlid = plastic lids.
case which of
A : do Q( A );
B : do Q( B ).
do R.

procedure P : ..... ;
procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
procedure R : ..... ;
```

In the environment :

nothing in particular.

An odd one I didn't expect, but it makes sense. If you're using a variable system, there has to be some variation - you can't build everything into the programme. (Or, generally, you can't bind everything early, so there'll be some others in this category.)

PPE

Gross level programme from programme;
Fine level programme from programme;
variant selected by environment.

In the programme :

```
do P;
Read from the environment whichlid given whichcontainer and
     whichfood.
case which of
A : do Q( A );
B : do Q( B ).
do R.

procedure P : ..... ;
procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
procedure R : ..... ;
```

In the environment :

a database of the values of `whichlid` required to select the correct lid for all combinations of values of `whichcontainer` and `whichfood`.

Like PPP.

PEI

Gross level programme from programme;
Fine level programme from environment;
variant selected by input.

In the programme :

```
do P;
Read from the environment the details of Q( which ).
do Q;
do R.

procedure P : ..... ;
procedure R : ..... ;
```

In the environment :

a database containing

```
procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
```

Another good one, like PPI.

PEP

Gross level programme from programme;
Fine level programme from environment;
variant selected by programme.

In the programme :

```
do P;
if bottles
then if sardines
     then whichlid = metal lids
     else whichlid = plastic lids.
Read from the environment the details of Q( which ).
do Q;
do R.

procedure P : ..... ;
procedure R : ..... ;
```

In the environment :

a database containing

```
procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
```

Like PPP.

PEE

Gross level programme from programme;
Fine level programme from environment;
variant selected by environment.

In the programme :

```
do P;
Read from the environment whichlid given whichcontainer and
     whichfood.
Read from the environment the details of Q( which ).
do Q;
do R.

procedure P : ..... ;
procedure R : ..... ;
```

In the environment :

a database of the values of `whichlid` required to select the correct lid for all combinations of values of `whichcontainer` and `whichfood`.

a database containing

```
procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
```

Like PPP.

EPI

Gross level programme from environment;
Fine level programme from programme;
variant selected by input.

In the programme :

```
Read a description of the complete process from the
     environment.

procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
```

In the environment :

```
do P;
case which of
A : do Q( A );
B : do Q( B ).
do R.

procedure P : ..... ;
procedure R : ..... ;
```

Like PPI, but the knowledge bound early is distributed between the programme and the environment. Notice that, once we start reading the gross level programme from the environment, that's where some of the things labelled "from programme" must be put. More precisely, actual components of the executable code can only sensibly be put in their proper places; but I've left logically separate procedures in the programme.

EPP

Gross level programme from environment;
Fine level programme from programme;
variant selected by programme.

In the programme :

```
Read a description of the complete process from the
     environment.

procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
```

In the environment :

```
do P;
if bottles
then if sardines
     then whichlid = metal lids
     else whichlid = plastic lids.
case which of
A : do Q( A );
B : do Q( B ).
do R.

procedure P : ..... ;
procedure R : ..... ;
```

Like PPP.

EPE

Gross level programme from environment;
Fine level programme from programme;
variant selected by environment.

In the programme :

```
Read a description of the complete process from the
     environment.

procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
```

In the environment :

```
do P;
Read from the environment whichlid given whichcontainer and
     whichfood.
case which of
A : do Q( A );
B : do Q( B ).
do R.

procedure P : ..... ;
procedure R : ..... ;
```

a database of the values of `whichlid` required to select the correct lid for all combinations of values of `whichcontainer` and `whichfood`.

Like PPP.

EEI

Gross level programme from environment;
Fine level programme from environment;
variant selected by input.

In the programme :

```
Read a description of the complete process from the
     environment.
```

In the environment :

```
do P;
Read from the environment the details of Q( which ).
do Q;
do R.

procedure P : ..... ;
procedure R : ..... ;
```

a database containing

```
procedure Q( A ) : ..... ;
procedure Q( B ) : ..... ;
```

Like PPI, but with all the knowledge in the environment.

EEP

Gross level programme from environment;
Fine level programme from environment;
variant selected by programme.

In the programme :

```
Read a description of the complete process from the
     environment.
```

In the environment :

```
do P;
if bottles
then if sardines
     then whichlid = metal lids
     else whichlid = plastic lids.
Read from the environment the details of Q( which ).
do Q;
do R.

procedure P : ..... ;
procedure R : ..... ;
```

a database containing

```
     procedure Q( A ) : ..... ;
     procedure Q( B ) : ..... ;
```

Like PPP.

EEE

Gross level programme from environment;
Fine level programme from environment;
variant selected by environment.

In the programme :

```
Read a description of the complete process from the
     environment.
```

In the environment :

```
do P;
Read from the environment whichlid given whichcontainer and
     whichfood.
Read from the environment the details of Q( which ).
do Q;
do R.

procedure P : ..... ;
procedure R : ..... ;
```

a database of the values of `whichlid` required to select the correct lid for all combinations of values of `whichcontainer` and `whichfood`.

a database containing

```
     procedure Q( A ) : ..... ;
     procedure Q( B ) : ..... ;
```

Like PPP.

CONCLUSIONS.

Reading the process description from the input is pretty implausible - not intrinsically, but in combination with the other bits. It's implausible because if you were giving the instructions, you'd give the right instructions, and there wouldn't be any question of variation in the programme. I can make a sort of sense of some of the possibilities, but by and large it's clear that you only need worry about variation if you bind the programme early - which is sensible enough.

For variants to make sense, there has to be some late binding and some early binding. Not surprisingly, the most plausible candidate for late binding is the choice of what to make; how to handle the variant part sometimes makes sense, provided that this is one possibility out of several.

I wondered earlier whether there was a significant difference between late binding and very-late-indeed binding. This survey suggests that the difference may not be very important, except that there is never a case for making arbitrary decisions which affect the gross programme during execution. It certainly didn't obtrude itself as I worked through the different possibilities.

REFERENCES.

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

2 : A. Kusiak : "Process planning : a knowledge-based and optimization perspective", *IEEE Trans. Robotics Automation* **7**, 257 (1991).

3 : T. Smithers : I think, in an Edinburgh report somewhere, but I can't find it.