The Pipelining Transformation on Plans for Manufacturing Cells with Robots

Erik Sandewall
Department of Computer and Information Science
Linkoping University
S-58183 Linkoping, Sweden
E-mail: ejs@liuida.uucp

Abstract: The plan or program for a manufacturing cell may be seen either from the workpiece (describing the successive operations that it wants to undergo during its track through the cell) or from the perspective of the workcell (describing its quasi-parallel operation on several workpieces which are processed simultaneously in the cell}. This paper addresses the question of how to verify tnat a proposed cell program implements a given workpiece program. The verification method is described informally using an example, and then formulated in precise and formal terms.

0. The Planning Problem in Factory Automation.

One of the important research problems in factory automation is improved methods for programming production cells. The flexible manufacturing environment has high demands on easy development and update of programs, and requires convenient, application- level dialogue between the engineer or operator who is in charge of the production cell, and the controlling computer system. It is not practically possible to satisfy these requirements only using conventional software engineering techniques.

Good software systems for programming production cells should make it possible to describe the desired behavior of the cell in terms of the objects which are present there (e.g. NC machines, robots, and conveyors), and in terms of the natural, elementary actions that those objects can perform (e.g. elementary move operations performed by an industrial robot). The techniques of knowledge based programming, often known as expert systems techniques, are well suited for realizing such a problem- oriented programming style.

If artificial intelligence techniques are therefore likely to be useful for factory automation in this particular respect, it is equally true that the factory automation domain offers important and interesting problems for artificial intelligence. In particular, the production cell is a 'world' of non-trivial but limited complexity, where current techniques for planning and problem-solving can be further developed and refined. The present paper •hows how a method for planning and for reasoning about actions, suitably extended, can be used for a practically significant planning problem in production cells.

This research was supported by the Swedish Board of Technical Development.

1. Pipelining of Manufacturing Plans.

Consider a simplified production cell for automatic manufacture, as illustrated in figure 1. Successive workpieces arrive on the incoming conveyor (fI, and are processed by the three successive machines (X,Y, and Z). A handling robot (R) moves the workpieces from I to X (movement operation A), from X to Y (movement operation B), from Y to Z (movement operation C), and from Z to outgoing conveyor, 0 (movement operation D). Each of the machines may be for example an NC machine tool, a machine for finishing surfaces on the workpiece, or a machine for checking tolerances.

We will use X also as a name for the operation performed by machine X, and similarly for Y and Z. We can usually assume that each of the operations X,Y,Z,A,B,C,D has a constant duration in time, and that the X,Y, and Z operations take significantly longer time than the movement operations A-D. Successive work cycles can be assumed to use the same "program" or "plan", which implies that the whole production cell has a well defined cycle time. One needs to determine the time-optimal plan, both in order to maximize the throughput of the production cell, and in order to balance the total production line which consists of several cells.

The program or plan for the production cell may be seen from two different perspectives. For a particular workpiece, one will have the plan shown in figure 2. It says that the workpiece at hand first undergoes the A (movement) operation, then the X (processing) operation, etc.

From the perspective of the production cell, we will instead have the plan shown in figure 3, where full-line arrows indicate steps in one work-cycle, and broken-line arrows indicate steps in the preceeding and succeeding work-cycles. The best way to understand the plan is by analogy with a 15-puzzle: when all machines contain a workpiece each, the only empty slot to which a workpiece may be moved is on the outgoing conveyor belt. The 'first* operation must therefore be for R to move a workpiece from Z to O (operation D). After that, the workpiece that has finished operation Y may be moved from Y to Z (operation C). After that again, the new workpiece in Z may be processed, and at the same time the empty slot in machine Y may be filled using operation B, and so the analysis continues.

Both these perspectives are legitimate and useful; we shall call them the *workpieee program* and the *cell program,* respectively. The cell program is significant because it is the program that the control software for the production cell primarily executes; it is the program on which the cell's cycle time must be calculated; and it describes the behavior of the cell as seen by the operator in charge.

The workpieee program, on the other hand, represents an underlying and more fundamental plan. Along the path of successive transformations, from the information in the CAD system to the automatic manufacture, one would expect to go first from CAD system information to the workpieee program, and then by another transformation to the cell program. In particular, if the configuration of the production cell is changed (for example by having a robot with two hands instead of one), then the cell program may change but the workpieee program remains constant.

In the long run, we would therefore like to have methods which can determine an optimal cell program for a given workpieee program and a given cell configuration. (Practical programs have a lot more complexity than this simple example shows). As a first step, we look for methods for verifying that a proposed cell program correctly implements a given workpieee program in a given cell configuration. The present paper addresses that problem.

In our example, the workpieee program is entirely sequential, whereas the cell program involves parallelism. In more general cases, the workpieee program may also contain parallellism, e.g. in an assembly task where several sub-assemblies may be put together independently of each other, and then be finally mounted together. The transformation from workpieee program to cell program may sometimes *impose constraints* on the parallel execution of operations, namely if there is competition for shared resources such as (typically) the handling robot or the workpieee. On the other hand, the transformation may also introduce *additional possibility* for parallel execution of operations if the cell program handles several workpieces concurrently, in successive stages of completion. That latter aspect is similar to the concept of *pipelining* in computer architectures, and we shall use the term pipelining in the case of production cells as well.

2. The solution through an example.

We first show through the example how the workpieee program and the cell program may be related, in order to convey the general idea in an informal way. In the following sections we proceed to the formal treatment.

Consider the proposed cell program for the example, as represented graphically in figure 3. Assume that the workpieces that flow through the production cell are assigned serial numbers, so that the first object to be manufactured obtains serial number 1, etc. Within one cycle of the program, we label each operation which involves a workpieee, with the serial number of the workpieee that is processed there during the cycle in question.

In figure 4, we see the results of the labeling, together with the continuation of the same labeling. In the next cycle, all operations have their serial numbers increased by 1, and so on in later cycles. We think of the *cell program* as a sequence of very many repeated occurrences of the same cycle (rather than as a formula saying * repeat the following cycle N times"). However, in spite of the arbitrary length of the cell program, those operations which are labeled with a given serial number will clearly be limited to five consecutive cycles of the cell program, corresponding to the number of cycles during which a workpieee stays within the production cell.

Now select from the cell program in figure 4, the substructure of those operations labeled with a certain serial number, as shown in figure 5. The resulting sub-structure represents the operations performed on a certain workpieee, and the relative ordering on those operations in time. We will call it the workpieee program that has been *extracted* from the cell program (modulo the initial serial number labeling). We can see how figure 2 re-appears in figure 5, i.e. the original workpieee program is re-obtained by extraction from the cell program.

The basic criterium on the relationship between the workpieee program and the derived cell program can now be expressed: For the proposed cell program, there must exist some serial-number labeling such that the resulting, extracted workpieee program is equal to, or a temporal strengthening of the given workpieee program. We say that P' is a temporal strengthening of P if P and P' contain the same operations, and every temporal ordering in P occurs also in P\ but not necessarily the other way.

In the example, P and P' are identical. In other examples where the given workpieee program contains parallel operations, it may happen that the cell program introduces additional temporal constraints due to shared resources, whereby also P' contains additional temporal orderings besides those found in P.

Besides this necessary relationship between the workpieee program and the cell program, there are also some other constraints on those programs, namely the familiar precondition/ postcondition constraints and the constraints of shared resources. We will get back to those issues in the next section.

3. Formal preliminaries.

The previous section gave an intuitive notion of the method through an example, but it remains to cast it in precise and general form. The formal methods for reasoning about time and actions are legio. We will use the method previously described in (ref. 1), and we also refer to that paper for an overview of alternative approaches.

In the approach we will be using, the world in which the plans are executed is characterized by a number of *features,* each of which may have either of a number of values (often a choice of two, Boolean). A *partial state* assigns either of those values, or undefined, to each of the features. The possible values of a feature are allowed to be chosen from a flat lattice, with undefined as the

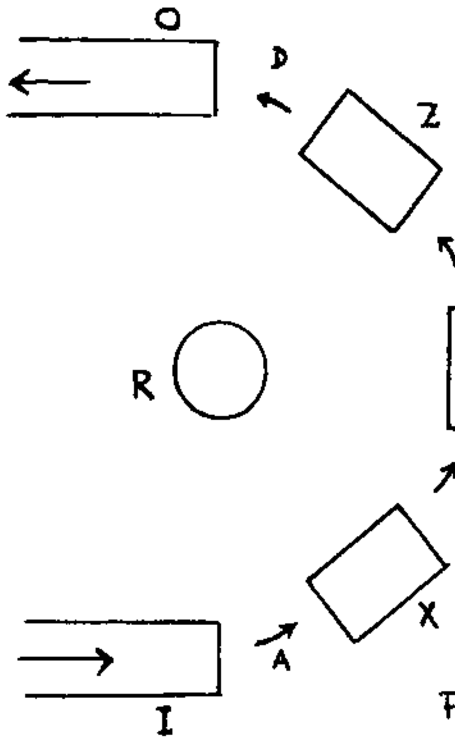

Fig. 1.

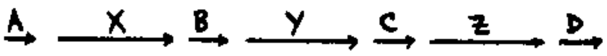

Fig. 2.

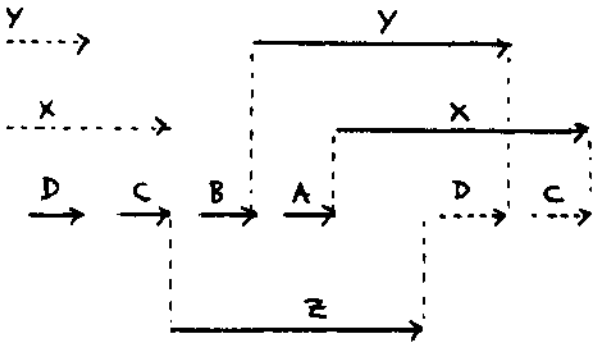

Fig. 3.

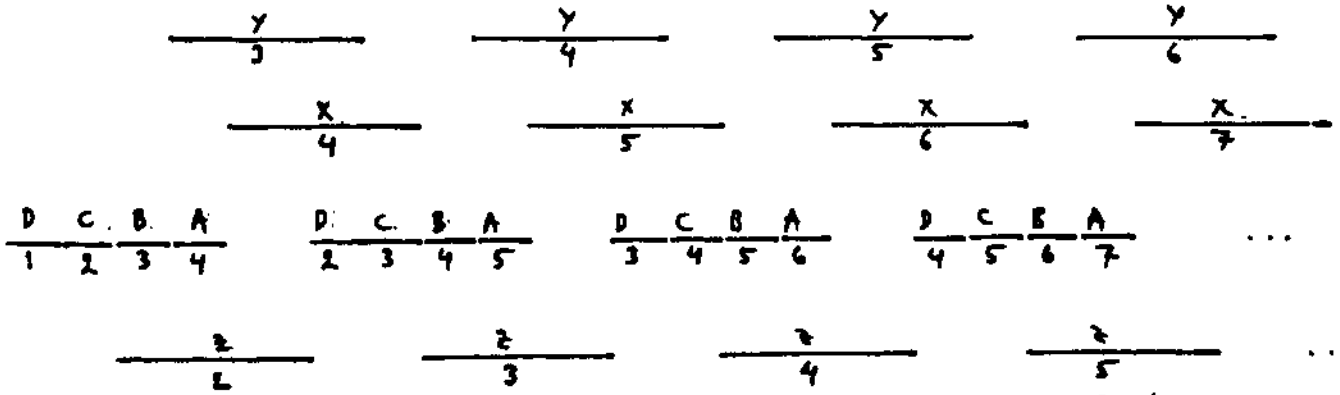

Fig. 4.

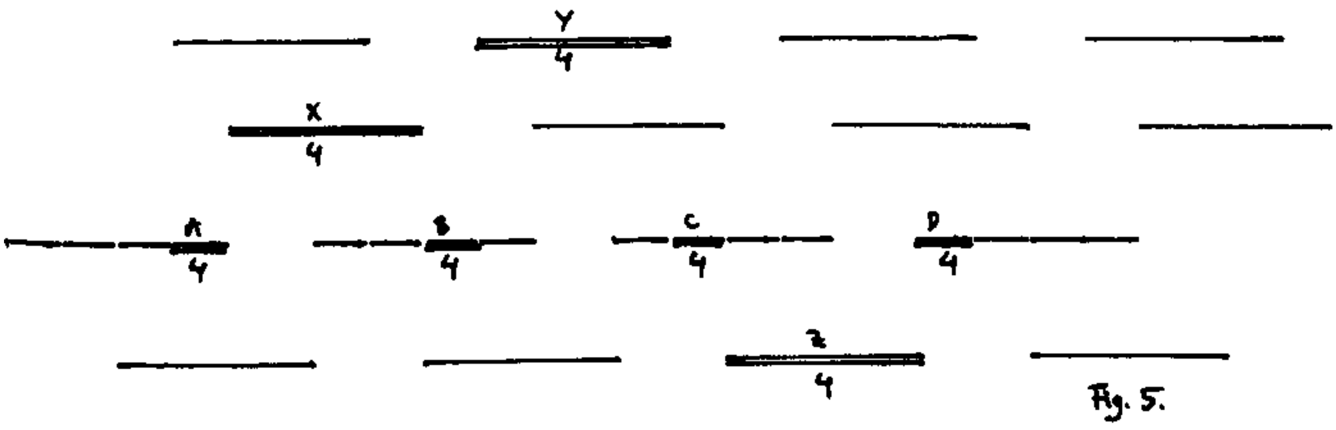

Fig. 5.

bottom element, the 'defined' feature values as the middle level elements in the lattice, and with an additional 'top' element in order to make the lattice complete.

Let Fi be the domain of the i:th feature, represented as such a flat lattice. Each partial state s is then a member of the Cartesian product S = F1 x F2 x ... x Fn of the n feature domains. The domain S is also considered as a lattice in the obvious way.

The element of Fi which is used for forming s, will be called the *projection* of s into the *dimension* i, and will be written s[i]. The element s will be said to *have the i:th feature* iff s[i] is different from undefined. We write

dim(s)

for the set of all i such that s has the i:th feature.

Two states s and s'are said to be *co-dimensional* iff

dim(s) = dim(s')

and they are said to be *anti-dimensional* iff dim(s) and dim(s') are disjoint sets. (We henceforth drop the prefix 'partial' and say just 'state' covering also partial states).

The state s is said to be *consistent* iff none of its projections is the top element in its domain lattice.

An *action* is a fourtuple [f,b,v,e] where f,b, and e are states, and v is an *operation*, i.e. a member of a domain V whose members are called operations, and for which no inner structure is assumed. From the domain of actions, we distinguish a subset A of *valid actions*. Intuitively, members of A are those fourtuples where f characterizes those features in the state of the world which are constant for the duration of the action, b characterizes the state of the world immediately before the operation v takes place, and e characterizes the state of the world at its conclusions. The states f, b, and e are called *prevail conditions*, *preconditions*, and *postconditions*, respectively.

We always require from valid actions [f,b,v,e] that b and e must be co-dimensional with each other, and anti-dimensional with f.

The operation Noop leaves every state unchanged, i.e.

[s, $\perp$, Noop, $\perp$]

is a member of A for every consistent s in S, where $\perp$ is the bottom element in the lattice S (all features undefined).

We assume a set T of *time-points*, and a partial order $\leq$ defined on T, representing the order of temporal precedence.

An *action structure* over a set A of valid actions is a triple

[T, $\leq$, p]

where p (for plan, or program) is a set of triples

[t,a,t']

where again t and t' are time-points in T, a $\in$ A, and t $\leq$ t' for every triple in the set p. Each member triple

[t,a,t']

will be called an *action occurrence.*

When analyzing the appropriateness of such plans, we must be concerned with precondition/postcondition relationships: earlier actions in a plan may have effects which are required in the preconditions or prevail conditions of later actions. The temporal ordering $\leq$ must be such that all preconditions of the actions in the plan are guaranteed to be satisfied.

Secondly, we must be concerned with the problem of shared resources, for actions which may occur in parallel. In our terms, if a plan contains two actions,

[t1, [f1,b1,v1,e1], t1']

and

[t2, [f2,b2,v2,e2], t2']

where dim(b1) and dim(b2) are non-disjoint, then the partial order $\leq$ must satisfy

t1' $\leq$ t2 or t2' $\leq$ t1

In (ref. 1) we showed a set of conditions, expressed in these formal terms, which guarantee that the precondition/ postcondition requirements and the shared resource requirements are satisfied. The basic requirement is as follows.

Let [T,$\leq$,p] be an action structure. For each member t of T, we define the *incoming action occurrences* to be those members of p having the form

[t',a,t]

i.e. having the given t as their last element. The *incoming states* for t are defined as follows:
- the post-condition (e) of each incoming action occurrence, is an incoming state
- the join of the prevail-conditions (f) of all the incoming action occurrences, is an incoming state.

The *outgoing action occurrences* and *outgoing states* are defined similarly. The requirement on [T,$\leq$,p] is now that for each t in T:

1. The incoming states are consistent and anti-dimensional,

2. The outgoing states are consistent and anti-dimensional

3. The join of the incoming states equals the join of the outgoing states, if the time-point has both incoming and outgoing states (balancing condition).

or, that one can add to p some number of action occurrences of the form

[t, [s, $\perp$, Noop, $\perp$], t']

so that the resulting action structure satisfies the above three requirements.

Example: suppose we have a time-point with two incoming action occurrences, a1 and a2, and two outgoing action occurrences, a3 and a4. These are defined so that a1 fills vessels c1, c2 and c3, and a2 fills vessels c4 and c5. Also, a3 and a4 both require vessel c1 to be full while they are being performed; a3 consumes the contents of vessels c2 and c4, and a4 consumes the contents of c3 and c5. Clearly the joint results of the incoming actions, matches the joint requirements of the outgoing actions. In our terms, there will be the following three incoming states for this time-point:
- c1, c2, and c3 full
- c4 and c5 full
- empty state (since neither of the incoming action occurrences has any prevail condition)

and the following three outgoing states:

- c2 and c4 full
- c3 and c5 full
- cl full (join of prevail conditions of outgoing action occurrences)

The same feature does not occur in several incoming states, which means that the incoming states are anti-dimensional. Also, if there were a conflict among the prevail conditions, the last incoming (resp. outgoing) state would be inconsistent. Finally, the balancing condition is that each incoming feature assignment is also an outgoing feature assignment, and vice versa. (End of example).

The requirements that have been specified here are not entirely sufficient; there is also an additional condition which is needed in order to avoid conflicting change of the same feature, by actions which are in the middle of two parallel actions sequences. For details please refer to ref. 1.

4. Representing concatenation of action structures.

We can now proceed to those extensions to the formalism which are needed in order to formulate the relationship between the workpiece program and the cell program in general terms. We shall need formal ways to concatenate action structures, for example those representing individual cycles, into larger structures. We shall also need a way of assigning, and dealing with the 'serial numbers' of the workpieces.

Let it be assumed, then, that a workpiece program and a cell program are given as action structures, using states from the same state space and actions from the same set A of valid actions. The states should not register the serial number or other identification of workpieces, but there may be features which e.g. signify the presence or absence of a workpiece in a machine, or which signify whether a machine has finished processing the workpiece that it is currently holding.

In order to analyze the cell program as in the example, we must first concatenate several cycles for the cell. There are well known formalisms which allow sequential composition of 'programs' in the sense of 'composite action descriptions', for example dynamic logic (ref. 2,3) and the programming language Occam (ref. 4). These notations however work only if the cycle has a single starting point and end-point, i.e. if the partial order < in the action structure [T, <, p) has a smallest and a largest element.

In our applications it is not possible to build up a cell program within the limits of single starting-points and end- points for all intermediate structures. We therefore need a structure which has several "entry points" and "exit points", in a way which resembles the concept of ports in programming languages. The concept of action structure is generalized as follows:

An *action fragment* is a fivetuple [T, <, m, p, r], where
T, <, p are like for action structures;
m and r are sequences of members of T;
m and r have equal length.

Also, the identity of the members of T is unimportant, so that if some other time-points are substituted throughout all five elements of the action fragment, we still have the same action fragment. (In other words we are really using the quotient structure w.r.t. the equivalence operation of permutation of time-points).

Consider two action fragments
g1 = |T1, <1 ml, p1, r1]
g2 = [T2, <2, m2, p2, r2]

Intuitively, ml contains the time-points along the "left edge" of gl, and r1 the time-points along its "right edge". Two fragments are concatenated by matching the right edge of one against the left edge of the next.

The *concatenated* action fragment gl;g2 is therefore obtained as follows: rename the time-points in T2 in such a way that rl=m2 (the elements in the two sequences are pairwise equal), and so that T1 and T2 are disjoint sets outside rl or m2. Then form
gl;g2 = (Tl u T2, <, ml, p1 u p2, r2]
where the relation < is obtained so that x < y iff either of the following conditions holds:
x < 1 y
x<2Y
for some z in rl, x<1z and z <2 y

It is easily seen that this definition corresponds to intuitions about the successive execution of (equal or different) cycles, and that the composition operation ; is associative.

Besides the criteria that have just been given, there must be additional conditions on the incoming and outgoing features in the edge nodes. We will show these conditions through a concrete example.

5. Representing the repeated cycle formally, in the example.

Figure 6 shows how the work cycle in the workcell example can be characterized as an action fragment, with the preconditions, postconditions and prevail conditions of the various action instances written out. The following graphical notations are used:

Full arrows (—) represent actions. The feature transitions of the action are written underneath the arrow. For example, 'Lx:4' under the X arrow means that while X is performed, the feature Lx should have the value 4 as a prevail condition, i.e. throughout the action. '4[Li]E' means that the Li feature should have the value 4 as a precondition and the value E as a postcondition. The operation marked in this way (namely A) changes the value of Li from 4 to E.

For each 'place' where a workpiece may be, there is a corresponding feature whose value is the serial number of the current workpiece in that place, or E if the place is empty. For example, the feature Lx represents the serial number of the workpiece that is currently in the X machine. Therefore the value of the feature Lx must be kept constant while X is performed. Also, we see in the figure how the move operations (A,B,C, and D) are characterized simply by how they change the values of the place features. Thus if the Lx place contains a workpiece with serial number 3, the Ly place contains no object (feature value E), and the B operation is performed, Ly receives the value 3 and Lx reverts to the value E.

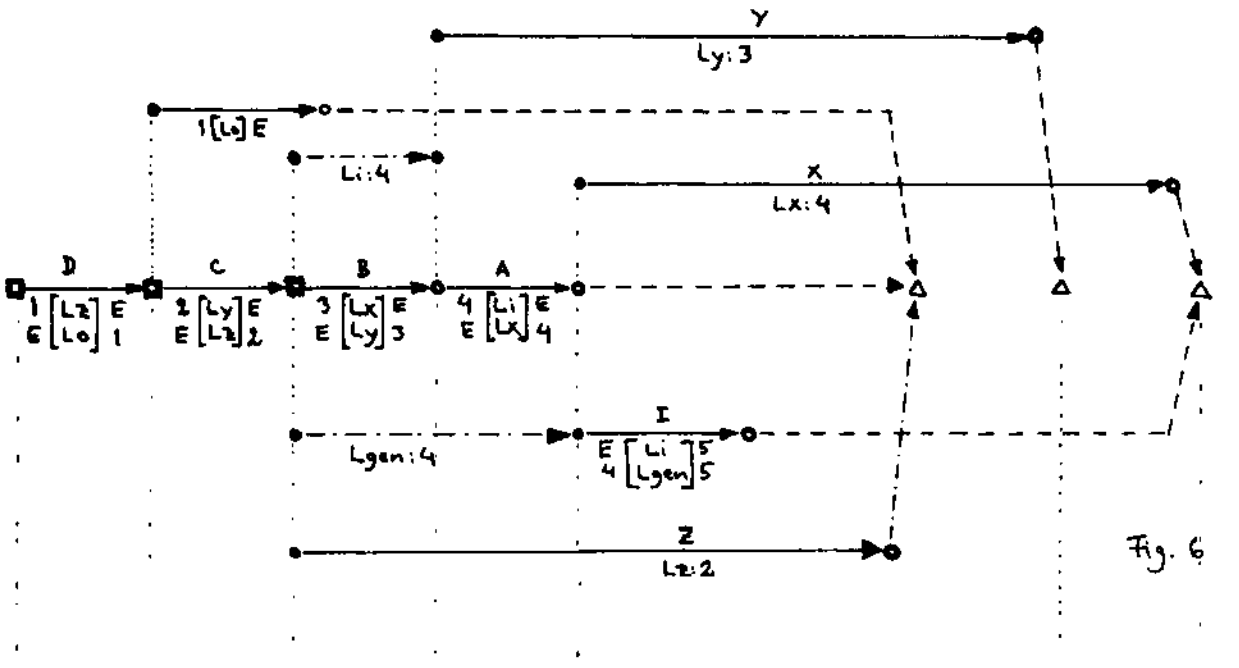

Fig. 6

Li	-	-	-/4	4/4	E/E	-	-	5/-
Lx	-	-	-/3	E/E	4/4	-	-	4/-
Ly	-	-/2	E/E	3/3	-	-	3/-	-
Lz	-/1	E/E	2/2	-	-	2/-	-	-
Lo	-/E	1/1	-	-	-	E/-	-	-
Lgen	-	-	-/4	-	4/4	-	-	5/-

Timepoints are represented as nodes in the graph. Square nodes represent timepoints on the left edge; triangle nodes represent timepoints on the right edge, and circular nodes represent other timepoints. Solid black nodes represent duplicate copies of the same timepoint in the graph, and dotted connections (....) connect one or more duplicates with the node they are duplicating.

Arrows with a broken line (—) represent the partial order < between nodes in those cases where there is no action connecting one to the other. (When there is such an action, the ordering < always follows). Broken/dotted arrows, finally, (-.-.-) represent additional persistence or 'noop' actions whicn have been inserted artificially in order to transfer the postcondition of one time-point to be a pre-condition of a later time-point. The broken/dotted line is omitted, and the persistence operations are understood, when there is already a broken line connecting the same nodes.

The lower part of the figure summarizes, for each node, which are the incoming and outgoing values of each of the features. For example, the node at the end of the B operation has the incoming values (4, E, 3) for the three features (Li, Lx, Ly), and the same outgoing values. This is an example of the node balancing condition on the action structure which was introduced at the end of section 3.

If two action fragments have been concatenated, then the nodes on their common edge must also satisfy the node balancing condition when actions from both the two fragments are used. However, if one only considers the action occurrences inside one fragment, then the edge nodes of that fragment usually do not satisfy the node balancing condition.

For those action fragments which are intended to be concatenated with themselves, in order to obtain cyclic behavior, and which we shall call *rtpcatable* action fragments, the balancing condition on the edge nodes can be obtained by combining corresponding left-edge and right-edge nodes. As is easily seen in figure 6, we can take the left-edge nodes, increase all serial-number values with 1, form the join with the corresponding right-edge node, where the join is taken separately for the incoming state and the outgoing state of the node, and then the node balancing criterium is satisfied.

Consider for example the second left-edge node, between the actions D and C in the figure. For the features (Ly, Lz, Lo), this node has incoming values (-, E, I) and outgoing values (2, E, 1), or the value pairs (-/2, E/E, 1/1). The second right-edge node has the value pairs (3/-, -/-, -/-) for the same features. Incrementing all left-edge values by 1 (except E which remains unchanged) we obtain (-/3, E/E, 2/2). Combining with the right- edge node we obtain (3/3, E/E, 2/2) which satisfies the node balancing criterium. (Here we let - represent 'undefined').

Most of the features (Li, Lx, Ly, Lz, Lo) represent the various positions in the cell where a workpiece may 'be', and most of the actions represent either the movement of a workpiece, or the operation on a workpiece inside one of the machines. The last feature, Lgen, is used as the counter that assigns serial numbers. We see how the operation I, which advances the incoming conveyor, changes the contents of the Li location from E to 5 in the figure, and at the same time it increases Lgen from 4 to 5. In general, valid I actions are those where Lgen is increased by one, and Li changes from E to the new value of Lgen. The auxiliary, 'persistence' arrows transmit the value of Lgen from the third left-edge node to the beginning node of the I action, and from the end node of the I action to the third right-edge node.

In summary, then, figure 6 shows how the simple action-plan from figure 3 could be 'decorated' with feature values for preconditions, postconditions, and prevail conditions, and how it is extended with auxiliary persistence actions, so that the resulting action-plan satisfies a revised node balancing criterium which also takes the edge nodes into account.

6. Relationship between workpiece program and cell program in general and formal terms.

We are now ready for the goal of this article, namely to formulate in precise terms the relationship between a workpiece program and a proposed cell program. We assume that the workpiece program is given as an action structure

gw = [Tw, $<_w$, pw]

and that the proposed cell program is given as a repeatable action fragment

gc = [Tc, $<_c$, mc, pc, re]

The same feature domains should be used for gw and gc, with the following exceptions:
- the 'counter' feature Lgen is omitted in gw;
- the defined values of the position features is E or serial number in gc, but E or T (E = empty, T = taken) in gw.

Let succ(gc) be the modified action fragment obtained from gc by increasing all numerical feature values with 1, and leaving the others (E and undefined) unchanged. It follows from the construction, and the criterium for well-formedness and repeatability for fragments, that succ(gc) is also a correctly formed, and repeatable action fragment. (In fact a number of fairly routine constraints must also be satisfied, such as the existence of arcs that forward the current value of the Lgen 'counter' to the next cycle, but this is not of any particular interest). Therefore the cyclically repeated action fragment

gc* = gc ; succ(gc) ; succ(succ(gc)) ...

is well formed.

If [f,b,v,e] is an action, and k is an integer, then the corresponding *purged* action

purge([f,b,v,e],k)

is obtained in the following fashion: first remove from f, b, and e the Lgen feature and all features which have a numerical value different from k in either of f, b, or e. The only remaining values are then k, E, and undefined. Then change all occurrences of the k value to the value T.

For example, a feature which has the value E in b and k in e is retained, but with the value T in e. A feature which has the value k+1 in b and the value E in e is removed from both b and e, i.e. obtains the value undefined in both.

The purge of an action occurrence [t, [f,b,v,e], t'] is obtained by purging its middle element.

If k is an integer, and gc = [Tc, $\leq_c$, mc, pc, re], let the *extracted* action Extr(gc,k) be defined as that action structure

[Tk, $\leq_k$, pk]

where:

pk consists of purge(a,k) for those action occurrences

a = [t,[f,b,v,e],t']

in pc where some feature except Lgen is defined and has the value k in either f, b or e;

Tk consists of those t in Tc which occur in at least one action occurrence in pk;

$\leq_k$ is the restriction of $\leq_c$ to Tk.

It is easily seen that

Extr(gc*, k)

is independent of k if k is > some small N (whose value is a function of how the Lgen feature values were chosen in gc to begin with). This Extr(gc*,k) for sufficiently large k will be written Extr(gc*).

The requirement on correspondence between gw and gc is now that, if Extr(gc*) = [Tr, <*, prl, after suitable permutation of the time-points, we snail nave:

$$Tw \subseteq Tr$$
$$t \leq_w t \quad \rightarrow \quad t \leq_r t'$$
$$pw \subseteq pr$$

If the other, previously mentioned conditions are satisfied, i.e. gw and gc are coherent etc., and gc is repeatable, and if the correspondence condition is satisfied, then the cell program gc will correctly implement the workpiece program gw on successive workpieces in pipeline fashion.

7. Discussion, limitations.

The most obvious limitation on the result above is that it only applies to the 'steady state' when the production process is running, all machines contain a workpiece each, etc. The analysis does not cover the start-up and shut-down stages, or the handling of exceptional cases such as when a workpiece breaks and has to be taken out of the cell. The analysis of those situations is a natural next step.

The analysis also depends on the peculiar characters of the processing and move operations. If one would add e.g. an atomic 'swap[1] operation which exchanges the contents of two locations, then the definition of the 'purge' function and the subsequent analysis would have to be revised.

The formal treatment here only considers one type of workpieces, and requires non-trivial extensions for mixed-mode production where several types of workpieces are mixed in the same flow.

A number of formal details have had to be omitted in this version of the paper, which is only intended for overview. The more precise version of the paper must be consulted by the reader who is interested in the exact and complete formulation of the conditions for correspondence between workpiece program and cell program.

For similar reasons, the detailed aspects of how to generalize the requirements at the end of section 3 for edge-nodes in action fragments have been omitted here.

Related work. The theories and languages for concurrent programming address the issue of specifying 'two or more sequential programs that may be executed concurrently as parallel processes' (ref. 5). Their goal is therefore different from the goal of the present work, which is to characterize parallel processes in the world outside the computer, but evidently the techniques may sometimes be interchangable.

The method of *path expressions* (ref. 6) is similar in some respects to the approach taken in this paper. In particular, path expressions also separate the specification of operations from the the specification of constraints on the execution order. However, the analysis of preconditions/ postconditions/ prevail-conditions does not (to our knowledge) have a counterpart in path expressions. Also the use of 'action fragments' whose left and right edges are sequences of time-points or execution states, are believed to be novel.

A more extensive survey of methods from concurrent programming, and their relevance to the description of action structures is given in section 11 of (ref. 1).

Pipelining is a well known technique also in computer architectures. The classical technique for analysis of such pipelines is based on 'reservation tables' (ref. 7) which specify which resource(s) are needed in each stage of processing, but which do not easily lend themselves to the analysis of the three types of logical conditions which are at the core of our approach.

Intentions for continued work. Our own plans, for a longer perspective, is that we would like to have a formal characterization of all the essential steps when going from the product information in the CAD system, to the production cell program, including also the formal description of the production equipment, the procedures for dealing with exceptional situations, etc. Furthermore, the formal characterization should be usable as a specification and/or as a knowledge base for the actual software that controls the production cell. The work in this paper is one of the steps towards that goal.

References.

1. E. Sandewall and R. Ronnquist: "A Representation of Action Structures". Proc. 1986 AAAI.

2. D. Harel: "Dynamic Logic". In "Handbook of Philosophical Logic", Vol. 2, Reidel publishing Co., Holland/USA.

3. R. Parikh: "Propositional Dynamic Logics of Programs: A Survey". In "Logics of Programs", Ed. E. Engeler, Springer LNCS 125, pp. 102-144.

4. D. May: "Occam". SIGPLAN Notices, April 1983. (Occam is a trademark of Inmos Ltd.)

5. G.R. Andrews and F.B. Schneider: "Concepts and Notations for Concurrent Programming". Computing Surveys, Vol. 15, No. 1, March 1983.

6. R.H. Campbell and A.N. Habermann: "The specification of process synchronization by path expressions". Springer LNCS, vol. 16, 1974, pp. 89-102.

7. E. Davidson: "The Design and Control of Pipelined Function Generators". In Proc. 1971 International IEEE Conference on Systems, Networks, and Computers, pp. 19-21, 1971.