Automatically Generating Plans for Manufacturing*

Billy Harris† **Diane J. Cook**‡ **Frank Lewis**§

January 14, 2000

Abstract

While machine planning has attracted great interest among researchers, it has seldom been used outside research labs. One impediment to wide-spread use is that existing planners are often difficult to integrate with other parts of a manufacturing system. We address this problem by showing how assembly trees (constructs often used by factories identifying how to construct an object) can easily be converted into HTN operators for our machine planner. We also demonstrate that our plans can be easily converted to a Petri-Net or matrix representation which ordinary discrete-event controllers can manipulate. We view our planner as one portion of a complete control system. We also demonstrate how our system can combine multiple alternatives into a single representation. Finally, we show that the combined representation can be converted back into a more conventional plan representation, allowing machine planners to compactly reason about alternate courses of action.

keywords: Machine planning, HTN planning, Intelligent control, discrete-event control, manufacturing

---

* This research was supported in part by the National Science Foundation, under grant GER-9355110.

† wharris@cse.uta.edu, Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019

‡ cook@cse.uta.edu, Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019

§ flewis@arri.uta.edu, Automation & Robotics Research Institute, 7300 Jack Newell Blvd. South, Fort Worth, TX 76118

1 INTRODUCTION

Global competition has forced manufacturers to continually inovate. Just-In-Time manufacturing, Kanban systems, and other technologies have allowed factories to reduce their inventory level and adapt quickly to changing market conditions. Shop floor managers have become increasingly interested in so-called flexible manufacturing systems which can quickly transition between products with minimal human intervention. We have addressed this problem from an AI perspective and have shown that a conventional AI planner can use the same repsentation that shop-floor managers currently use in their manufacturing cells and have outlined how this planner can be integrated into a complete control system. We have also combined ideas from AI planning and ideas from manufacturing into a compact representation for alternate courses of action.

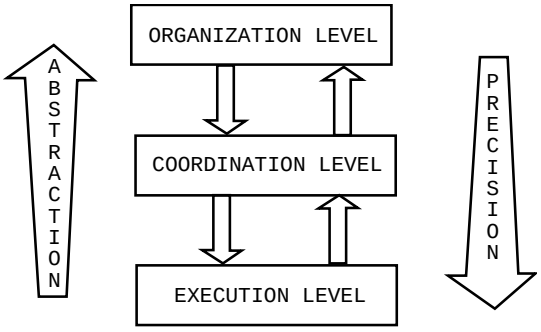


Figure 1: Three-Level Intelligent Control Architecture

Figure 1 shows Saridis's abstraction hierarchy (Saridis, 1983.). In this hierarchy, components in the Organization level function as managers, components in the Coordination level provide job-sequencing abilities, and components in the Execution level carry out the prescribed actions. Each level of Saridis's abstraction hierarchy may have more than one component; each component must sense current conditions, form a model of the world, make decisions, and give commands or status reports to other components.

Most of our paper discusses our planner, which belongs to the Organization level. Section 2 provides an introduction to machine planning; Section 3 describes our method for easily converting assembly trees into HTN operators. We discuss resources in Section 4. Section 4 also describes how we convert our plan representation into a set of matrices usable by our discrete-event controller.

Section 5 describes our control system. The controller belongs to Saridis's Coordination level; it decides when to execute each step of our plan based on which resources are available. In Section 6, we describe

how our system can combine multiple plans into a single representation; thus, our controller can decide in real-time which of several methods to use to assemble a particular part.

Section 5 briefly mentions the closed-loop controllers used for each workstation; each such controller belongs to Saridis's Execution level.

We conclude our discussion in Section 7.

2 MACHINE PLANNING BACKGROUND

This section introduces machine planning terms and gives a brief summary of machine planning research.

2.1 Introduction to Machine Planning

Machine planners seek to find a sequence of steps which can be executed to meet some specified goals in a particular domain. Machine planners must have a description of the domain and of the specific problem. *Operators* encode knowledge of how an agent's actions affect the world. Most planners use a variant of STRIPS operators, introduced by Fikes, Hart, and Nilsson (Fikes & Nilsson, 1971).

STRIPS operators contain three lists of first-order predicate calculus expressions. The precondition list contains predicates that must hold for the operator to be applicable. The add list contains predicates that will hold after the operator is executed and the delete list contains predicates that no longer hold after the operator is executed. Predicates which do not appear in either the add list or the delete list do not change their truth-values during the application of the operator. In most planners, the add list and delete list are combined to form the operator's effects or postconditions.

Here is a sample STRIPS-style operator:

Name:	Pickup
Parameters:	?BLOCK
Variables:	?BLOCK ?SUPPORT
Preconditions:	(HAND-EMPTY) (CLEAR ?BLOCK) (ON ?BLOCK ?SUPPORT)
Delete List:	(HAND-EMPTY) (ON ?BLOCK ?SUPPORT)
Add List:	(CARRYING ?BLOCK) (CLEAR ?SUPPORT)

This operator, from the Blocksworld domain, describes the action of picking up a block. To successfully pick up a block, the agent must have an empty hand, and the block the agent is picking up must be clear (meaning the block has no other blocks on top of it). After executing this operator, the agent's hand is no longer empty and the block is no longer on its support. Instead, the agent is now carrying the block and the block's old support is now clear. There are usually many ways to divide a domain into operators and many ways to encode each operator.

In addition to a domain description, machine planners need a description of a particular problem. Problems are normally represented by an initial state and a goal state, each a set of predicates from the appropriate domain. The *initial state* consists of the set of predicates completely describing the world's situation when the planner begins to plan. The *goal state* consists of a set of predicates which should be true when the plan has finished executing. Since literals not mentioned in the goal state description may be either true or false, the “goal state” actually describes a set of possible world states.

Originally, generated plans consisted of a totally ordered sequence of steps. Planners using *partial order planning*, introduced with Sacerdoti's NOAH system (Sacerdoti, 1975), produce plans with only a partially ordered sequence of steps. Partial ordering gives the plant’s executor some flexibility in the exact order the steps are followed. In a manufacturing context, two or more steps may be executed at the same time.

2.2 Hierarchical Task Network Planning

An alternative to traditional planning is hierarchical task network planning, or *HTN planning*[1]. In HTN planning, a planning system receives task schemas as well as traditional operator descriptions (Wilkins, 1984). Task schemas provide a method of grouping operators together to form higher-level operations. For example, Austin Tate uses this schema in his planner NONLIN (Tate, 1977):

[1] Other names for HTN planning include Task Network planning, Task Reduction Planning, Task-based planning, and Action-based planning.

```
(opschema makeclear
          :todo (cleartop ?x)
          :expansion (
                      (step1 :goal (cleartop ?y))
                      (step2 :action (puton ?y ?z))
                      )
          :orderings ((step1 -> step2))
          :conditions (
                       (:use-when (on ?y ?x) :at step2)
                       (:use-when (cleartop ?z) :at step2)
                       (:use-when (not (equal ?z ?y)) :at step1)
                       (:use-when (not (equal ?x ?z)) :at step1)
                       )
          :variables (?x ?y ?z)
        )
```

Task schemas allow HTN planners to discover plans faster than conventional planners. A conventional planner trying to clear block X would need to search for all operators with (cleartop X) as a postcondition. One such operator, (putdown X), would ultimately need (cleartop X) as its precondition. Thus, the conventional planner must backtrack until it stumbles upon (pickup Y) as the correct action to achieve (cleartop X). This particular schema is short and could be represented as a control rule or learned by planners using explanation-based learning (Minton et al., 1987). In general, however, HTN schemas can become quite complex and more expressive than conventional operator descriptions (Wilkins, 1994).

2.3 Plan Representation

Planners offer various extensions to the STRIPS paradigm and thus use different internal representations for their plans. However, each representation has a notion of "primitive" actions which are directly executable by an agent and a notion of ordering constraints specifying that one operation must complete before another can begin. Figure 2 shows a sample plan. The plan says that A must be drilled and B must be obtained before the agent can execute the operation "(Attach B C)," but these two steps can be executed in either order, or executed simultaneously.

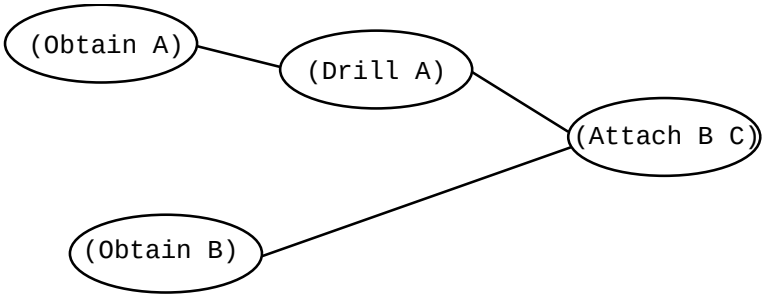


Figure 2: Sample Plan

3 CONVERTING ASSEMBLY TREES TO HTN OPERATORS

Manufacturers use assembly trees to represent manufacturing operations; machine planners use domain operators to represent manufacturing operations. In this section, we describe how to easily convert assembly trees into HTN operators.

3.1 Representing Assembly Trees

Traditionally, manufacturers have used assembly trees (Wolter, Chakrabarty & Tsao, 1992) to represent knowledge of how to construct a given component. Assembly trees (or, equivalently, a Bill of Materials (Baker, 1974)) can be viewed as a matrix for which entry i, j has a value of 1 if job j is an immediate prerequisite for job i. Assembly trees do not consider the resources needed to execute the jobs; they contain only product-specific job sequencing information.

We present assembly trees graphically; edges of an assembly tree correspond to manufacturing operations and nodes correspond to parts or subassemblies. Figure 3 shows a sample assembly tree. A work cell can create part B by obtaining part A and drilling it. The cell can assemble parts B and C to form the single component D.

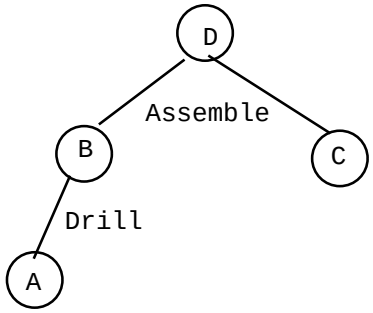


Figure 3: Sample Assembly Tree

3.2 Relating Assembly Trees and HTN Operators

Non-leaf nodes in an assembly tree represent the result of applying some manufacturing operation. Thus, the node's label identifies the goal accomplished by the corresponding plan operator. Children of an assembly tree node represent parts used by the manufacturing operation and correspond to subgoals in the domain operator.

Assembly tree arcs represent specific manufacturing operations and correspond to plan steps (primitive actions) in our plan operator. Figure 4 summarizes these correspondences. Figure 5 shows a sample assembly tree; using our approach, the interior nodes can be converted into NONLIN (Tate, 1977) operators shown in Figure 6.

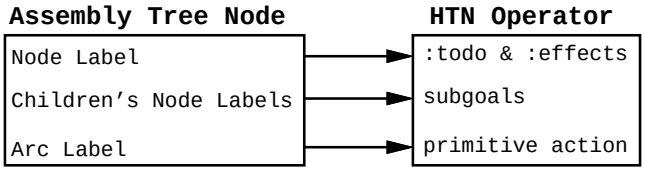


Figure 4: Converting Assembly Trees into HTN Operators

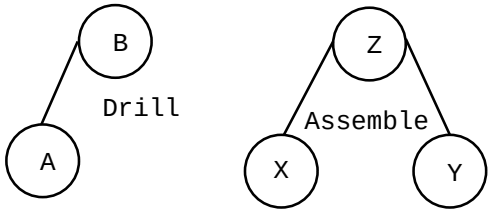


Figure 5: Sample Assembly Trees

```
(actschema Build-B
        :todo (Assembled B)
        :expansion ( (step1 :goal (Assembled A))
                     (step2 :primitive (Drill A)))
        :effects ( (step2 :assert (Assembled B)))
        :orderings ((step1 -> step2)))

(actschema Build-Z
        :todo (Assembled Z)
        :expansion ( (step1 :goal (Assembled X))
                     (step2 :goal (Assembled Y))
                     (step3 :primitive (Attach X Y)))
        :effects ( (step3 :assert (Assembled Z)))
        :orderings ((step1 -> step3) (step2 -> step3)))
```

Figure 6: Operator Descriptions

Leaf nodes in an assembly tree correspond to incoming parts—parts which our manufacturing cell does not produce locally. These can be represented as HTN operators which have no subgoals. For example, if part *A* of Figure 5 is not locally produced, the HTN operator shown in Figure 7 can "assemble" it without forming subgoals.

```
(actschema Prepare-A
         :todo (Assembled A)
         :expansion ( (step1 :primitive (PutOn A Pallet)))
         :effects ( (step1 :assert (Assembled A))))
```

Figure 7: Handling Product-Ins

3.3 Flowlines, Assembly, and Job-shops

Flowlines represent a sequence of simple manufacturing operations. In assembly trees, flowlines correspond to a sequence of nodes with only one child each. Our previous section had an example of a very short flow line; the plan in Figure 8 shows the resulting plan.

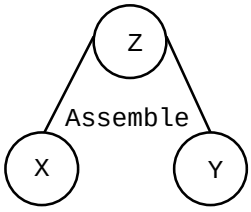
Figure 8: Representing Flowlines

Assembly operations represent attaching two (or more) parts together. In assembly trees, steps corresponding to assembly operations have more than one child. The HTN operators for assembly steps have more than one subgoal, and the resulting plans use partial ordering to allow the cell to complete the subassemblies in any order. The right portion of Figure 5 shows a assembly tree containing an assembly operation; Figure 6 includes the corresponding HTN operators. Figure 9 shows a plan involving assembly; the two subgoals may be achieved in either order (or simultaneously).

If two or more operations must be performed on the same part, but the operations may be executed in any order, then the job dispatcher faces a job-shop choice. Assembly trees do not have a standard method of representing job-shop choices; Figure 10a shows two possible methods of representing choices. The left assembly tree shows a “compound” operation in which one arc incorporates two distinct steps which may be performed in either order. The assembly trees on the right explicitly represent the alternate orderings as separate but related assembly trees.

Figure 10b shows HTN operators for this representation. Plan operators provide an intuitive representation of the job-shop choice; part A must be drilled and sanded but these two operations may be

performed in any order. The resulting plan (in Figure 10c) splits into two different sections and then rejoins. In Figure 9, the joining of two plan strands represents an assembly operation; the two “threads” which join are operating on different parts and the manufacturing step at the merged node (Attach X Y) indicates that this portion of the plan corresponds to an assembly operation. In contrast, the threads in Figure 10 correspond to operations on an identical part, and the manufacturing step at the merged node (Clean A) does not imply an assembly operation. Thus, the merging of two threads in Figure 10 corresponds to the end of a job-shop choice and not to an assembly operation.




```
(actschema Build-Z
         :todo (Assembled Z)
         :expansion ( (step1 :goal (Assembled X))
                      (step2 :goal (Assembled Y))
                      (step3 :primative (Attach X Y)))
         :effects ( (step3 :assert (Assembled Z)))
         :orderings ((step1 -> step3) (step2 -> step3)))

(actschema Prepare-X
         :todo (Assembled X)
         :expansion ( (step1 :primitive (PutOn X Pallet)))
         :effects ( (step1 :assert (Assembled X))))

(actschema Prepare-Y
         :todo (Assembled Y)
         :expansion ( (step1 :primitive (PutOn Y Pallet)))
         :effects ( (step1 :assert (Assembled Y))))
```

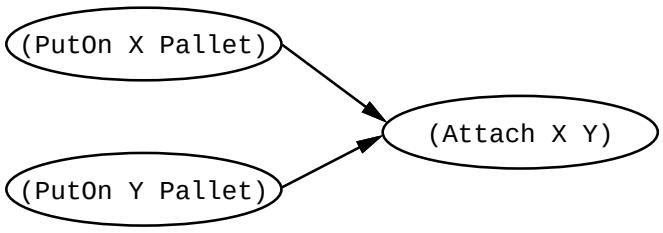
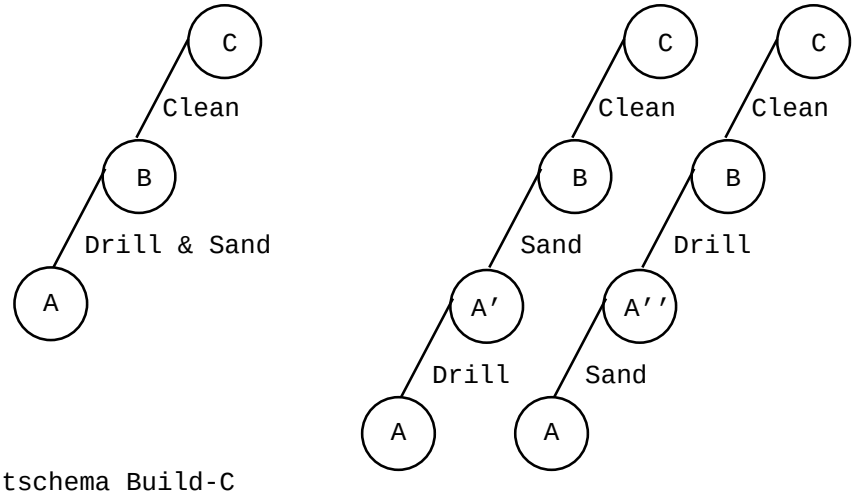


Figure 9: Representing Assembly

Thus, HTN operators can represent all information stored in an assembly tree and can also represent job-shop scheduling choices that are difficult to represent in assembly trees. We will later show in Section 6 that a

planner can also consider alternate methods of constructing parts (corresponding to multiple assembly trees).

A)



B)

```
(actschema Build-C
        :todo (Assembled C)
        :expansion ( (step1 :goal (Drilled A))
                     (step2 :goal (Sanded A))
                     (step3 :primitive (Clean A)))
        :orderings ((step1 -> step3) (step2 -> step3))

(actschma Drill-A
        :todo (Drilled A)
        :expansion ( (step1 :goal (Assembled A))
                     (step2 :primitive (Drill A)))
        :effects ( (step2 :assert (Drilled A)))
        :orderings ((step1 -> step2)))

(actschma Sand-A
        :todo (Sanded A)
        :expansion ( (step1 :goal (Assembled A))
                     (step2 :primitive (Samd A)))
        :effects ( (step2 :assert (Sanded A)))
        :orderings ((step1 -> step2)))

(actschema Prepare-A
        :todo (Assmebled A)
        :expansion ( (step1 :primitive (PutOn A Pallet)))
        :effects (step1 :assert (Assembled A)))
```

C)

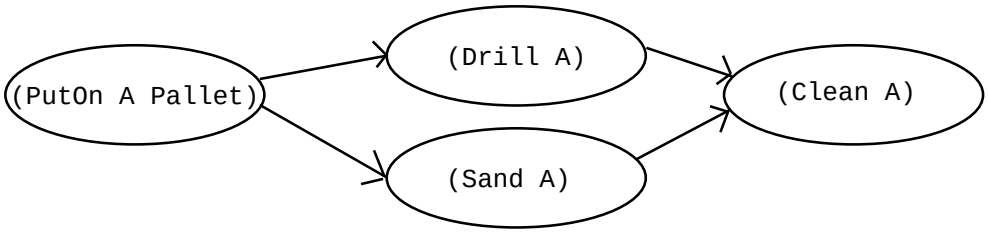


Figure 10: Representing Job-Shop Choices

4 CONVERTING PLANS TO MATRICES

Current controllers used in manufacturing systems use matrices to describe the ordering of jobs and the resources needed to perform the jobs. In this section, we describe how our plans may be converted into two matrices, F_v and S_v ,which incorporate the plan's job sequencing information. We describe how the structure of the plan and information on which resources are currently available can be combined into two additional matrices, F_r and S_r , which describe the resources needed to perform the plan steps. These matrices can be interpreted as a Petri-Net, so we begin by describing Petri-Nets.

4.1 Petri-Nets

Petri-Nets can be represented mathematically by several sets:

- P, a set of places. Initially, each place represents a particular action of our plan. Later, we add places representing resources needed by plan actions. Each place can hold one or more *tokens*. Tokens residing in an action place mean that the action has been performed on one ormore parts. Tokens residing in a resource place indicate that one or more instances of that resource are available for consumption.

- T, a set of transitions. Each transition indicates the cessation of one action and the initiation of another action, and the corresponding release of one resource and the acquisition of another resource.

- I, an "Input Set" mapping places to transitions. When transition T_i fires, tokens are removed from each place P_j for which (P_j, T_i) is an element of I.

- O, an "Output Set" mapping transitions to places. When transition T_i fires, tokens are inserted into each place P_j for which (T_i, P_j) is an element of O.

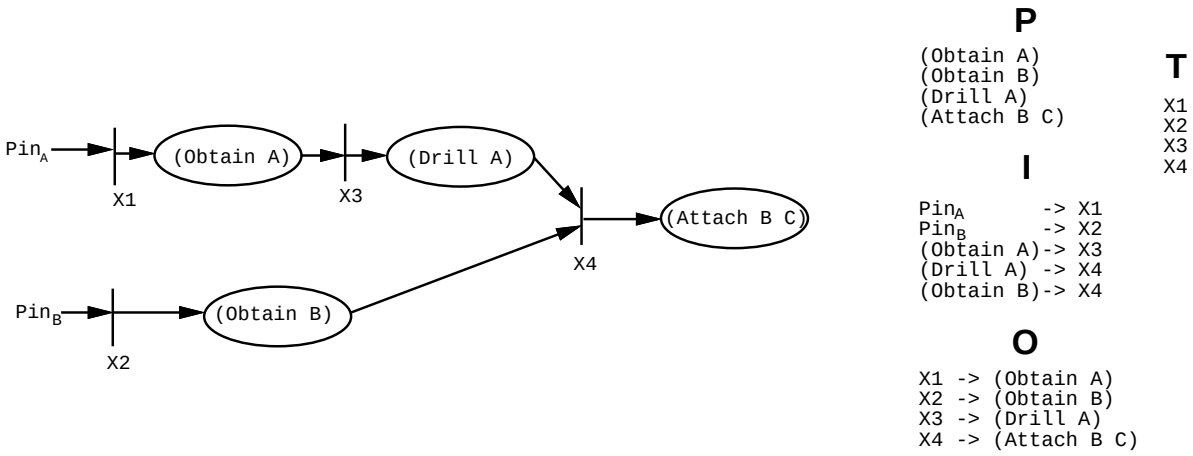


Figure 11: A Sample Petri-Net

Figure 11 shows a sample Petri-Net. In this Petri-Net, both (Drill A) and (Obtain B) must be complete (have tokens in them) before transition X_4 can fire, but the two tasks may be performed in either order. When X_4 fires, tokens are removed from (Drill A) and (Obtain B), and the controller executes the (Attach B C) task. When this task completes, a token is placed in the (Attach B C) node and other transitions may then fire.

Manufacturing researchers have studied Petri Nets extensively (Desrochers, 1990; Jeng & DiCesare, 1992; Murata et al., 1986; Zhou & DiCesare, 1993); researchers have investigated job sequencing controller design, deadlock avoidance, reachability analysis, and system liveness tests.

4.2 Converting Plans to Petri-Nets

Figure 12 shows a sample plan which includes assembly steps and routing choices. In particular, to complete the plan, an agent must perform both step F and step G, but the agent may perform these two steps in either order. Figure 13 shows the corresponding Petri-Net in which the possible routing choices have been explicitly listed. The agent can either perform steps $F1$ and $G1$, meaning the agent performs step F first, or the agent can perform steps $G2$ and $F2$, meaning the agent performs step G first. Each alternative is given a unique label to prevent the alternatives from being merged by our algorithm for combining multiple plans (described in Section 6).

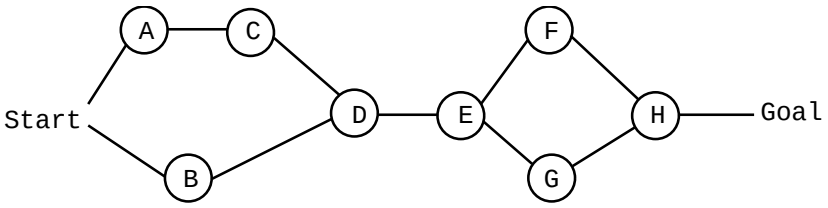


Figure 12: A Sample Plan

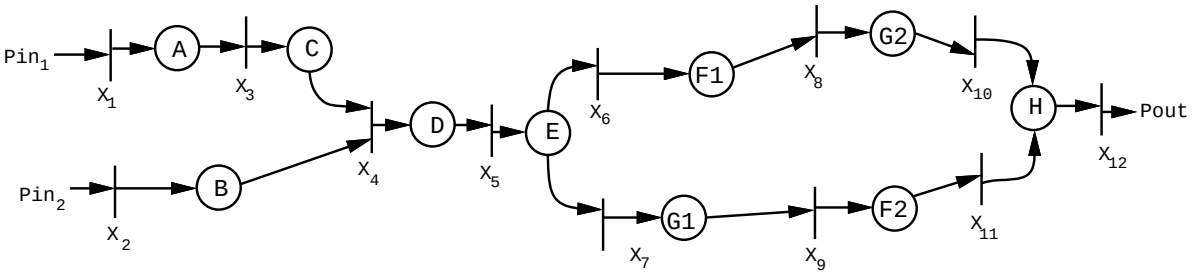


Figure 13: Petri Net Representation of Plan

4.3 Converting Plans to Matrices

The Petri-Net in Figure 13 is equivalent to the two matrices shown in Figure14. The F_v matrix maps actions to transitions and corresponds to the I set of the Petri-Net; a 1 in location i, j means that transition X_i cannot fire until action A_j completes. The S_v matrix maps transitions into actions and corresponds to the O set of the Petri-Net; a 1 in location i, j of this matrix means that when transition X_j fires, action A_i is started. Assembly operations are signaled by two or more 1s in a single row of F_v. In our example D is the action of assembling the parts produced by B and C; the X_4 transition has two 1s indicating the assembly step. The start of a routing decision is signaled by having more than one 1 in a column of F_v; in our example, E can enable either X_6 or X_7. The end of a routing decision is signaled by two or more 1s in the same row of S_v; action H will be started after either X_{10} or X_{11} fires.

$F_V =$

	P_{inA}	P_{inB}	A	B	C	D	E	F1	G1	G2	F2	H
x_1	1	0	0	0	0	0	0	0	0	0	0	0
x_2	0	1	0	0	0	0	0	0	0	0	0	0
x_3	0	0	1	0	0	0	0	0	0	0	0	0
x_4	0	0	0	1	1	0	0	0	0	0	0	0
x_5	0	0	0	0	0	1	0	0	0	0	0	0
x_6	0	0	0	0	0	0	1	0	0	0	0	0
x_7	0	0	0	0	0	0	1	0	0	0	0	0
x_8	0	0	0	0	0	0	0	1	0	0	0	0
x_9	0	0	0	0	0	0	0	0	1	0	0	0
x_{10}	0	0	0	0	0	0	0	0	0	1	0	0
x_{11}	0	0	0	0	0	0	0	0	0	0	1	0
x_{12}	0	0	0	0	0	0	0	0	0	0	0	1

$S_V =$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
A	1	0	0	0	0	0	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0
C	0	0	1	0	0	0	0	0	0	0	0	0
D	0	0	0	1	0	0	0	0	0	0	0	0
E	0	0	0	0	1	0	0	0	0	0	0	0
F1	0	0	0	0	0	1	0	0	0	0	0	0
G1	0	0	0	0	0	0	1	0	0	0	0	0
G2	0	0	0	0	0	0	0	1	0	0	0	0
F2	0	0	0	0	0	0	0	0	1	0	0	0
H	0	0	0	0	0	0	0	0	0	1	1	0
P_{out}	0	0	0	0	0	0	0	0	0	0	0	1

Figure 14: F_v and S_v Matrices

4.4 Incorporating Resource Information

When we execute our plan, we will need to use some resources. For example, action C in our plan may correspond to a painting operation; to perform this action (i.e. to fire transition X_3), we must use some type of painting tool. We use two additional matrices, F_r and S_r , to represent the resources needed to perform the jobs. A 1 in position i, j of the F_r matrix means that resource R_j must be secured before transition X_i can fire. A 1 in position i, j in the S_r matrix means that when transition X_j fires, resource R_i is released. If a column j of F_r contains more than one 1, resource j is being shared by more than one job. Matrix F_r has been called the resource requirements matrix (Kusiak, 1992).

Initially, we assume that every action has a dedicated resource. That is, if a transition starts action A_i, it will also reserve a dedicated resource $\widehat{R}_i$, and if the completion of an action A_j causes a transition to fire, the

transition will also release a resource $\hat{R}_j$. Figure 15 shows the Petri-Net describing these dedicated resources. Notice that this Petri-Net is identical to the one in Figure 13 except for the direction of the arcs. Because of this similarity, we can quickly form resource matrices for our dedicated resources:

$\hat{F}_r = S_v^T$, with the product-out column(s) removed.

$\hat{S}_r = F_v^T$, with the product-in row(s) removed.

Figure 16 shows the resulting matrices.

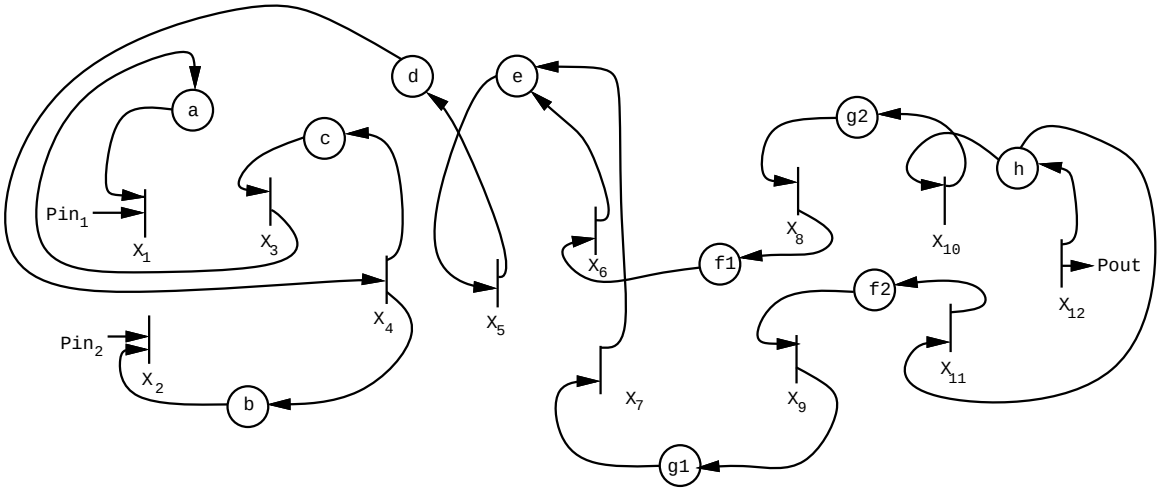


Figure 15: Petri Net for Dedicated Resources

4.5 Resource Assignment

We formed our initial resource matrices by assuming that each action has a dedicated resource. In most cases, this assumption is unrealistic. For example, if actions A, D, and E all involve drilling operations but we only have one drill, our drill must be shared by the three operations. In terms of dedicated resources, this sharing means that $\hat{a}$, $\hat{d}$, and $\hat{e}$ all correspond to the same actual resource. We represent the sharing in a resource assignment matrix F_a. If F_a contains a 1 in position i,j, then our actual resource R_j is performing the duties of our idealized dedicated resource $\hat{R}_i$. Shared resources are represented by columns of F_a containing two or more

$\hat{F}_r =$

	$\hat{a}$	$\hat{b}$	$\hat{c}$	$\hat{d}$	$\hat{e}$	$\hat{f1}$	$\hat{g1}$	$\hat{g2}$	$\hat{f2}$	$\hat{h}$
x_1	1	0	0	0	0	0	0	0	0	0
x_2	0	1	0	0	0	0	0	0	0	0
x_3	0	0	1	0	0	0	0	0	0	0
x_4	0	0	0	1	0	0	0	0	0	0
x_5	0	0	0	0	1	0	0	0	0	0
x_6	0	0	0	0	0	1	0	0	0	0
x_7	0	0	0	0	0	0	1	0	0	0
x_8	0	0	0	0	0	0	0	1	0	0
x_9	0	0	0	0	0	0	0	0	1	0
x_{10}	0	0	0	0	0	0	0	0	0	1
x_{11}	0	0	0	0	0	0	0	0	0	1
x_{12}	0	0	0	0	0	0	0	0	0	0

$\hat{S}_r =$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
$\hat{a}$	0	0	1	0	0	0	0	0	0	0	0	0
$\hat{b}$	0	0	0	1	0	0	0	0	0	0	0	0
$\hat{c}$	0	0	0	1	0	0	0	0	0	0	0	0
$\hat{d}$	0	0	0	0	1	0	0	0	0	0	0	0
$\hat{e}$	0	0	0	0	0	1	1	0	0	0	0	0
$\hat{f1}$	0	0	0	0	0	0	0	1	0	0	0	0
$\hat{g1}$	0	0	0	0	0	0	0	0	1	0	0	0
$\hat{g2}$	0	0	0	0	0	0	0	0	0	1	0	0
$\hat{f2}$	0	0	0	0	0	0	0	0	0	0	1	0
$\hat{h}$	0	0	0	0	0	0	0	0	0	0	0	1

Figure 16: $\hat{S}_r$ and $\hat{F}_r$ Matrices

1s. Figure 17 shows a resource assignment matrix containing two shared resources; a single resource *ade* performs the functions of the generic resources $\hat{a}$, $\hat{d}$, and $\hat{e}$ (and thus will be shared among actions *A*, *D*, and *E*) and a single resource *f* will perform the functions of generic resources $\hat{f}1$ and $\hat{f}2$ (and thus will be shared by actions *F*1 and *F*2).

Once we have formed our resource assignment matrix, we can easily compute our final S_r and F_r matrices:

$$F_r = \hat{F}_r * F_a$$

$$S_r = F_a^T * \hat{S}_r$$

$F_a =$

	ade	b	c	f	g1	g2	h
a	1	0	0	0	0	0	0
b	0	1	0	0	0	0	0
c	0	0	1	0	0	0	0
d	1	0	0	0	0	0	0
e	1	0	0	0	0	0	0
f1	0	0	0	1	0	0	0
g1	0	0	0	0	1	0	0
g2	0	0	0	0	0	1	0
f2	0	0	0	1	0	0	0
h	0	0	0	0	0	0	1

Figure 17: F_a Assigns Resources

In some cases, our values for F_r and S_r may contain self-loops. For example, consider resource ade. With dedicated resources, transition X_5 reserves $\hat{e}$ and releases $\hat{d}$. Now, transition X_5 reserves resource ade and releases the same resource ade. This behavior is not correct; intuitively, it means that at the instant X_5 fires, two uses of resource ade are held. We eliminate this self-loop by finding i,j pairs such that $F_r[i,j] = S_r[j,i] = 1$ and changing both values to 0. This action corresponds to three matrix equations, in which "&" represents an element-by-element logical AND operation and "-" represents ordinary matrix subtraction:

$$T_s = F_r \,\&\, S_r^T$$
$$F_{r_{new}} = F_{r_{old}} - T_s$$
$$S_{r_{new}} = S_{r_{old}} - T_s^T$$

Figure 18 shows the F_r and S_r matrices we get after applying the resource assignment shown in Figure 17 and removing the self-loop. Note that places from the F_v and S_v matrices (Figure 14) describe actions and places from the F_r and S_r matrices describe resources. Figure 19 shows the complete Petri-Net with both job places and resource places. When one action completes and when the resource needed for the next action becomes available, our controller will fire the appropriate transition to start a new action and release the resource used by the old action.

$$F_r = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{array} \begin{array}{c} \begin{array}{ccccccc} ade & b & c & f & g1 & g2 & h \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$S_r = \begin{array}{c} \\ ade \\ b \\ c \\ f \\ g1 \\ g2 \\ h \end{array} \begin{array}{c} \begin{array}{cccccccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & x_{12} \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Figure 18: Final S_r and F_r Matrices

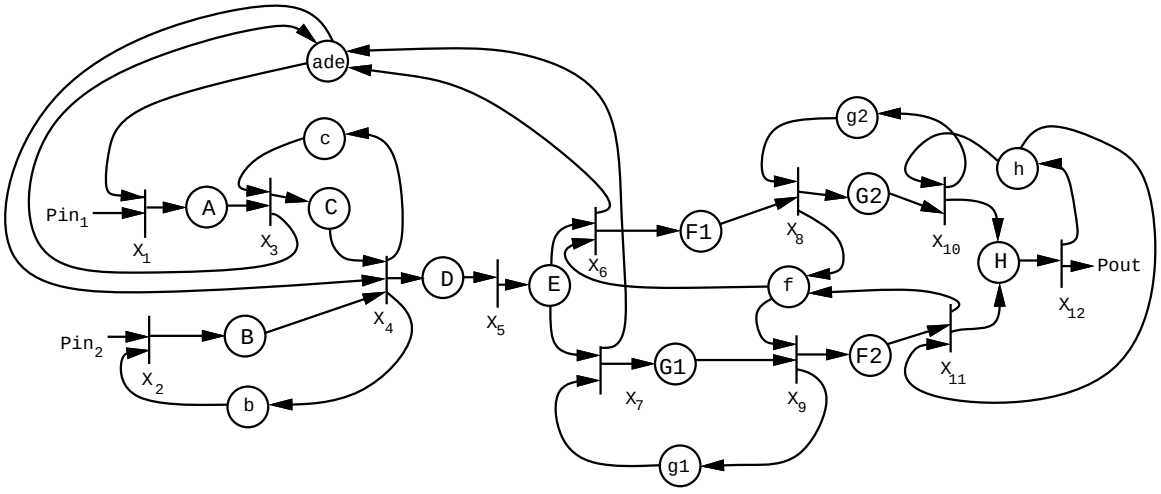


Figure 19: Our Completed Petri Net

5 EXECUTING MANUFACTURING PLANS

So far, we have focused on the operators used by our planner and the completed plans formed by our planner.

Actually, the planner is only one component of a complete system. In this section, we describe the other components of our manufacturing cell. Our planner provides a partially-ordered list of manufacturing steps in the F_v and S_v matrices. Assigned resources are described in the F_r and S_r matrices. A dispatcher uses the matrices, along with real-time status information, to decide when to commence the next manufacturing operation. In a Petri-Net, commencing a manufacturing operation corresponds to firing a transition; new resources will be reserved, old resources will be released, and the cell will switch tasks.

5.1 Introduction to Manufacturing

Flexible manufacturing systems have four major components (Buzacott & Yao, 1986):

1. A set of machines or workstations.
2. An automated material handling system allowing flexible job routing.
3. Distributed buffer storage sites.
4. A computer-based *supervisory controller* which monitors the status of jobs and directs part and machine job selections.

Flexible manufacturing systems can produce different products by varying their supervisory controller. If the same resource is used for more than one task, then the controller must perform *dispatching* to determine the order in which the resource performs the tasks. In addition to optimizing some performance measure (such as maximizing throughput or machine utilization), the dispatcher must avoid *deadlock*. In a deadlocked system, some resource is being held pending an event which will never occur. Thus, the deadlocked resource will never again become available.

5.2 Coordination

Figure 20 shows our complete manufacturing system. The planner performs high-level job sequencing; it belongs to the Organization level of Saridis's hierarchy (Figure 1). The resource assignment module decides which actual resources will be used to implement the plan and forms F_a. These two modules provide the controller with four task matrices (F_v, S_v, F_r, S_r).

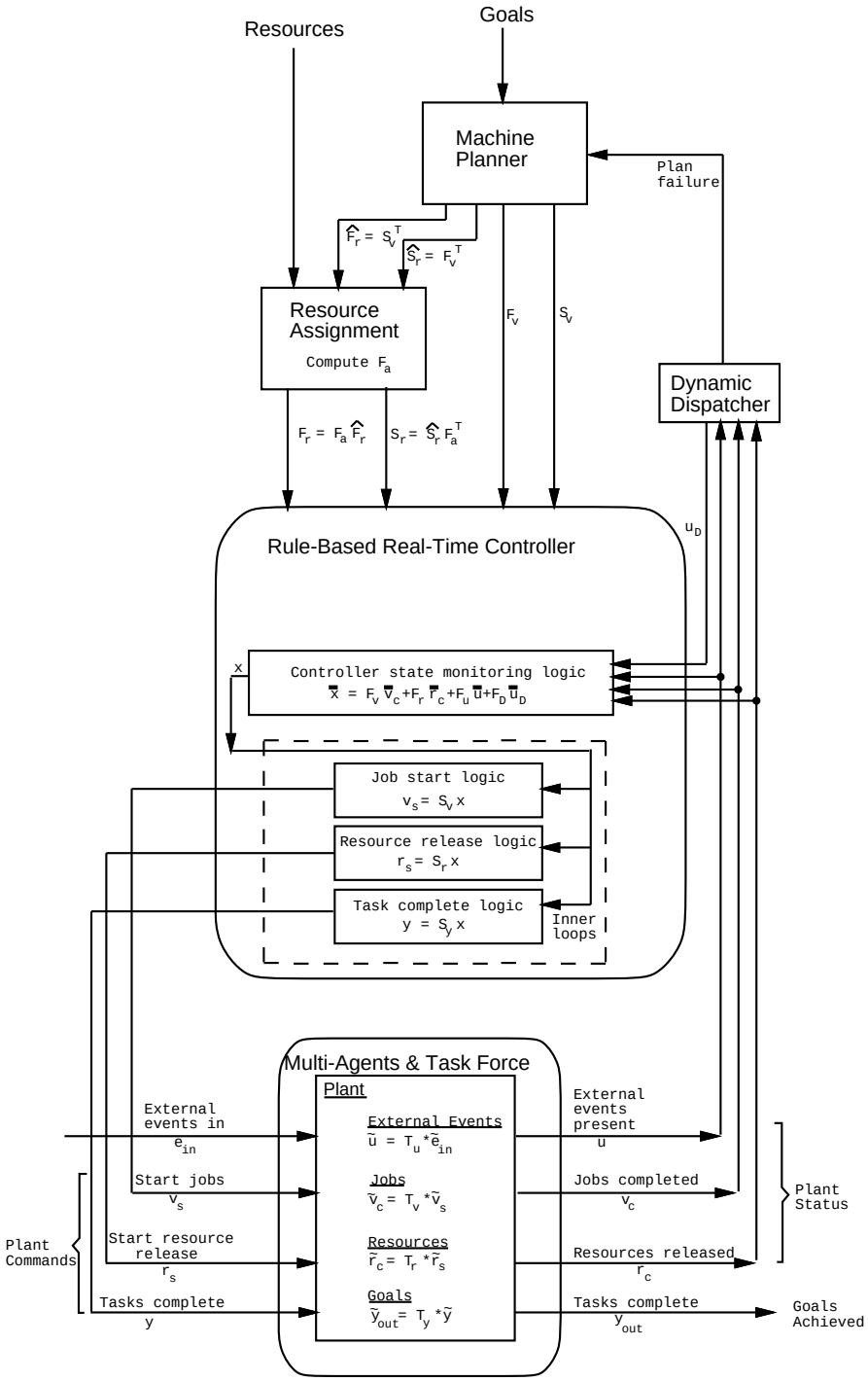


Figure 20: Matrix-based Supervisory Controller

Our supervisory controller belongs to the “Coordination Level” of Figure 1. The rule-based controller is described by several equations:

Matrix Controller State Equation

$$\bar{x} = F_v \bar{v}_c + F_r \bar{r}_c + F_u \bar{u} + F_D \bar{u}_D \quad (1)$$

Job Start Equation

$$v_s = S_v x \quad (2)$$

Resource Release Equation

$$r_s = S_r x \quad (3)$$

Task Complete Equation

$$y = S_y x \quad (4)$$

These matrix operations use the *or/and algebra*, where "+" denotes logical or and "×" denotes logical and. The overbar in (1) denotes logical negation (e.g. so that jobs complete are denoted by 0). Thus, equation (1) amounts to and operations (for assignment of resources) while equations (2)-(4) amount to or operations (for release of resources). In Petri-Net parlance, the controller state equation (1) is responsible for firing the transitions; the controller state vector x is isomorphic to the vector of Petri-Net transitions. These are *logical* equations, and so form a *rule base*. The coefficient matrices are sparse, so that real-time computations are easy even for large interconnected systems; the rules can be fired using efficient algorithms such as the *Rete algorithm*.

The matrix formulation allows: (1) computer simulation and (2) computer implementation of the controller on an actual work-cell. Input u represents raw parts entering the cell and y represents completed tasks or products leaving the cell. The controller, shown in Figure 20, observes the *status outputs* of the system or work-cell, namely, job vector v_c, whose entries of '1' represent 'completed jobs' and resource vector r_c, whose entries of '1' represent 'resources currently available'. The vector [v r] is isomorphic to the Petri-Net place vector. (Subscript 'c' denotes 'complete' or 'available' status, while subscript 's' denotes 'start' or 'release' commands.) The *controller state equation* (1) checks the conditions required for performing the next jobs in the system. Based on these conditions, stored in the logical vector x, the job start equation (2) computes which jobs are activated and may be started, and the resource release equation (3) computes which resources should be

released (due to completed jobs). Then, the controller sends *commands* to the system, namely, vector v_s, whose '1' entries denote which jobs are to be started, and vector r_s, whose '1' entries denote which resources are to be released. Completed tasks are given by (4).

The matrix-based logical controller has the *multiloop feedback control* structure shown in Figure 20, with *inner loops* where there are no shared resources, and *outer loops* containing shared resources where dispatching and/or routing decisions are needed to determine u_D, which is a *conflict resolution input* that selects which jobs to initiate when there are simultaneous requests involving shared resources. This *dispatching* input is selected in higher-level control loops using priority assignment techniques (e.g. (Panwalker & Iskander 1977)) in accordance with prescribed performance objectives such as minimum resource idle time, task priority orderings, task due dates, minimum time of task accomplishment, and so on as prescribed by the user.

The T_u, T_v, T_r, and T_y matrices shown in Figure 20 describe the job durations and resource set-up times. These matrices are described in (Tacconi & Lewis, 1997). It is easy to show that a Petri Net description can be derived from the matrices. In fact, we define the *activity completion matrix* F and the *activity start matrix* S as

$$F = \begin{bmatrix} F_v & F_r \end{bmatrix},\ S = \begin{bmatrix} S_v \\ S_r \end{bmatrix}. \tag{5}$$

We define transition vector X as the set of elements of controller state vector x, and place vector A (activities) as the set of elements of the job and resource vectors v and r. Then (A,X,F,S^T) is a Petri-Net. The new matrix model overcomes one of the prime deficiencies of Petri-Net theory-- it provides *rigorous computational techniques for dynamic systems*. It has been shown (in (Tacconi & Lewis, 1997) and (Lewis et al., 1995)) that one may compute directly in terms of F_v, F_r, S_v, S_r all the resource loops (p-invariants), all the circular waits of resources, and give algorithms for dispatching shared resources with guaranteed avoidance of deadlock.

6 COMBINING ALTERNATE PLANS

So far, we have shown that an assembly tree can be converted into a set of HTN operators and that a machine planner can use these operators to form a plan which in turn can be converted into four matrices usable by our supervisory controller. In this section, we show how multiple assembly trees can result in more than one possible plan; these plans can be combined into a single set of matrices. We present a polynomial-time algorithm for combining the plans into a single matrix representation.

6.1 Forming more than one Plan

In (Gracanin, et al., 1994), Gracanin uses the two assembly trees shown in in Figure 21. Gracanin combines the two trees using a parameterized Petri-net. We incorporate the alternatives into a matrix notation, which is computationally easier to manipulate.

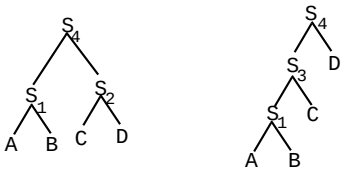


Figure 21: Alternate Assembly Trees

The assembly trees correspond to the HTN operators shown in Figure 22. The planner has more than one sequence of steps which can solve the goal corresponding to S_4. If we can determine in advance the conditions under which one sequence of steps will be “better”, then we can encode this information into the planner (possibly adding subgoals not shown in the assembly tree) and allow our planner to determine the best plan based on relatively static information. Alternatively, we can have the planner generate all possible plans and allow a lower-level dispatcher to switch between them based on real-time (dynamic) conditions.

```
(actschema Build-s4-with-s2
       :todo (Assembled s4)
       :expansion ( (step1 :goal (Assembled s1))
                    (step2 :goal (Assembled s2))
                    (step3 :primitive (Attach s1 s2)))
       :effects ( (step3 :assert (Assembled s4)))
       :orderings ((step1 -> step3) (step2 -> step3)))

(actschema Build-s4-with-D
       :todo (Assembled s4)
       :expansion ( (step1 :goal (Assembled s3))
                    (step2 :goal (Assembled D))
                    (step3 :primitive (Attach s3 D)))
       :effects ( (step3 :assert (Assembled s4)))
       :orderings ((step1 -> step3) (step2 -> step3)))

(actschema Build-s1
       :todo (Assembled s1)
       :expansion ( (step1 :goal (Assembled A))
                    (step2 :goal (Assembled B))
                    (step3 :primitive (Attach A B)))
       :effects ( (step3 :assert (Assembled s1)))
       :orderings ((step1 -> step3) (step2 -> step3)))

(actschema Build-s2
       :todo (Assembled s2)
       :expansion ( (step1 :goal (Assembled C))
                    (step2 :goal (Assembled D))
                    (step3 :primitive (Attach C D)))
       :effects ( (step3 :assert (Assembled s2)))
       :orderings ((step1 -> step3) (step2 -> step3)))

(actschema Build-s3
       :todo (Assembled s3)
       :expansion ( (step1 :goal (Assembled s1))
                    (step2 :goal (Assembled C))
                    (step3 :primitive (Attach s1 C)))
       :effects ( (step3 :assert (Assembled s3)))
       :orderings ((step1 -> step3) (step2 -> step3)))

(actschema Prepare-A
       :todo (Assembled A)
       :expansion ( (step1 :primitve (Collect A)))
       :effects ( (step1 :assert (Assembled A))))

(actschema Prepare-B
       :todo (Assembled B)
       :expansion ( (step1 :primitve (Collect B)))
       :effects ( (step1 :assert (Assembled B))))

(actschema Prepare-C
       :todo (Assembled C)
       :expansion ( (step1 :primitive (Collect C)))
       :effects ( (step1 :assert (Assembled C))))

(actschema Prepare-D
       :todo (Assembled D)
       :expansion ( (step1 :primitive (Collect D)))
       :effects ( (step1 :assert (Assembled D))))
```

Figure 22: HTN Operators for Figure 21

Figure 23 shows the two possible plans for assembling an S_4 part. As described in Section 4, we can use each plan to form from a Petri-Net and a set of matrices F_v and S_v. Figure 24 shows the two Petri Nets for this problem, and Figure 25 shows the corresponding pairs of matrices.

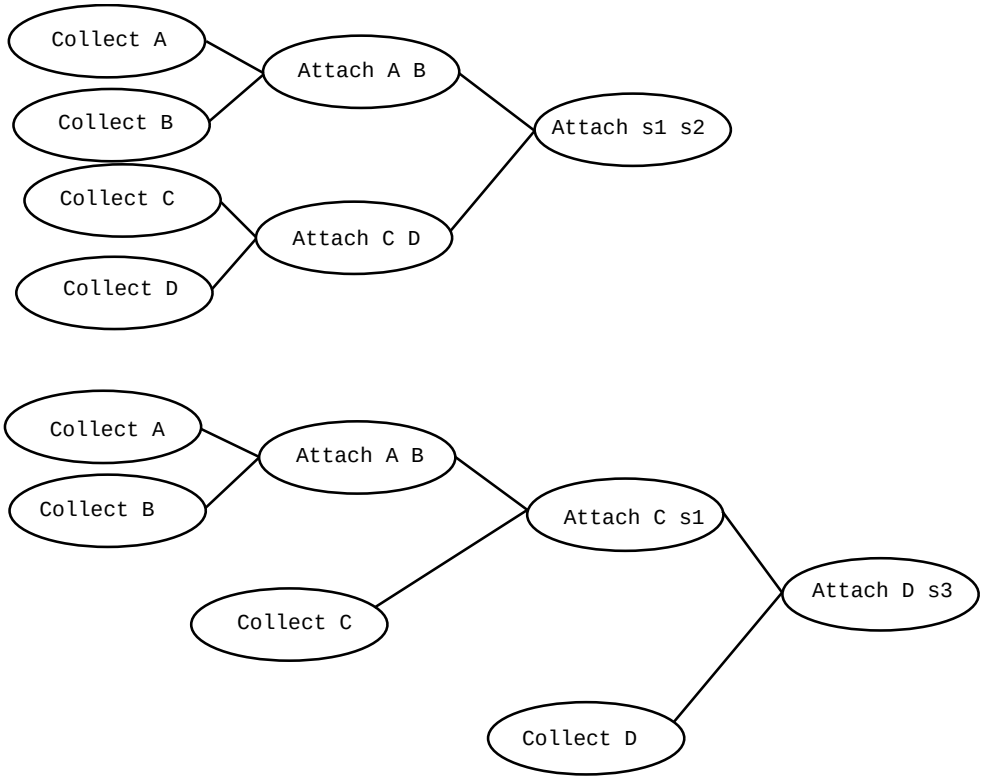


Figure 23: Multiple Plans for Assembling an S_4 Part

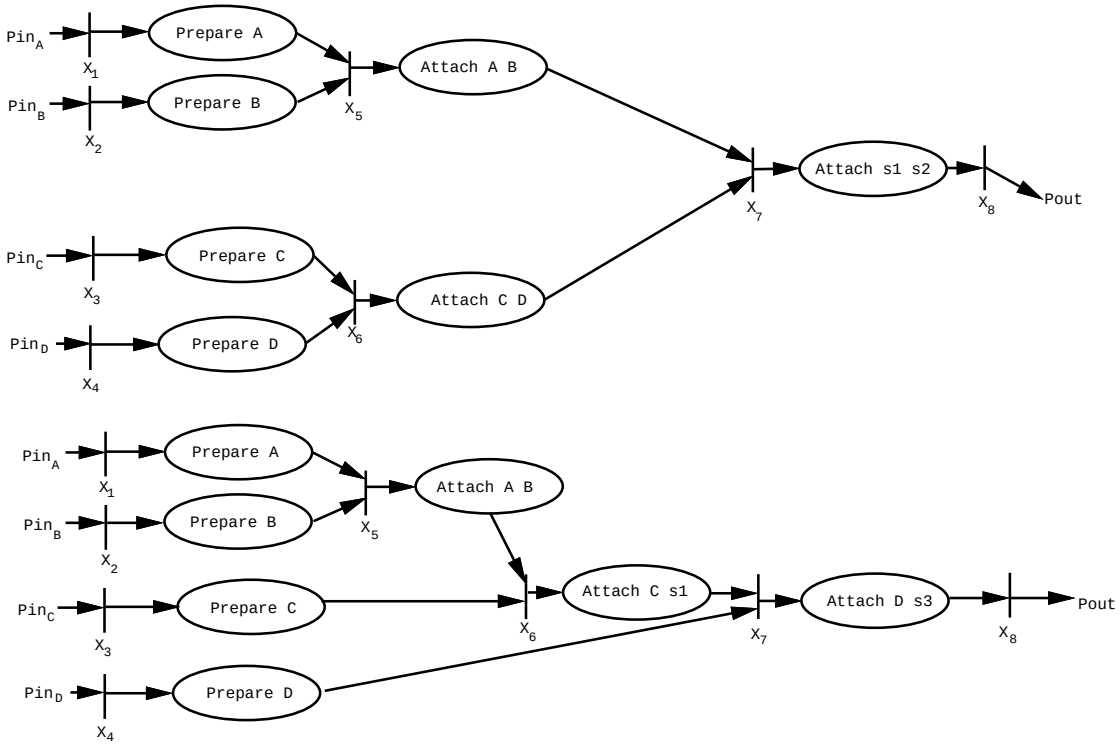


Figure 24: Multiple Petri Nets for Assembling an S_4 Part

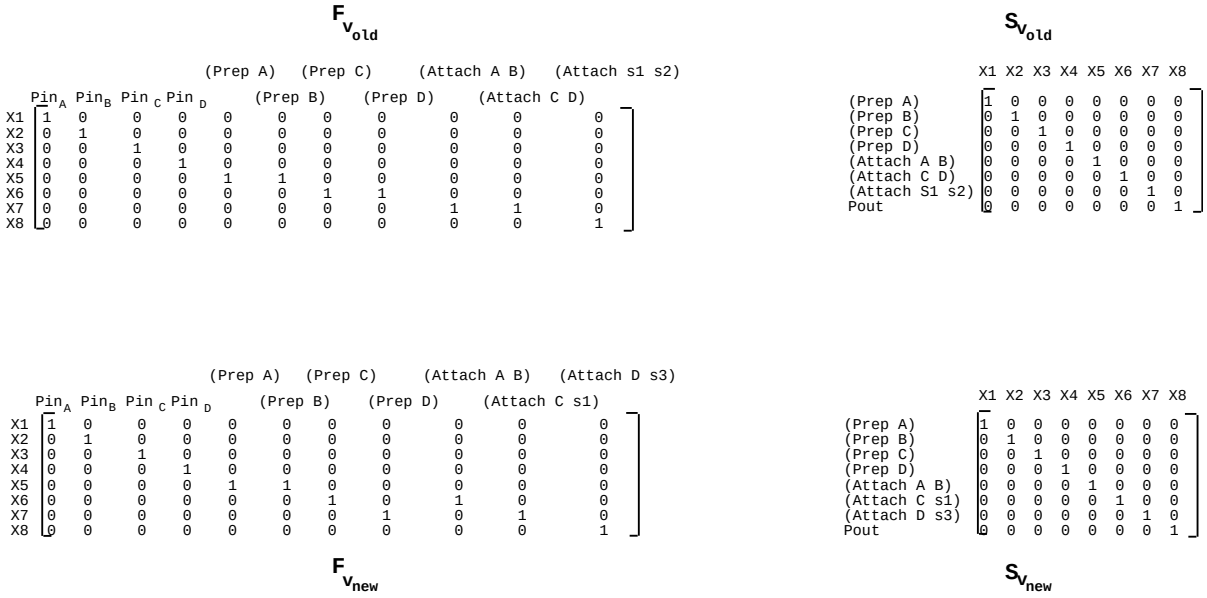


Figure 25: Multiple Matrices for Assembling an S_4 Part

6.2 Combining Multiple Plans

We will incorporate the two matrix pairs in Figure 25 into a single matrix pair in polynomial time. F_{v_c} and S_{v_c} will hold our combined job sequence. We initialize F_{v_c} to the F_v matrix of our first plan and we initialize F_{v_c} to the S_v matrix of our first plan. For each additional plan, we use our algorithm to incorporate additional choices into the matrices; after we have finished, F_{v_c} and F_{v_c} contain information on all possible plans. Once we form F_{v_c} and F_{v_c}, we can assign resources as described in Section 4 and allow our controller (Section 5) to decide in real-time which method it will use to produce the desired part. Our algorithm makes two assumptions:

1. Parts with the same label on two different assembly trees represent the same part.

2. The uses of a part are independent of the method used to construct the part.

Without assumption 1, made implicitly in (Gracanin et al., 1994), it would be extremely difficult to combine multiple assembly trees at all. Assumption 2 means that if two nodes in different plans have the same node labels, then they are identical nodes. Without this assumption, combining plans becomes an instance of subgraph isomorphism, an NP-Complete problem (Garey & Johnson, 1979). Assumption 2 is reasonable for manufacturing operators; the results of the action "Drill C to produce D" do not normally depend on which

previous actions we used to produce the C part.

Figure 26 shows our algorithm. First, we associate each place in our new Petri-Net with a (possibly new) place in our old Petri-Net. Second, we examine each transition in the new Petri-Net and decide if that transition should be added to our old Petri-Net.

```
CombineMatrix(F_v_old, S_v_old, F_v_new, S_v_new)

  For each place in F_v_new
    Does the place exist in F_v_old?
    If so:
      Associate the place with the corresponding place of F_v_old
      Associate the corresponding place of S_v_new with S_v_old
    If not:
      Create a new place in F_v_old and S_v_old
      Associate the place in F_v_new with the new place in F_v_old
      Associate the corresponding place of S_v_new with the new place in S_v_old

  For each transition in F_v_new, S_v_new
    Does the corresponding transition exist in F_v_old, S_v_old?
    If so:
      (do nothing)
    If not:
      Create the transition
```

Figure 26: Combining Multiple Plans

Rows in S_v and columns of F_v correspond to primitive actions, or to places in a Petri-Net. Our algorithm begins by associating each place in our new matrix pair with a (possibly new) place in our accumulating matrix pair. If the current place has a label identical to a place in our accumulating matrices, then (by Assumption 2), the current place is equivalent to the accumulating place with the matching label. If no place label in our accumulating matrix pair matches the current place, we create a new place in our accumulating matrix pair; the current place is equivalent to this new place. Initially, new places are formed without any incoming or outgoing transitions; the newly created rows of S_v and columns of F_v are initialized to zero. Figure 27 shows the associations formed by this phase of our algorithm. Rows and columns in *italics* are newly added places.

After we associate each place in our new matrix pair with a (possibly new) place in our accumulating pair, we examine each transitions in our new matrix pair. Our algorithm examines the accumulated plan to

decide whether the current transition already exists *based on the associated nodes* of our accumulated plan. Our algorithm judges transitions based on the node labels of the places the transitions link rather than on the place

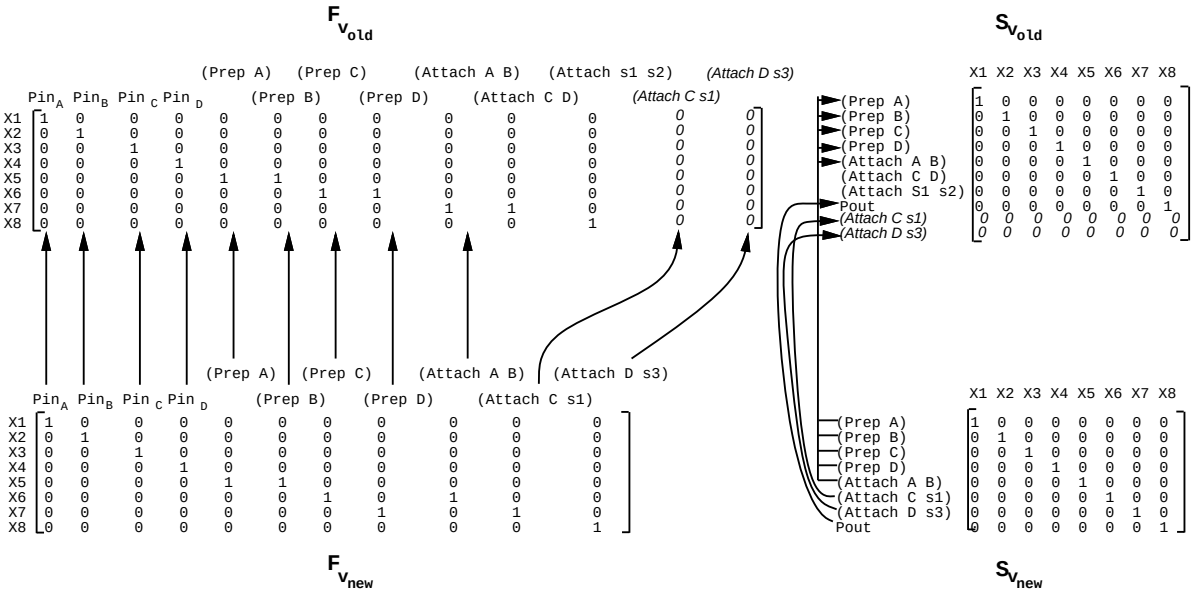


Figure 27: Intermediate Results: Combining Places from Multiple Plans

numbering used by our new matrices. For example, transition X_4 of the new matrix set links P_{in_D} and (Prepare D). Our accumulating matrix set has an existing link (by coincidence also X_4) which links its copy of P_{in_D} and (Prepare D), so our algorithm does nothing for this transition. In contrast, transition X_7 of the new matrix set links (Attach A B) and (Attach C s1) to (Attach D s3). No existing transition in our accumulating set makes this connection, so we create a new transition X_{10} incorporating this link.

Figure 28 shows the final F_{v_c} and F_{v_c} matrices. Transitions in the new matrix set which do not have corresponding transitions in the accumulating set are marked with a *, and the newly created transitions are in *italic type*. The single set of matrices $F_{v_{\text{old}}}$ and $S_{v_{\text{old}}}$ now represent two different methods of producing S_4 parts. Figure 29 shows the combined Petri Net.

6.3 Complexity Analysis

Suppose we have m plans and, after converting the plans to Petri-Nets, each plan has an average of n steps. The number of transitions is $\theta(n)$.

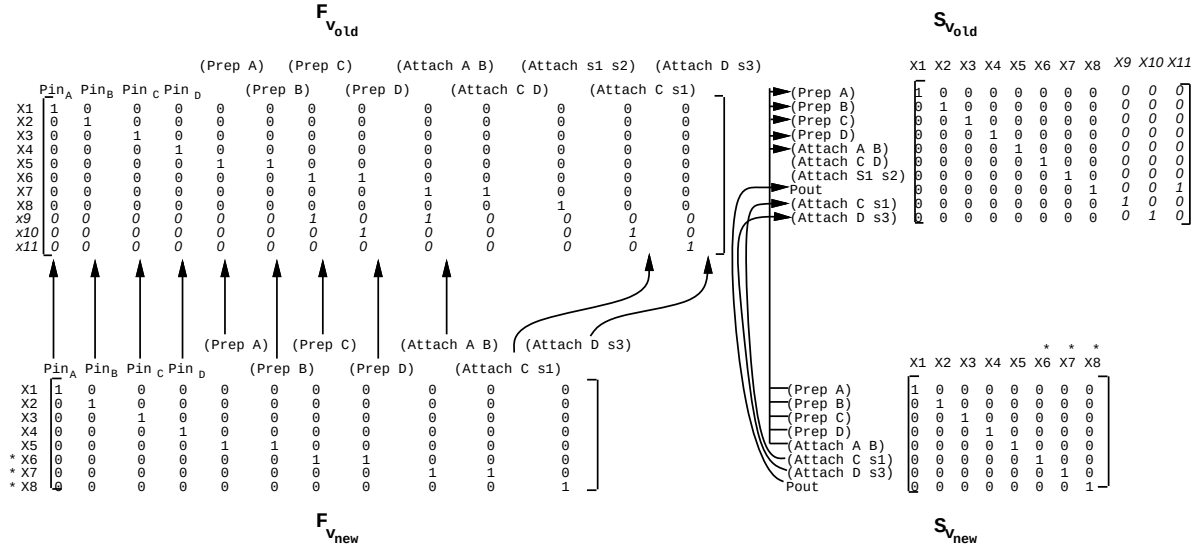


Figure 28: Final Results: Combining Transitions from Multiple Plans

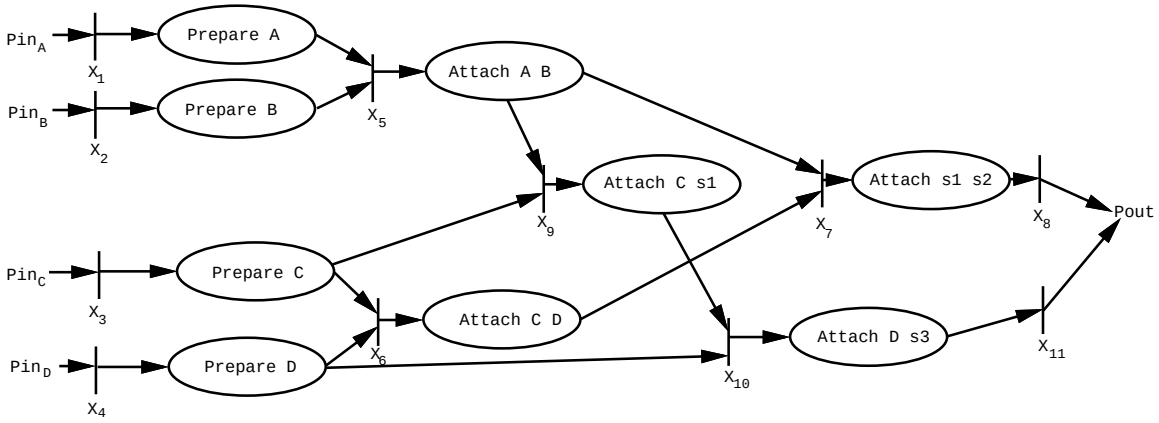


Figure 29: Combined Petri Net

Our algorithm is run m-1 times. The algorithm will trace through n places and $\theta(n)$ transitions. We can create a place or transition in amortized constant time. The amount of time to decide whether a place or transition exits depends on the number of places/transitions we have accumulated so far.

One (unlikely) possibility is that each plan is completely different from every other plan and thus our

final matrix will accumulate $O(m*n)$ places. In this case (assuming linear-time search), it will take $O(m*n)$ time to decide whether a given place or transition exists in our matrix, for a total of $m * n * O(m*n)$ or $O(m^2 * n^2)$ time.

A more likely possibility is that most plans are nearly identical and that a given pass through our algorithm adds only a constant number of places, giving a final total of $O(n+m)$ places. This gives (still assuming linear-time search) $O(m+n)$ time to decide whether or not a given place or transition exists in our matrix, for a total of $m * n * O(m+n) = O(n\ m^2 + m\ n^2)$ time.

6.4 Converting Matrices into Plans

Once we have combined our matrices into a single F_v and S_v matrix pair, we can convert the matrices back into a more traditional plan representation. The columns of F_v and the rows of S_v (the places of a Petri-Net) correspond to primitive steps or plan nodes. The rows of F_v and the columns of S_v (the transitions of a Petri-Net) correspond to ordering constraints on the primitive steps, or links between plan nodes. Using this correspondence, we can convert our combined matrix set (presented as a Petri-Net in Figure 29) into the plan shown in Figure30. Dashed lines represent alternatives; solid lines represent ordinary temporal constraints.

In Section 4, we described how partially-ordered plans could be represented in matrix form as a choice between different totally-ordered alternatives. In simple cases, we can recombine the alternatives into traditional planning notation, but after incorporating other plans into a single representation, the combined plan may be too complex to automatically reconvert a choice of totally ordered plan segments into the traditional partial-order notation.

Notice that because of the correspondance between places and primitive steps and between transitions and plan orderings, future planners may use our algorithm for combining plans without commiting to our matrix notation.

7 CONCLUSION

In this paper we have shown how machine planners can be integrated with real-time intelligent control systems. Planners can use existing documentation (assembly trees) to form their operators. Plan operators can represent flow-lines, assembly operations, and routing choices. Our completed plan can be converted into a set of matrices which can be executed by a rule-based controller.

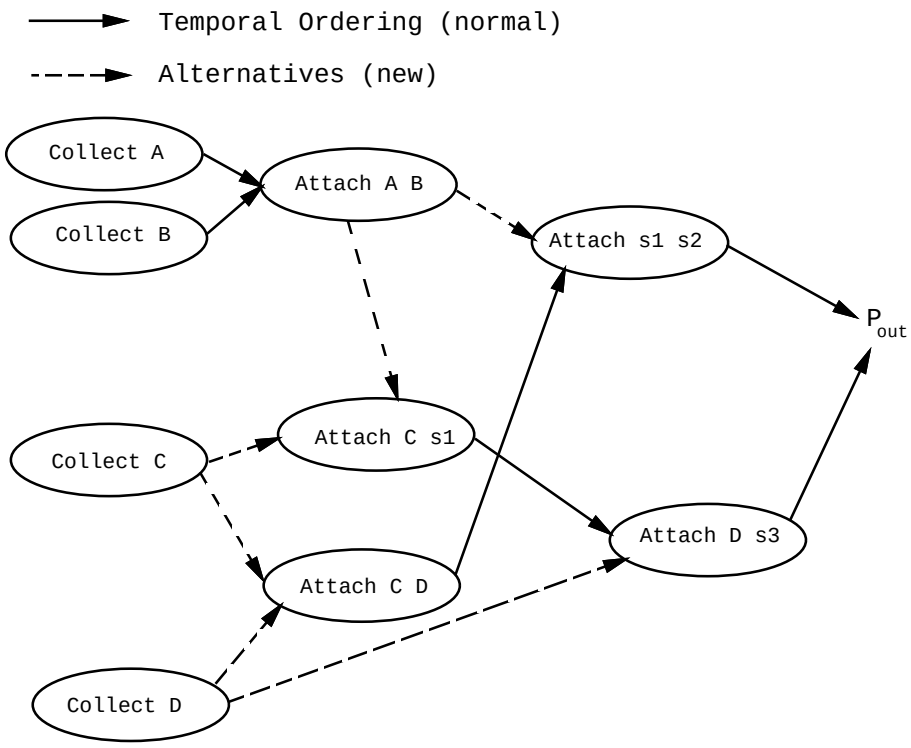


Figure 30: Combined Plan

We have presented a polynomial-time algorithm for combining multiple plans. Our method of combining multiple plans into a single framework simplifies re-planning, and allows an agent to choose in real time which of several alternatives will best accomplish a goal.

References

Baker, K. (1974). *Introduction to Sequencing and Scheduling*, John Wiley and Sons, New York.

Buzacott, J. & Yao, D. (1986). Flexible manufacturing systems: A review of analytical models, *Management Science* **32**(7): 890-905.

Desrochers, A. A. (1990). *Modeling and Control of Automated Manufacturing Systems*, IEEE Computer Society Press.

Fikes, R. & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence* **2**:189-208.

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W H Freeman & Co.

Gracanin, D., Srinivasan, P. & Valavanis, K. (1994). Parameterized petri nets and their application to planning and coordination in intelligent systems, *IEEE Transactions on Systems, Man, and Cybernetics* **24**(10): 1483-1497.

Jeng, M. D. & DiCesare, F. (1992). A synthesis method for petri net modeling of automated manufacturing systems with shared resources, *Proceedings of IEEE Conference on Decision and Control*, pp. 1184-1189.

Kusiak, A. (1992). Intelligent scheduling of automated machining systems, *Intelligent Design and Manufacturing*, New York: Wiley.

Lewis, F., Huang, H., Fierro, R. & Tacconi, D. (1995). Real-time task planning, resource allocation, and deadlock avoidance, *Proceedings of Workshop on Architectures for Semiotic Modeling*, IEEE International Symposium on Intelligent Control, Monterey, pp. 347-355.

Minton, S. et al. (1987). Acquiring effective search control rules: Explanation-based learning in the prodigy system, *Proceedings of Machine Learning Conference* pp. 122-133.

Murata, T., Komoda, N., Matsumoto, K. & Haruna, K. (1986). A petri net-based controller for flexible and

maintainable sequence control and its applications in factory automation, *IEEE Transactions on Industrial Electronics* **IE-33**(1).

Panwalker, S. & Iskander, W. (1977) A survey of scheduling rules, *Operations Research* **26**(1): 46-61.

Sacerdoti, E. (1975). The nonlinear nature of plans, *IJCAI* pp. 206-214.

Saridis, G. N. (1983). Intelligent robotic control,*IEEE Transactions on Automatic Control* **28**(4): 547-557.

Tacconi, D. & Lewis, F. (1997). A new matrix model for discrete event systems: Application to simulation, *IEEE Control Systems Magazine.*

Tate, A. (1977). Generating project networks, *International Joint Conference on Artificial Intelligence* pp. 888-893.

Wilkins, D. (1984). Domain-independent planning: Representation and plan generation, *Artificial Intelligence* **22**(3): 269-301.

Wilkins, D. E. (1994). Comparitive analysis of AI planning systems: A report on the AAAI workshop, *AI Magazine* **15**(4): 69-70.

Wolter, J., Chakrabarty, S., & Tsao, J. (1992). Methods of knowledge representation for assembly planning, *Proceedings of NSF Design and Manufacturing Systems Conference* pp. 463-468.

Zhou, M. C. & DiCesare, F. *Petri-Net Synthesis for Discrete Event Control of Manufacturing Systems*, Kluwer, Boston.