Machine Planning for Manufacturing: Dynamic Resource Allocation and On-line Supervisory Control*

Billy Harris† Frank Lewis‡ Diane J. Cook§

February 5, 1998

Submitted to Journal of Intelligent Manufacturing

Abstract

In this paper, we provide tools for integrating machine planning and manufacturing. Specifically, we show how assembly trees can be coded into operators for machine planners and how machine planners can represent flow-lines, assembly, and job-shop choices. We provide a polynomial-time algorithm for succinctly combining multiple plans; the resulting plan can be expressed as four matrices that are equivalent

*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

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

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

to a Petri Net. We also provide a dynamic supervisory controller which can execute a single plan or switch between multiple plans as real-time conditions change.

1 Introduction

Machine planning is a very active subfield of Artificial Intelligence. Planning researchers hope to produce an agent which does not need a human to provide step-by-step direction of the agent's actions. Instead, the agent uses its own knowledge about a particular domain to solve the problems it encounters.

While machine planners have been used in a few "real world" applications (Wilkins 1990, Currie & Tate 1991, Russel & Norvig 1995, Tate 1977), machine planning has not been widely adopted outside the research lab. One difficulty lies in using the planner as one component of a real-time manufacturing system. Ideally, a manufacturer should be able to use existing documentation—such as an assembly tree—to directly program the planner. Ideally, a planner should impose only the minimum constraints needed to solve a particular problem; it should allow the partial ordering of jobs and work in conjunction with existing techniques for resource assignment. Finally, an ideal planner should output a structure which could be used directly by schedulers and dispatchers to control the manufacturing process on line.

This paper presents an on-line method of using machine planners to achieve this ideal. In particular, we demonstrate how to:

- use a manufacturing bill of materials or assembly tree to form

the operators for a machine planner,

- use the idea of "generic resources" to separate job sequencing from resource assignment,
- convert the planner's output to a matrix representation downloadable to a supervisory rule-based controller, which can sequence jobs and assign resources in real time depending on the actual system's status, and
- dynamically switch between multiple stored plans in real time as conditions and available resources change.

In addition, we illustrate how traditional manufacturing concepts such as flow-lines and job shops can be represented and manipulated by our machine planning system. Our notation can compactly represent alternate plans. We give an algorithm for finding a minimal matrix representation for multiple plans. Finally, we describe a matrix-based supervisory controller which can switch between strategies in real-time.

Section 2 describes intelligent control architectures and describes the representation used for assembly trees. Section 3 provides an overview of machine planning, with an emphasis on HTN planning. Section 3 also reviews Petri Nets. In Section 4, we demonstrate how assembly trees can be easily converted into plan operators. Section

5 shows how flow-lines, assembly, and job-shop scheduling appear in assembly trees, in HTN operators and in the resulting plan. In Section 6, we detail our matrix representation for task sequencing, introduce the notion of generic resources, and add resource assignment to our plan notation. Section 7 describes how alternate plans can be combined into a single system and presents a polynomial algorithm for finding a minimal representation for multiple plans. Finally, Section 8 shows that the matrices produced by the planner can be used in a decision-making supervisory controller that operates the manufacturing process on-line in real time, sequencing the jobs and assigning available resources as the status of the process changes.

2 Manufacturing Background

In this section, we describe how our planning system can serve as one component of an intelligent control system. We define manufacturing terms and describe manufacturing assembly trees, which we will use to form the operators for our planner.

2.1 Introduction to Manufacturing

To meet competition in a global marketplace and provide flexible manufacturing in a high-mix low-volume manufacturing environment,

manufacturing systems have moved away from fixed-hardware sequential assembly lines with dedicated workstations. The trend for several years has been toward *flexible manufacturing systems* (FMS), which 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 routing and machine job selections.

By selecting different supervisory control strategies, the same FMS can perform different functions to produce different products. Types of manufacturing systems are defined according to the information and part flow protocols in the FMS. In the general *job shop* neither the sequence of jobs nor the assignment of resources to jobs is fixed. The effect of a job shop is that part *routing decisions* must be made during processing; a router must decide the order in which jobs are performed for a particular part. In the *flow line with deterministic routing*, the sequence of jobs for each part type is fixed and the assignment of resources to the jobs is fixed. A flow line may have assembly operations in which two or more parts are combined to yield one part or

subassembly. The result of a flow line is that each part of the same type visits the resource pools in the same sequence, although different part types may have different sequences. If the same resource is used for more than one task, then a controller must perform *dispatching* to determine on which part the resource will operate at a given time. 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.

2.2 Planning and Intelligent Control

While many different structures have been proposed for "intelligent control architectures" (Antsaklis & Passino 1992), these structures are actually very similar; the major differences are due to focusing on different aspects of intelligent control or different levels of abstraction. We will use a very general architecture shown in Figure 1. This architecture, based on work by Saridis, focuses on the principle of decreasing precision with increasing abstraction.

Components located in the organization level function as managers; these components schedule tasks, perform task decomposition, and determine the resources needed for each task. Our work on plan-

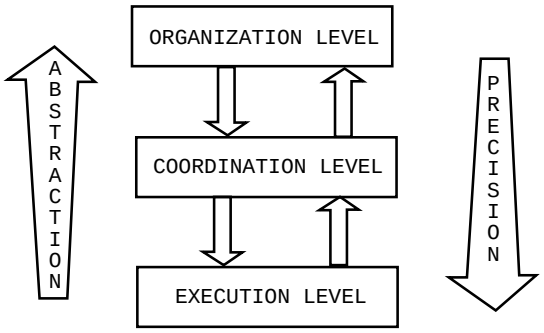


Figure 1: Three-Level Intelligent Control Architecture

ning and resource assignment fits into this part of Saridis's hierarchy.

Components in the Coordination Level perform the prescribed job sequencing, supervise and coordinate the work-cell agents or resources, and perform dispatching and conflict resolution to manage shared resources. Our supervisory controller performs these functions. The *agents* or *resources* of the work-cell include robot manipulators, grippers and tools, conveyors and part feeders, sensors (e.g. cameras), mobile robots, and so on.

The Execution Level contains a closed-loop controller for each agent that is responsible for the real-time performance of that resource, including trajectory generation, motion and force feedback servo-level control, and so on. Some permanent built-in motion sequencing may be included (e.g. stop robot motion prior to opening the gripper).

Each level of the hierarchical IC architecture may have several components; for example, the Execution Level has one real-time controller per work-cell agent. A supervisory controller in the Coordination Level may coordinate several agents to sequence the jobs needed for a given task. Each system must sense the current conditions, make decisions, and give commands or status reports to other systems.

Our planner functions at the Organization Level of Saridis's hierarchy. We will show that a task plan is equivalent to four matrices; these matrices may be passed to a supervisory controller in the Coordination level which sequences jobs and assigns resources.

2.3 Assembly Trees

Assembly trees are used in manufacturing to specify a partial ordering of jobs required to complete a finished product. An assembly tree (Wolter, Chakrabarty & Tsao 1992) has a node for each subassembly, and contains information equivalent to the manufacturing bill of materials (BOM) (Baker 1974). An assembly tree or BOM can be considered as a matrix for which entry i, j has a value of 1 if job j is an immediate prerequisite for job i. Neither assembly trees nor BOMs contain any information about the resources needed for the jobs; they contain only product-specific job sequencing information.

We represent assembly trees graphically, with an edge for each

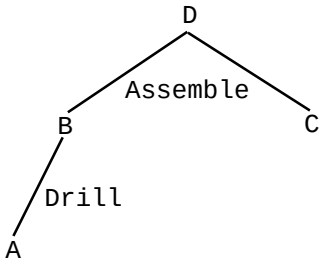


Figure 2: Sample Assembly Tree

manufacturing operation. Nodes in the graph represent parts or sub-assemblies; the type of the part changes as operations are performed. Figure 2 shows a sample assembly tree; notice that by drilling part A, we create part B. Parts C and D can be assembled to form the single part D.

3 Machine Planning Background

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

3.1 Introduction to Machine Planning

To solve problems in a given domain, a machine planner must have a domain description and a description of the specific problem to solve. *Operators*, a crucial component of a domain description, detail how

the agent can affect the environment. Most planners use a variant of the operator representation introduced by Fikes, Hart, and Nilson's STRIPS planner (Fikes & Nilsson 1971).

STRIPS operators have three components: a precondition list, an add list, and a delete list. Each list is composed of first order predicate calculus expressions. The precondition list enumerates the predicates which must be satisfied for the operator to be applicable. The add list indicates which previously false predicates will become true after the operator is applied. The delete list indicates which previously true predicates will no longer hold after the operator is applied. Predicates which are not mentioned in the add list or in the delete list do not change their truth value during the operator's application. 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 allows the robot to pick up blocks. To pick up a

block, the robot must have a free hand, and the block must be clear (have nothing on top of it). After picking up the block, the robot no longer has a free hand and the block is no longer on top of its support. On the other hand, the robot now is carrying a block, and the block's previous support is now clear. In general, there are many ways to divide a domain into operators and many ways to encode each operator.

In addition to a general domain description, planners require a specific problem to solve. Problems are normally represented by an initial state and a goal state. 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.

Researchers have implemented several extensions to the STRIPS paradigm, including automatic learning of search-control rules (Minton, Carbonell, Etzioni, Knoblock & Kuokka 1987), organization of a domain description into different levels of abstraction (Yang & Tenenberg 1990), and generation of plans for multiple agents (von Martial 1992). One extension important for manufacturing is the idea of *partial or-*

der planning. Early planning systems output a step-by-step description of how to solve a problem —that is, they output a totally-ordered sequence of operators. Sacerdoti, however, introduced the ability to produce plans with only a partially-ordered sequence of steps(Sacerdoti 1975). In addition to correctly handling interacting subgoals (Sussman 1973), Sacerdoti's NOAH gives the executor of the plan some flexibility in the exact order the steps are followed. In some cases, two or more steps may be performed simultaneously.

3.2 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):

```
(opschema makeclear
  :todo (cleartop ?x)
  :expansion (
      (step1 :goal (cleartop ?y))
      (step2 :action (puton ?y ?z))
      )
  :orderings ((step1 -> step2))
  :conditions (
```

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

```
      (: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)
  )
```

By using this schema, an HTN planner can discover a plan to uncover a particular block faster than a conventional planner. 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).

3.3 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

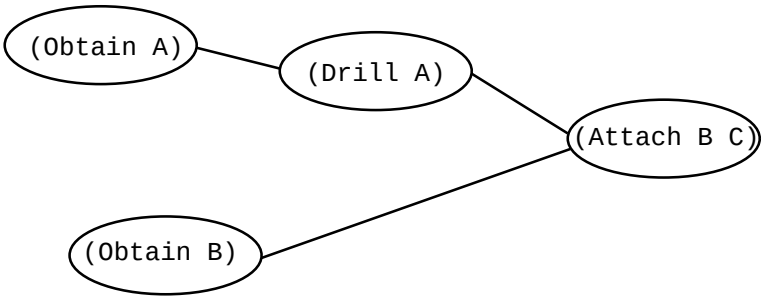


Figure 3: Sample Plan

3 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.

3.4 Petri Nets

Section 5 describes how we convert plans into a set of matrices suitable for a rule-based controller. We begin the process by converting the plan into a Petri net. A Petri net can be represented by several sets (Dessochers 1987):

- P, a set of places. Initially, each place represents a particular action of our plan. Later, we add places to represent 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 or more 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 4 shows a sample Petri Net. This Petri Net shows that 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 completed in either order or simultaneously. When X_4 fires, the action (Attach B C) is performed.

Petri Nets provide a graphical, intuitive representation for the analysis and design of manufacturing systems and supervisory controllers. Manufacturing researchers have studied Petri Nets extensively (Desrochers 1990, Jeng & DiCesare 1992, Murata, Komoda,

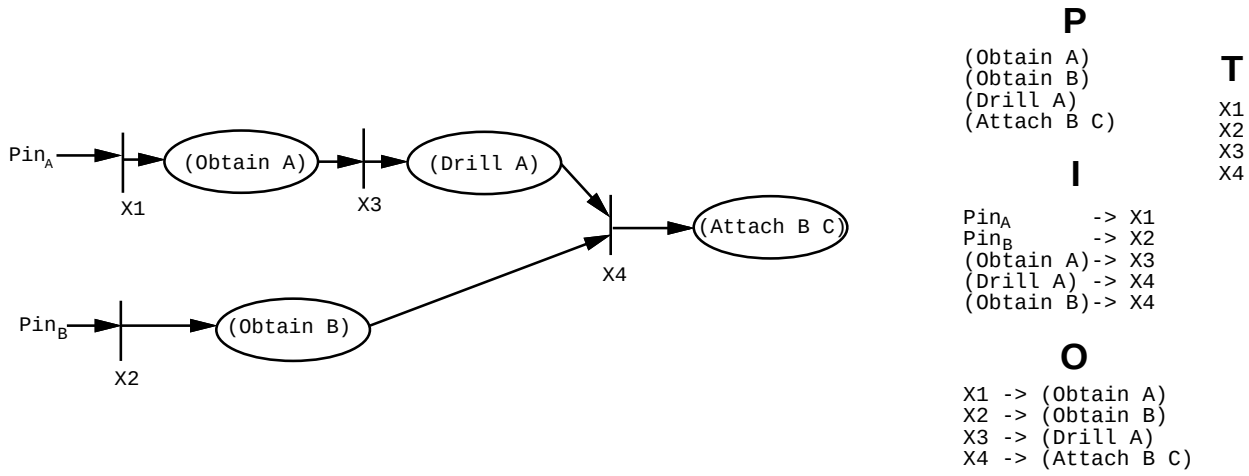


Figure 4: A Sample Petri Net

Matsumoto & Haruna 1986, Zhou & DiCesare 1993); job sequencing controller design, deadlock avoidance, reachability analysis, and system liveness tests have all been investigated. In this paper, we focus on a matrix-based controller derived from an automatically generated plan. The supervisory controller is easy to implement on actual workcells, and can also be used to derive the Petri-Net representation of the workcell so that Petri-Net techniques can be used for analysis and design.

4 Forming HTN Operators from Assembly Trees

HTN operators are defined in terms of actions achieved by other operators. Assembly trees, in which a desired part can be constructed by altering or assembling existing parts, can be easily expressed as HTN operators, as we show in this section. HTN operators, however, are more general than assembly trees; HTN operators can be combined in different ways to represent multiple methods of building a particular part. We choose HTN planners over conventional planners because HTN operators can represent known ordering constraints; this allows the planner to construct a plan faster than an ordinary planner which must search among several possible orderings. However, this method can be easily modified to generate more conventional planning operators. We used UM-Nonlin (Ghosh, Hendler, Kambhampati & Kettler 1992) to implement the operators described in this paper.

Figure 5 shows two sample assembly trees. The left tree indicates that part B can be constructed by drilling part A. The right tree indicates that part Z can be constructed by assembling parts X and Y. Figure 6 lists the HTN operators associated with each tree; Build-B corresponds to the root node of the left assembly tree and Build-Z corresponds to the root node of the right assembly tree.

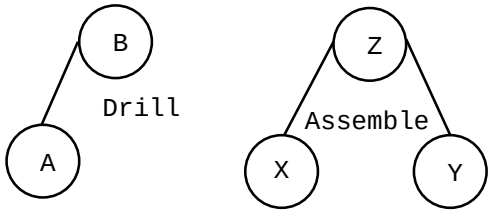


Figure 5: Sample Assembly Trees

An arbitrary node from an assembly tree can be easily converted into an HTN operator. The node's label (naming the part produced) becomes the :todo and :effects of the HTN operator. Each of the node's children becomes a subgoal of the planner (as (Assembled A), (Assembled X), and (Assembled Y) became subgoals for our sample trees). The action needed to produce the part becomes a primitive (directly executable) action for our planner (drilling and assembling for our sample trees). The operator is completed by specifying that the primitive action accomplishes the operator's goal. The ordering constraints are formed by specifying that each subgoal must be accomplished before the primitive action can be performed (for example, (Assembled X) and (Assembled Y) must be accomplished before the step (Attach X Y)).

Thus, each interior node of an assembly tree can be converted into an HTN operator with subgoals. Leaf nodes for an assembly tree, corresponding to incoming parts, can be converted into HTN

```
(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

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

Figure 7: Handling Product-Ins

operators with no subgoals. For example, if part A of Figure 5 is not constructed locally, the HTN operator shown in Figure 7 will "assemble" it without forming subgoals. Figure 8 summarizes our method of converting assembly trees into plan operators.

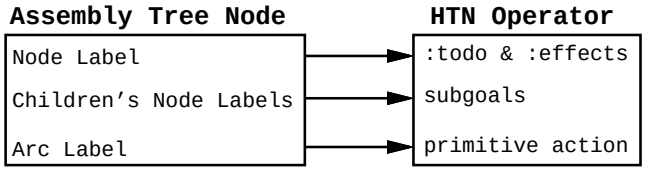


Figure 8: Converting Assembly Trees into HTN Operators

Once we have converted each node of our assembly tree into an HTN operator, we can ask our planner to form plans corresponding to different portions of our assembly trees. For example, suppose we ask our planner to accomplish (Assembled B). Using the operator descriptions in Figure 6 and Figure 7, our planner will tentatively decide that the Build-B operator should be used. This operator has the subgoal (Assembled A). The planner will look for a way to accomplish this goal. One possibility (in fact, the only possibility for this domain) is to use the Prepare-A operator. This operator does not add any new subgoals, so we are finished. Figure 9 shows the resulting two-step plan. The link in the graph specifies that (Puton A Pallet) must be performed before (Drill A). Notice that the plan nodes show the primitive actions performed and not the subgoals considered by the planner.

Plan operators have several advantages over conventional assembly tree representations. It is easier to change one or two operators in isolation than it is to change an entire tree. By adding new operators, we can easily accommodate products which used to be purchased but which are now produced locally. These operator changes will propagate to every product using these parts. In addition, as will be described in Section 7, plan operators can easily represent alternate means of constructing a particular part.

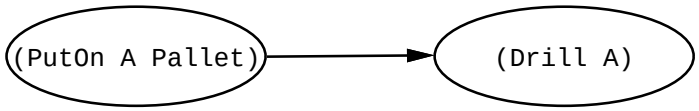


Figure 9: Plan to Assemble B

5 Flow-Lines, Assembly, and Job-Shop Scheduling

Our planner can form plans corresponding to flow-lines, assembly, and job-shop scheduling. This section presents assembly trees and HTN operators (for UM-Nonlin) for each construct and introduces a sample plan resulting from the operators.

5.1 Flow-lines

Manufacturing flow-lines are represented as assembly trees containing a sequence of nodes with only one child each. HTN operators represent flow-lines with operators having only a single sub-goal. The resulting plan contains a totally-ordered sequence of steps. The previous section had an example of a very short flow line; the plan in Figure 9 corresponds to a two-step flow line.

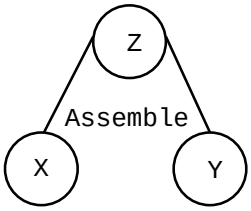
5.2 Assembly

Manufacturing assembly operations are represented as assembly trees containing a node with more than one child. The corresponding HTN operators have more than one subgoal, and the plans are partially ordered instead of totally ordered. In particular, the plan has two or more separate "strands" that eventually merge. Figure 10 expands our earlier assembly example to form a complete plan[2]. The plan indicates that before (Attach X Y) is performed, both (PutOn X Pallet) and (PutOn Y Pallet) must be completed, but the two PutOn steps may be performed in any order, or performed simultaneously.

5.3 Job Shops

There is no standard method of representing general job-shop choices as assembly trees. Figure 11a shows two possible representations; the left tree uses an action which actually combines two steps and the right tree shows two closely related trees explicitly showing the two total orderings. In either case, different HTN operators are used for the different actions; unlike assembly steps, the HTN operators for job-shops work on the same part. The resulting plan thus splits into two different sections and then rejoins. The plan says that part A

[2]Most partial-order planners use artificial "start" and "end" nodes which are not shown.




```
(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))))
```

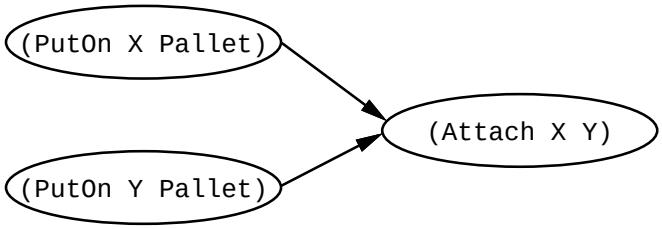


Figure 10: Representing Assembly

must be drilled and sanded before it can be cleaned, but the drill and sand operations may take place in either order (or even simultaneously, given a capable machine).

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 7 that a planner can also consider alternate methods of constructing parts (corresponding to multiple assembly trees). The next section introduces a method of converting our plans into a matrix representation suitable for a controller and demonstrates how our system manages resources.

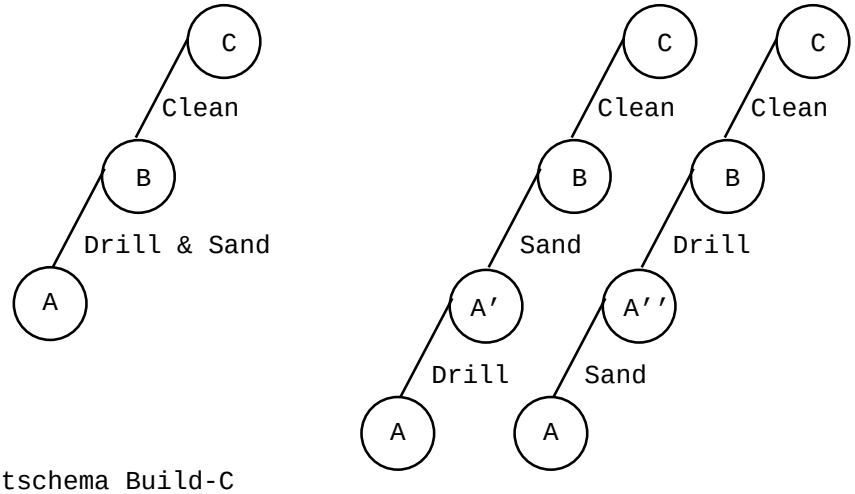
6 Matrix Representation of a Plan

In this section, we show that a plan can be represented by four matrices. Two matrices, F_v and S_v, describe the sequence of jobs to be performed and two others, F_r and S_r, describe the resources that are needed to perform the jobs.

6.1 Matrix Representation of Job Sequences

Consider the plan shown in Figure 12, which includes assembly and routing alternatives. In particular, to complete the plan, an agent

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)

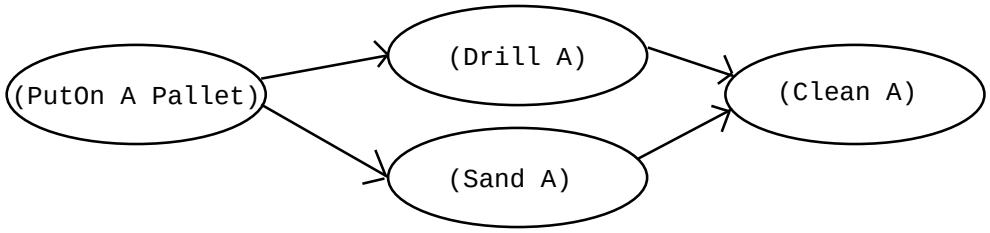


Figure 11: Representing Job-Shops

must perform both step F and step G, but the agent may perform these two steps in either order. This plan can be converted into the Petri-Net shown in Figure 13, in which the possible routing sequences have been explicitly enumerated. 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 7).

The two matrices shown in Figure 14 are equivalent to the Petri-Net shown in Figure 13. 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;

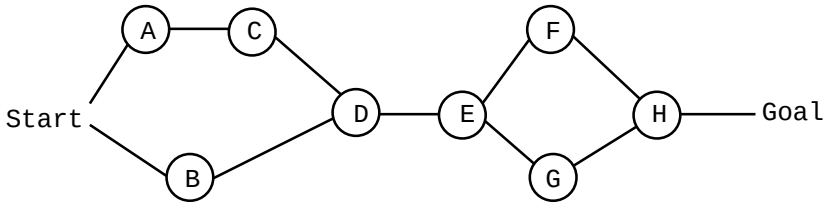


Figure 12: A Sample Plan

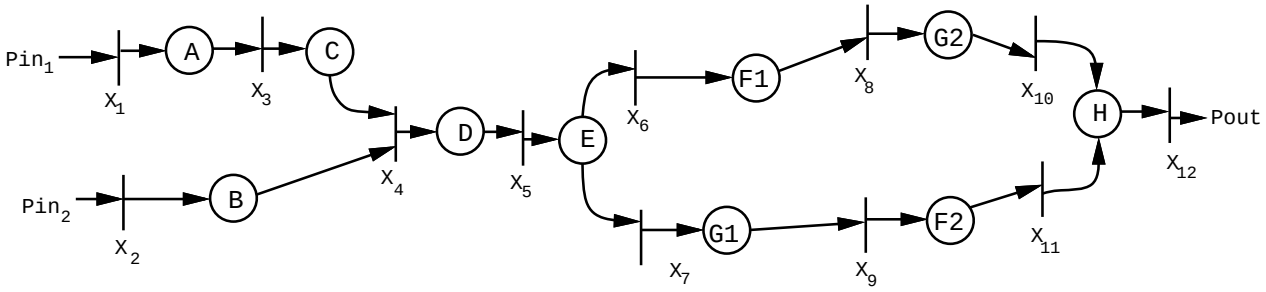


Figure 13: Petri Net Representation of Plan

action H will be started after either X_{10} or X_{11} fires.

6.2 Resource Usage and Generic Resources

To actually perform the actions suggested by our planner, we will need to use some resources. For example, if action C corresponds to "Paint the Block" then we must reserve some type of painting tool before we can start action C (i.e. before we can fire transition X_3). The F_r and S_r matrices represent the resources needed by the actions; 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

$$F_v = \begin{array}{c} \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \\ X_9 \\ X_{10} \\ X_{11} \\ X_{12} \end{array} \begin{array}{c} \begin{array}{cccccccccccc} P_{in\,A} & P_{in\,B} & A & B & C & D & E & F1 & G1 & G2 & F2 & H \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

$$S_v = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F1 \\ G1 \\ G2 \\ F2 \\ H \\ P_{out} \end{array} \begin{array}{c} \begin{array}{cccccccccccc} X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 & X_8 & X_9 & X_{10} & X_{11} & X_{12} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 14: F_v and S_v Matrices

means that when transition X_j fires, resource R_i is released.

Initially, we assume that every action has its own dedicated resource. That is, if a transition starts action A_i, it will also reserve resource $\hat{R}_i$ and if the completion of action A_j causes a transition to fire, then the transition will also release resource $\hat{R}_j$. Figure 15 shows the Petri Net corresponding to these resources. Notice that except for directionality, this Petri Net is identical to the one shown in Figure 13. Because of this similarity, our initial resource matrices can be quickly computed:

$\hat{F}_r = S_v^T$, 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.

6.3 Resource Assignment

Our initial matrices assume that each action has a dedicated resource. In most cases, this assumption is unrealistic. If actions A, D, and E all involve drilling something but we only have one machine that can drill, then the single resource must be shared among the three actions. This sharing of resources can be represented by a resource assignment matrix, F_a. A 1 in position i, j of F_a means that the actual resource R_j will be used to perform the functions of our dedicated resource $\hat{R}_i$. If a column j has two or more 1s in it, then the actual

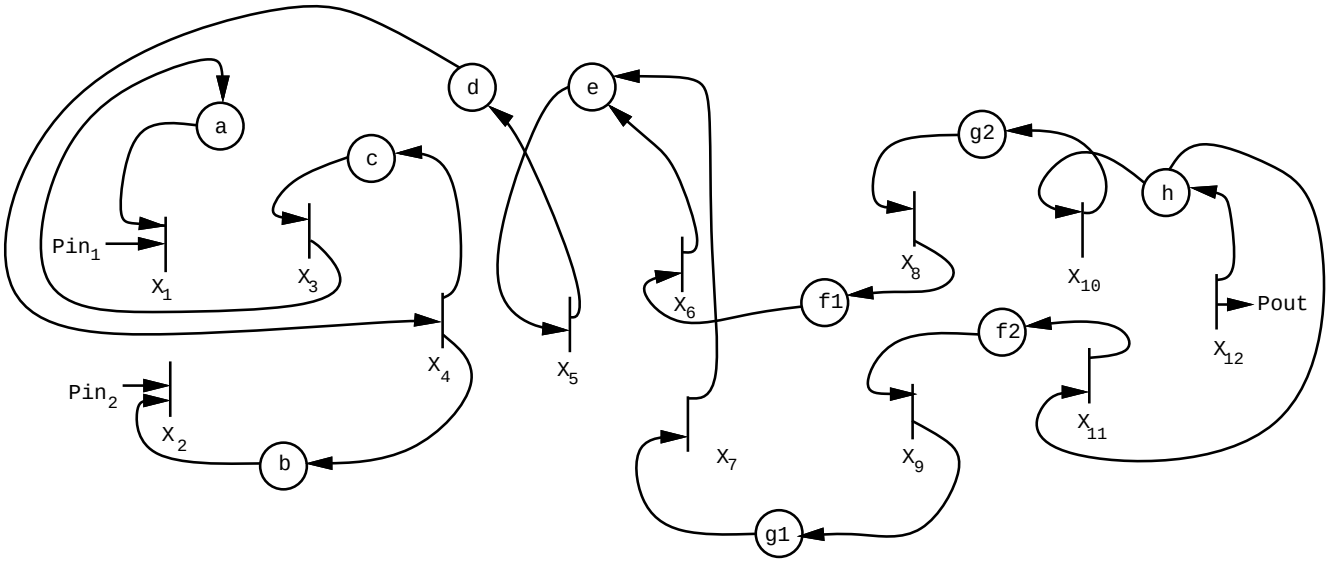


Figure 15: Petri Net for Generic Resources

resource R_j is performing the functions of more than one dedicated resource and is thus being shared among two or more actions. Figure 17 shows a resource assignment matrix with two shared resources; a single resource *ade* will perform the functions of the generic resources $\hat{a}$, $\hat{d}$, and $\hat{e}$ (and thus be shared among actions A, D, and E) and a single resource f will perform the functions of generic resources $\hat{f1}$ and $\hat{f2}$ (and thus be shared among actions $F1$ and $F2$).

We can use the resource assignment matrix to easily adjust our generic resource usage matrices into actual resource usage matrices:

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

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

$$\hat{F}_r = \begin{array}{c} \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \\ X_9 \\ X_{10} \\ X_{11} \\ X_{12} \end{array} \begin{array}{c} \begin{array}{cccccccccc} \hat{a} & \hat{b} & \hat{c} & \hat{d} & \hat{e} & \hat{f1} & \hat{g1} & \hat{g2} & \hat{f2} & \hat{h} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\hat{S}_r = \begin{array}{c} \\ \hat{a} \\ \hat{b} \\ \hat{c} \\ \hat{d} \\ \hat{e} \\ \hat{f1} \\ \hat{g1} \\ \hat{g2} \\ \hat{f2} \\ \hat{h} \end{array} \begin{array}{c} \begin{array}{cccccccccccc} X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 & X_8 & X_9 & X_{10} & X_{11} & X_{12} \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

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

$$F_a = \begin{array}{c} \\ \hat{a} \\ \hat{b} \\ \hat{c} \\ \hat{d} \\ \hat{e} \\ \hat{f1} \\ \hat{g1} \\ \hat{g2} \\ \hat{f2} \\ \hat{h} \end{array} \begin{array}{c} \begin{array}{ccccccc} ade & b & c & f & g1 & g2 & h \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 17: F_a Assigns Resources

Matrix F_r has been called the resource requirements matrix (Kusiak 1992). A 1 in position i, j of F_r means that resource j is needed to perform job i. If a column j has more than one 1, then resource j is being shared by more than one job.

One problem can result from our notation. Consider the actual resource *ade*. Originally, transition X_5 reserved resource $\hat{e}$ and released resource $\hat{d}$. 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 operation corresponds to the three matrix equations, in which "&" represents an element-by-element logical AND operation and "-" represents ordinary matrix subtraction:

$T_s = F_r \& S_r^T$

$F_{r_{new}} = F_{r_{old}} - T_s$

$S_{r_{new}} = S_{r_{old}} - T_s^T$

The completed F_r and S_r matrices for our sample problem are shown in Figure 18. These matrices correspond to the complete Petri Net shown in Figure 19. Note that places from the F_v and S_v matrices describe actions and places from the F_r and S_r matrices describe resources. 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.

7 Multiple Plans; One Controller

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. 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. In Section 7.2, we present a polynomial-time algorithm for combining the plans into a minimal matrix representation.

$$F_r = \begin{array}{c} \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \\ X_9 \\ X_{10} \\ X_{11} \\ X_{12} \end{array} \begin{array}{c} \begin{array}{ccccccc} ade & b & c & f & g1 & g2 & h \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$S_r = \begin{array}{c} \\ ade \\ b \\ c \\ f \\ g1 \\ g2 \\ h \end{array} \begin{array}{c} \begin{array}{cccccccccccc} X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 & X_8 & X_9 & X_{10} & X_{11} & X_{12} \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 18: Final S_r and F_r Matrices

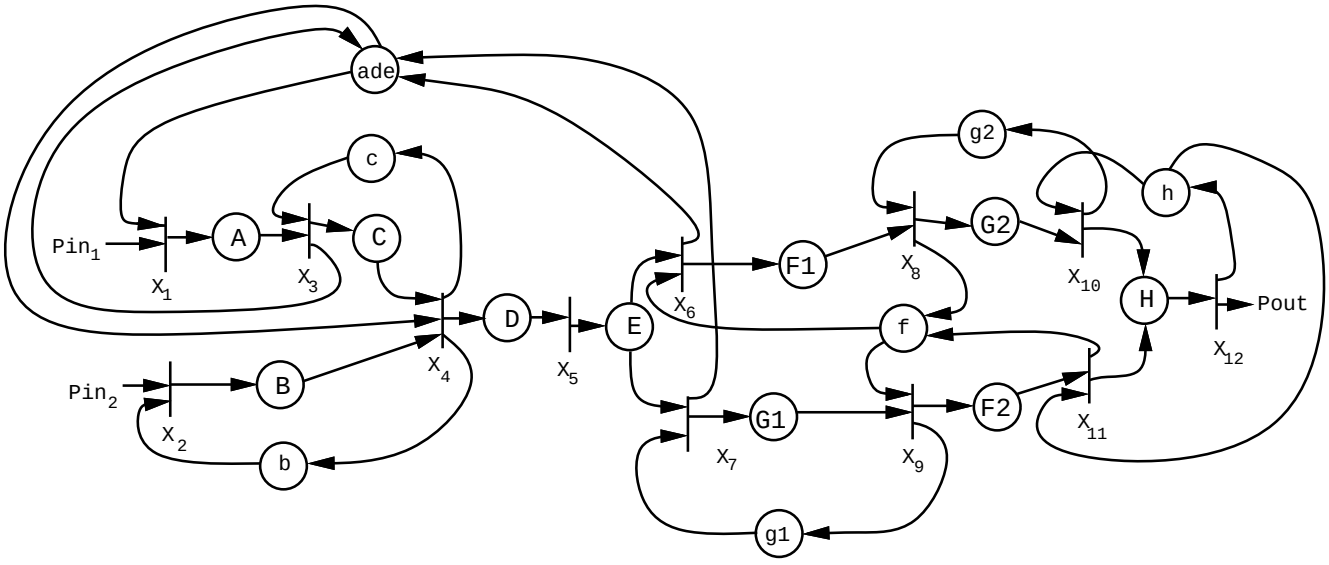


Figure 19: Our Completed Petri Net

7.1 Forming more than one Plan

Figure 20 shows two possible assembly trees for constructing an S_4 part. In (Gračanin, Srinivasan & Valavanis 1994), Gračanin uses a parameterized Petri net to represent the alternate means of assembling an S_4. Our system can incorporate the alternatives into a matrix notation, which is computationally easier to manipulate.

HTN operators corresponding to the two assembly trees are shown in Figure 21. 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 "bet-

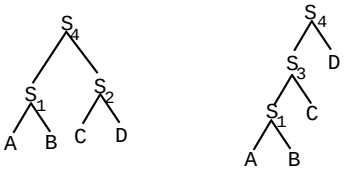


Figure 20: Alternate Assembly Trees

ter", 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.

Figure 22 shows the two possible plans for assembling an S_4 part. Next, we convert each plan into matrices as described in Section 5. Figure 23 shows the two Petri Nets for this problem, and Figure 24 shows the corresponding pairs of matrices.

7.2 Combining Multiple Plans

In this section, we present a polynomial-time algorithm for finding a minimum matrix representation for multiple plans. We use F_{v_c} and S_{v_c} to accumulate a matrix combining the alternatives described by each plan. We initialize F_{v_c} to the F_v matrix of our first plan and we initialize S_{v_c} to the S_v matrix of our first plan. We call our algorithm

```
(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 21: HTN Operators for Figure 16

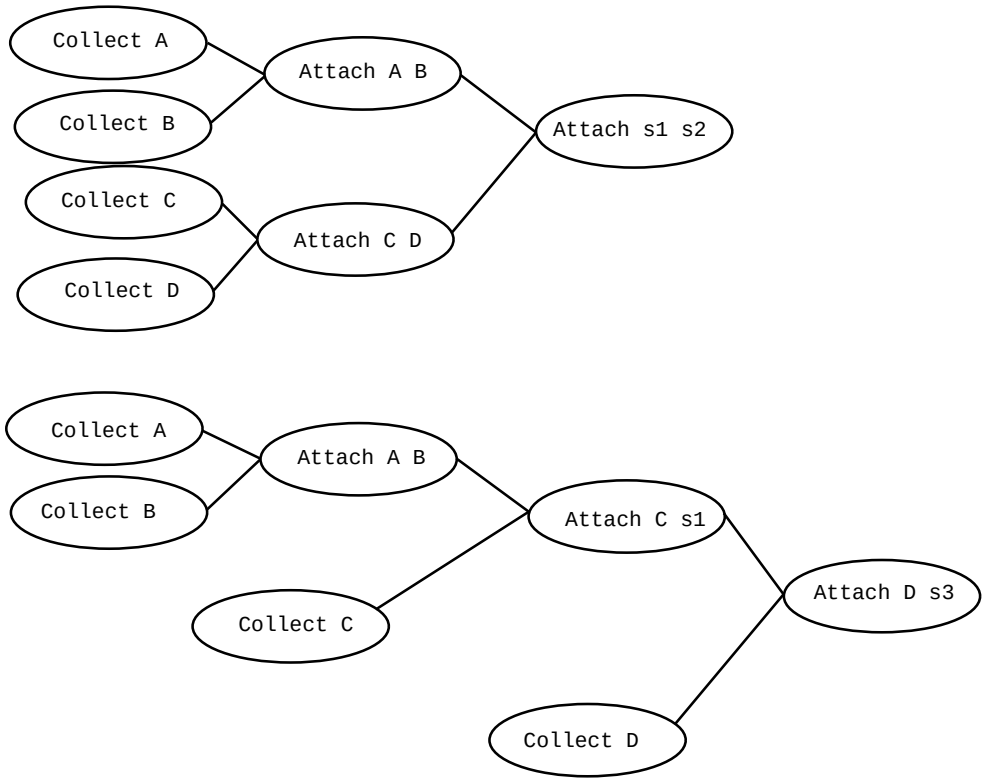


Figure 22: Multiple Plans for Assembling an S_4 Part

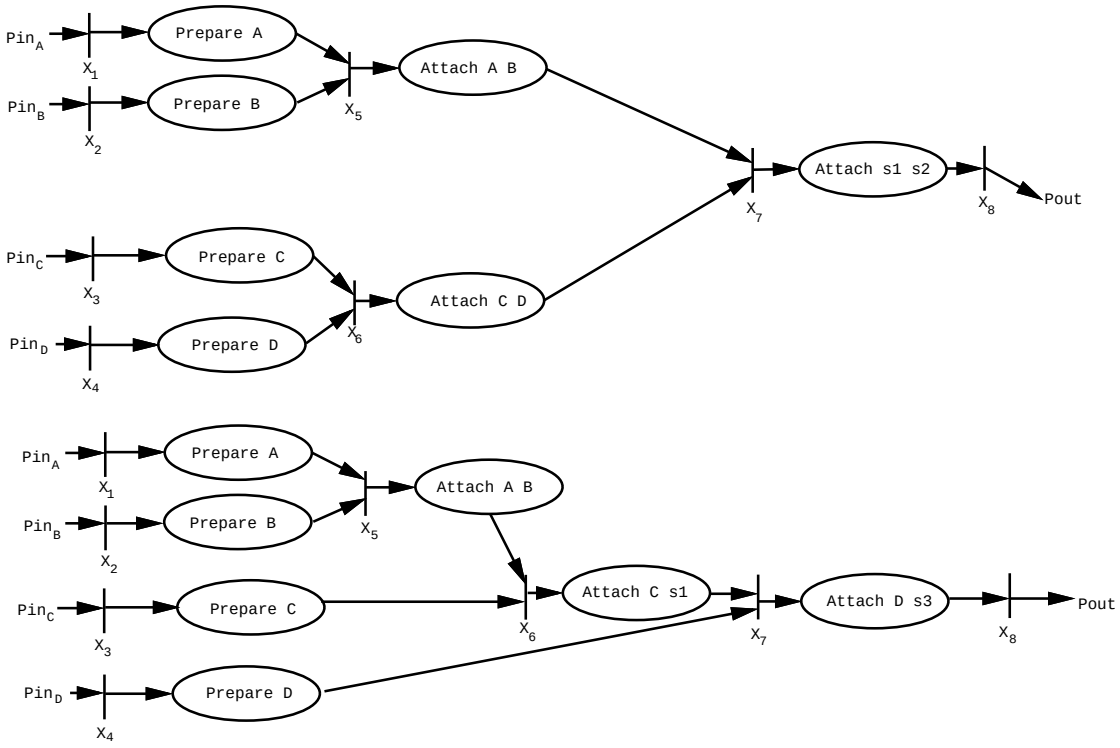


Figure 23: Multiple Petri Nets for Assembling an S_4 Part

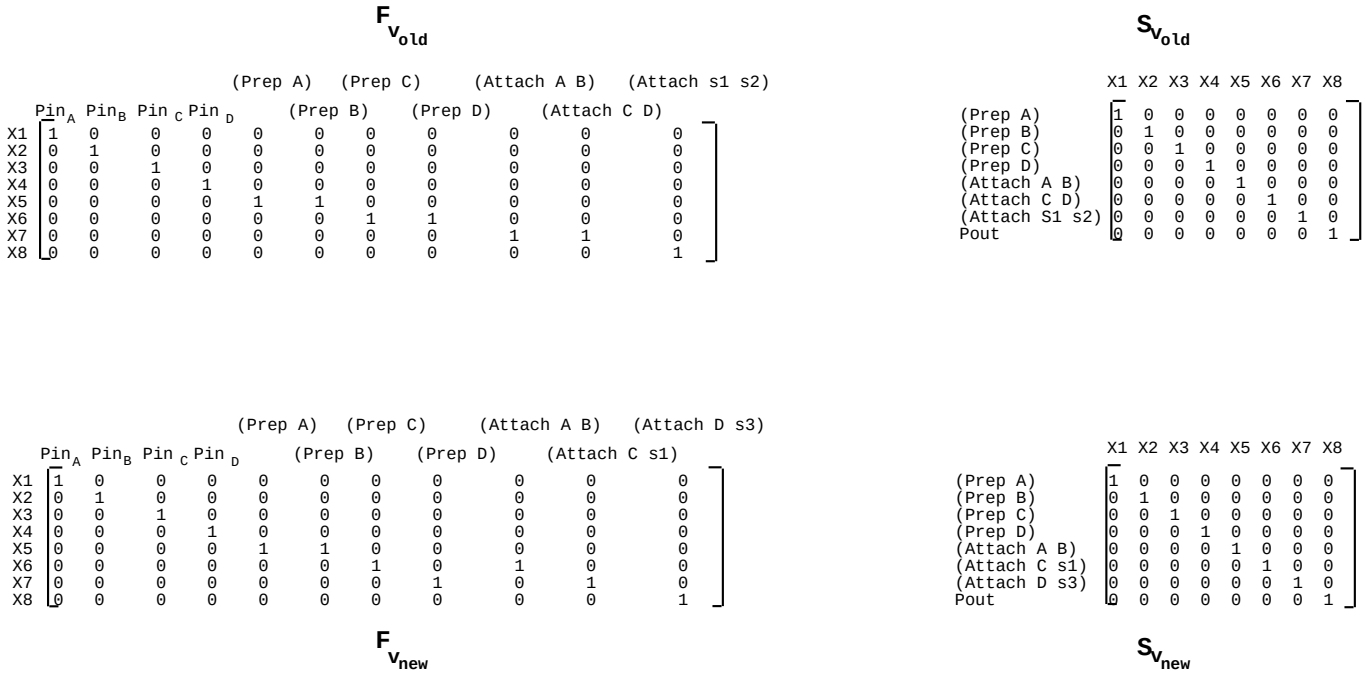


Figure 24: Multiple Matrices for Assembling an S_4 Part

once for each alternate plan; after we have finished, F_{v_c} and S_{v_c} contain information on all possible plans. We can now form generic resources and assign actual resources as described in Section 6.

Our algorithm relies on two crucial assumptions:

1. Parts with the same label on two different assembly trees are 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 (Gračanin et al. 1994), it would be extremely difficult to combine multiple assembly trees at all. Assumption 2 is crucial; it 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.

Our algorithm, shown in Figure 25, has two major steps. First, it associates each place in our new Petri Net with a (possibly new) place in our old Petri Net. Second, it examines each transition in the new Petri Net and decides if that transition should be added to our old

```
CombineMatrix(F_Vold, S_Vold, F_Vnew, S_Vnew)

  For each place in F_Vnew
    Does the place exist in F_Vold?
    If so:
       Associate the place with the correct place of F_Vold
       Associate the corresponding place of S_Vnew with S_Vold
    If not:
       Create a new place in F_Vold and S_Vold
       Associate the place in F_Vnew with the new place in F_Vold
       Associate the corresponding place of S_Vnew with the new place in S_Vold

  For each transition in F_Vnew, S_Vnew
    Does the corresponding transition exist in F_Vold, S_Vold?
    If so:
       (Do nothing)
    If not:
       Create the transition
```

Figure 25: Combining Multiple Plans

Petri Net. We describe these steps in detail in the next two sections.

7.2.1 Step 1: Examining Places

Each row of S_v and the corresponding column of F_v refers to a place in a Petri-Net corresponding to a primitive action. Our algorithm begins by associating each place in our new matrix pair with a (possibly new) place in our accumulating matrix pair. If our current place has a label identical to a place in our old matrices, then we can associate the current place with the matching place in the old matrix pair (using Assumption 2 above). Otherwise, we create a new place with that node label and associate the current place with the newly

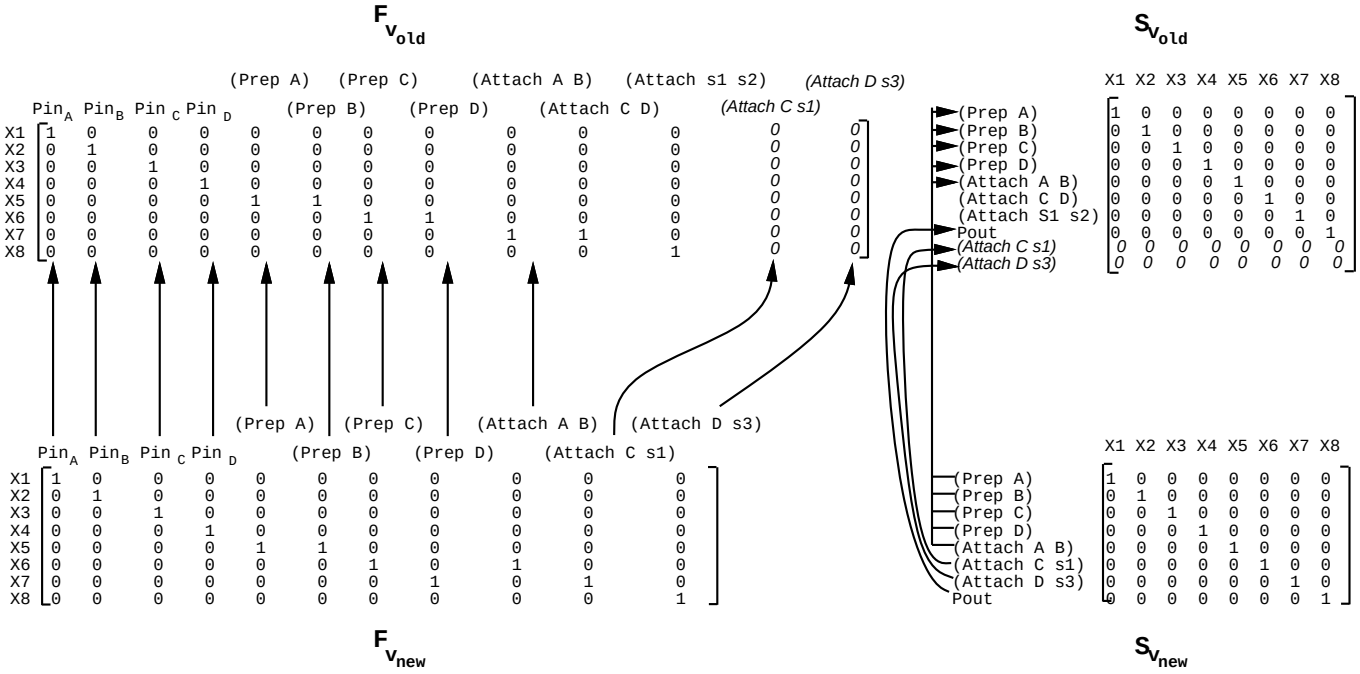


Figure 26: Intermediate Results: Combining Places from Multiple Plans

created 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 26 shows the associations formed by this phase of our algorithm. Rows and columns in *italics* are newly added places.

7.2.2 Step 2: Examining Transitions

After adding new places as needed, our algorithm examines each transition in our new plan. Our algorithm examines our accumulated plan

to decide whether the transition already exists *based on the associated nodes* of our accumulated plan. Thus, our algorithm is judging transitions based on the node labels of the places the transitions link rather than on the place numbering used by our new matrices. For example, transition X_4 of the new matrix set links P_{in_D} and (Prepare D). Our old 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 old set makes this connection, so we create a new transition X_{10} incorporating this link.

Figure 27 shows the final F_{v_c} and S_{v_c} matrices. Transitions in the new matrix set which do not have corresponding transitions in the old set are marked with a *, and the newly created transitions are in *italic type*. The single set of matrices $F_{v_{old}}$ and $S_{v_{old}}$ now represent two different methods of producing S_4 parts. Figure 28 shows the combined Petri Net.

7.3 Complexity Analysis

Suppose we have m plans and, after converting the plan to a Petri Net, each plan has an average of n steps. The number of transitions is $\theta(n)$ (We prove this in the Appendix).

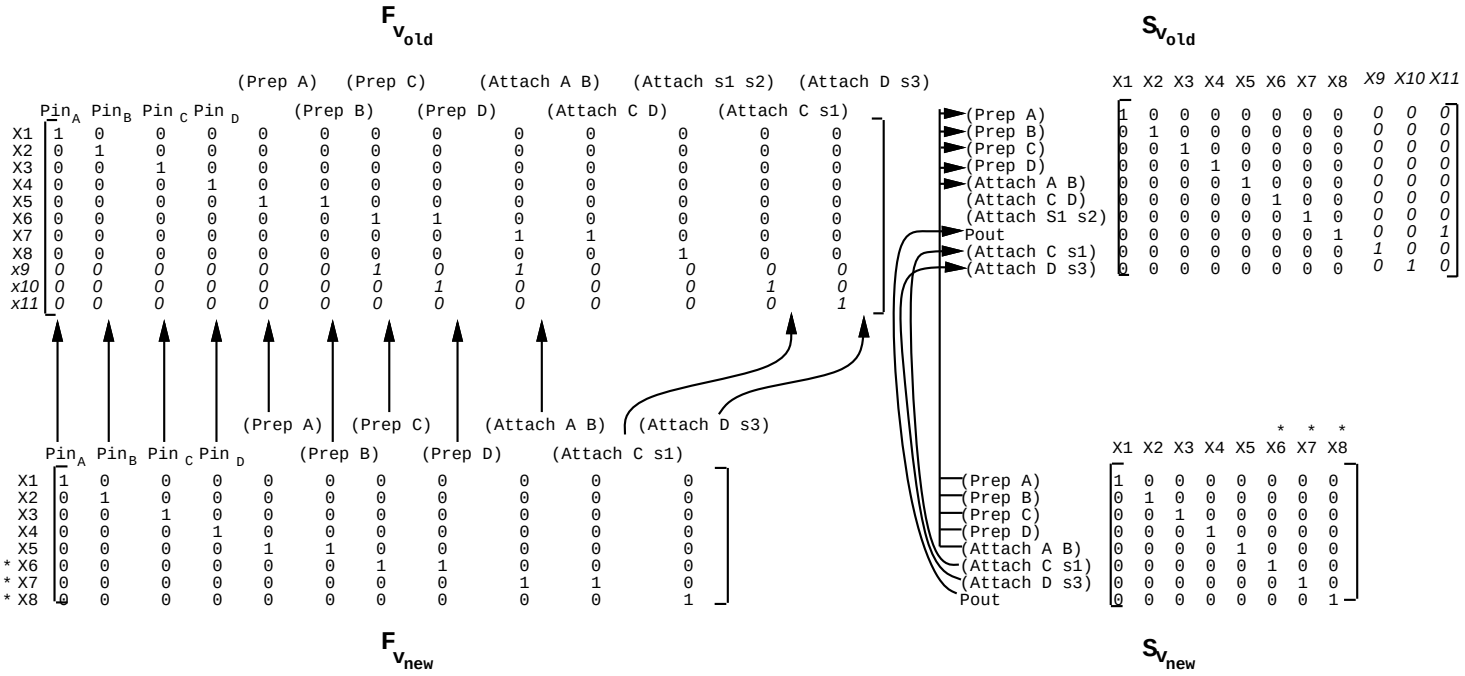


Figure 27: Final Results: Combining Transitions from Multiple Plans

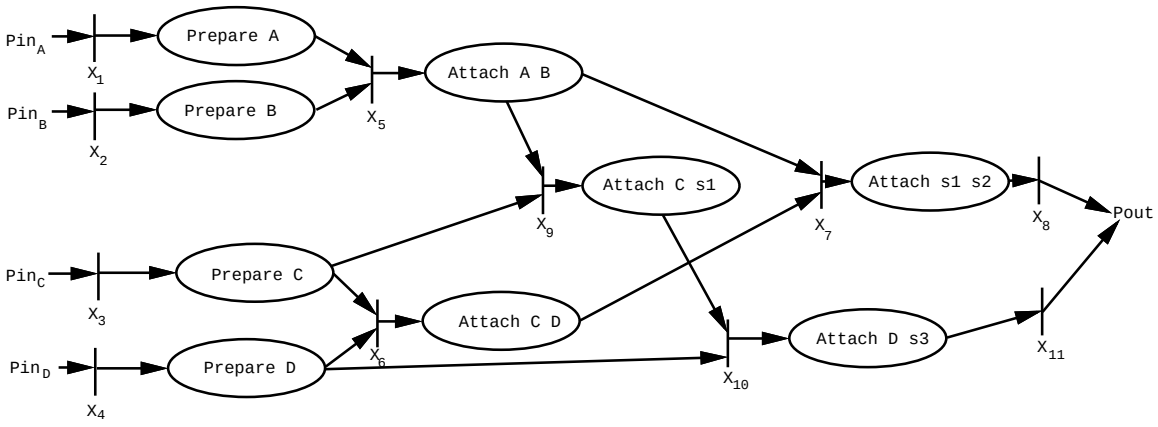


Figure 28: Combined Petri Net

We will call our algorithm $m-1$ 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 $m*n*O(m+n) = O(nm^2+mn^2)$ time.

8 Dynamic Supervisory Controller from Task Matrices

Our machine planner corresponds to the "Organization Level" of Figure 1. This planner gives us four Task Plan Matrices (F_v, S_v, F_r, S_r),

with which we can directly implement a supervisory job coordinator that performs dynamic on-line decision-making control, detailed sequencing and routing of jobs, and final dispatching assignment of shared resources. The supervisory controller performs the tasks of the "Coordination Level" of Figure 1. This matrix rule-based controller provides a framework for *rigorous analysis* of the system including its structure and protocols, complexity, circular waits, siphons, and deadlock. The controller is illustrated in Fig. 29 and described by the following equations:

Matrix Controller State Equation

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

Job Start Equation

$$v_s = S_v x \tag{2}$$

Resource Release Equation

$$r_s = S_r x \tag{3}$$

Task Complete Equation

$$y = S_y x \tag{4}$$

All matrix operations are defined to be in 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 Fig. 29, 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 $[v \quad r]$ 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

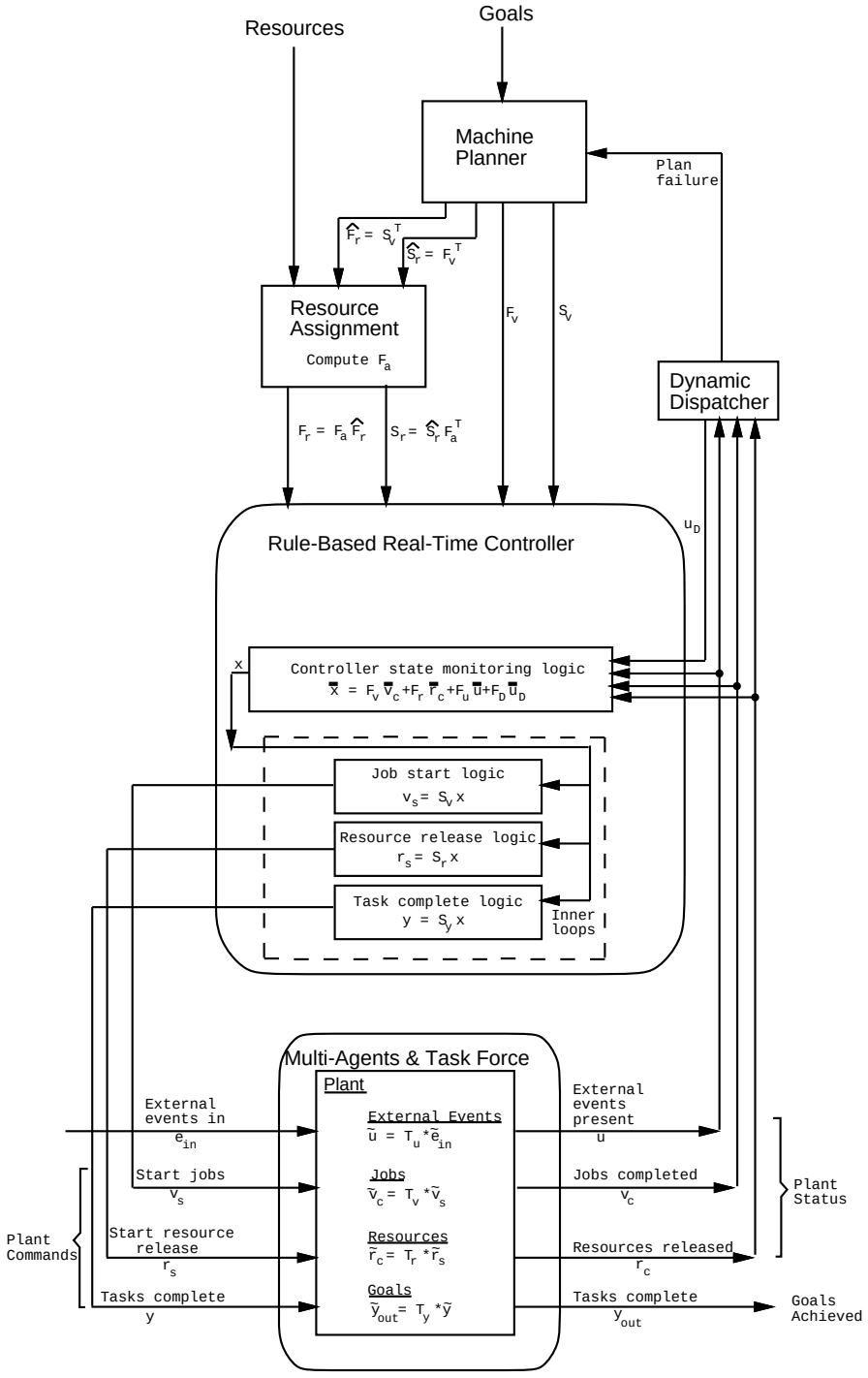


Figure 29: Matrix-based decision-making supervisory controller

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 Fig. 29, 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 29 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

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

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, Huang, Fierro & Tacconi 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.

9 Conclusion

In this paper we have shown how machine planners can be integrated with a real-time intelligent control system. Planners can use existing documentation (assembly trees) to form their operators. We have demonstrated how machine planners can represent flow-lines, assembly operations, and routing choices. The output of a planner can

be converted into a set of matrices which in turn can be executed by a matrix-based controller. We have given a polynomial-time algorithm for combining multiple plans into a single minimal matrix set. Our method of combining multiple plans into a single framework simplifies re-planning, and we hope to produce a general-purpose planner which can combine alternatives. However, our method does not handle partial ordering choices ideally and our assumption that identically-worded operators can be combined may not be realistic for other domains. We also hope to exploit the power of machine planning in other parts of the manufacturing process; perhaps a planner can assist with dispatching or can form plans even in the absence of assembly tree information.

A Proving a Bound on the Number of Transitions

Our complexity analysis uses the fact that a plan that has n steps (after partial-ordering choices are explicitly enumerated) will have $\theta(n)$ transitions. We now prove this. First, we will define the different types of transitions and places used in our plans. We establish a lower bound on the number of transitions by showing that at least half of all places will have at least one succeeding transition. We elaborate on the role of job-shop begin and job-shop end places. Next, we will assign each transition to an "owning" place. We establish an upper bound on the number of transitions by proving that no place will own more than two transitions. Throughout this section, we consider only "action" nodes — that is, our Petri Net does not contain any places corresponding to generic or actual resources. In addition, we assume that the maximum number of parts assembled in any assembly operation is bounded by a constant.

A.1 Types of Transitions and Places

Section 5 describes how Flow-Lines, Assembly, and routing choices are represented in our Petri Nets. More formally, we categorize the types of each transition and place:

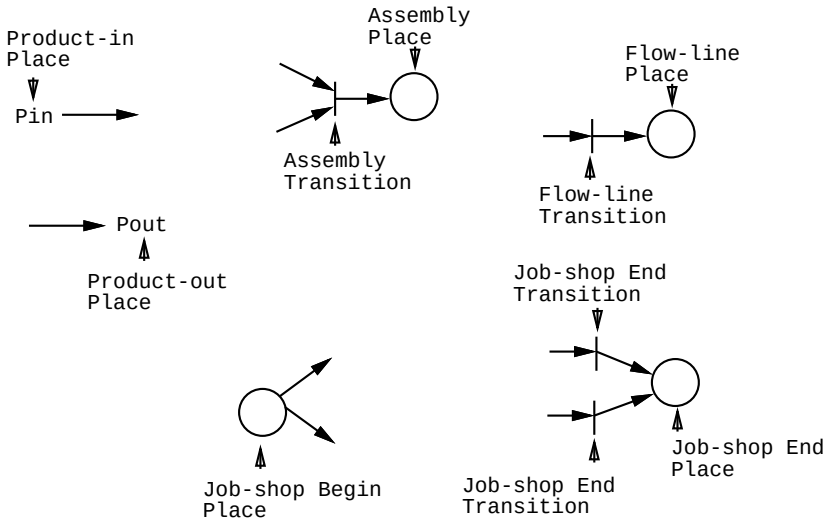


Figure 30: Types of Transitions and Places

Assembly A transition that has more than one input place is an *assembly* transition. Note that because we do not yet consider resource information, each transition will have only a single output place. In this case, the output place is an assembly place.

Flow-line If a transition has a single input place, and if the output place is **not** also an output place for another transition, then the transition is a *flow-line* transition. The output place is a flow-line place.

Job-shop End A transition with a single input place, but whose output place is also the output place of other transitions is part of a set of *job-shop end* transitions. The output place for these

transitions is a job-shop end place.

Job-shop Begin A place with more than one succeeding transition (that is, a place which is the input place for more than one transition) is a *job-shop begin* place. Note that each job-shop begin place will also have another type. That is, each job-shop begin place will also be an assembly place, a product-in, a flow-line place, or a job-shop end place.

Product-in A place which has no preceding transition (that is, no transition uses the place as an output place) is a product-in place.

Product-out A place that has no succeeding transitions (that is, no transition uses the place as an input place) is a product-out place. Like job-shop begin places, a product-out place will also have another type, but Product-out places will **not** also be job-shop begin places.

A.2 Bounding the Number of Product-out Places

Our Petri Net will perform operations on one or more input products to produce one or more output products. In particular, each output product will be the result of at least one operation applied to one or more input products—after all, if an output product were identical to some input product, the output product could be entirely replaced by

the input product and not included in our Petri Net. This means that for each product-out place, there is at least one interior place used to form that product. Thus, an n-place Petri Net will have at most $n/2$ product-out places, leaving a minimum of $n/2$ interior places. Every place other than a product-out place will have at least one succeeding transition. Thus, our Petri Net will have a minimum of $n/2$ or $\Omega(n)$ transitions.

A.3 Forming Job-shop Begin and Job-shop End Places

In our original plan, some steps were only partially ordered. When a Petri-Net corresponds to a single plan, a job-shop begin node indicates a choice of possible strategies. Each strategy will have one or more operations (corresponding to flow-line places or assembly places). Since the different strategies represent equivalent solutions, eventually the strategies will achieve the common result, corresponding to a job-shop end node. Thus, for our Petri-Nets, at least one flow-line place or assembly place will occur on each branch between a job-shop begin place and a job-shop end place. In particular, each job-shop end transition is preceded either by an assembly place or by a flow-line place.

A.4 Finding a Transition's Owner

Each transition is a flow-line transition, an assembly transition, or one of a set of job-shop end transitions. The owner of a flow-line transition is the relevant flow-line place. The owner of an assembly transition is the relevant assembly operation. The job-shop end transitions are **not** owned by the job-shop end place; rather, each job-shop end transition is owned by the preceding place. From above, this place will be either an assembly place or a flow-line place.

We now enumerate the possible numbers of transitions owned by each type of place. Assembly places and Flow-line places will own two transitions if they are succeeded by a job-shop end node; these places will own one transition otherwise. Product-in and job-shop end places will own no transitions. The number of transitions owned by a job-shop begin or product-out place depends on the type of the place—that is, a job-shop begin place which is also a product-in place will own no transitions and a product-out place which is also an assembly place will own one transition.

Thus, no place owns more than 2 transitions and a Petri-Net with n places will have a maximum of $2n$ transitions or $O(n)$ transitions. Since we have proven that a Petri-Net formed by our planner has $O(n)$ and $\Omega(n)$ transitions, it has $\theta(n)$ transitions.

References

Antsaklis, P. & Passino, K. (1992). *An Introduction to Intelligent and Autonomous Control*, Kluwer, Boston.

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.

Currie, K. & Tate, A. (1991). O-plan: The open planning architecture, *Artificial Intelligence* **52**(1): 49–86.

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

Dessochers, A. (1987). *Petri Nets for Automated Manufacturing Systems: Modeling, Control, and Performance Analysis*, Morgan Kaufmann Publishers, Inc., San Mateo, California.

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 and Co., New York.

Ghosh, S., Hendler, J., Kambhampati, S. & Kettler, B. (1992). *NONLIN Common Lisp Implementation (V1.2): USER MANUAL*, Computer Science Department, University of Maryland.

Gračanin, 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*, Tucson, pp. 1184–1189.

Kusiak, A. (1992). Intelligent scheduling of automated machining systems, *in* A. Kusiak (ed.), *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., Carbonell, J., Etzioni, O., Knoblock, C. & Kuokka, D. (1987). Acquiring effective search control rules: Explanation-

based learning in the prodigy system, *Proceedings of the Fourth International Workshop on Machine Learning* 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): 1–8.

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

Russel, S. & Norvig, P. (1995). *Practical Planning*, Artificial Intelligence: A Modern Approach, Prentice Hall, Upper Saddle River, New Jersey, chapter 12, pp. 367–380.

Sacerdoti, E. (1975). The nonlinear nature of plans, *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI'75)*, pp. 206–214.

Sussman, G. J. (1973). A computational model of skill acquisition, *Technical Report AT TR-297*, MIT:AI Laboratory.

Tacconi, D. & Lewis, F. (1997). A new matrix model for discrete event systems: Application to simulation, *IEEE Control Systems Magazine* pp. 62–71.

Tate, A. (1977). Generating project networks, *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJ-*

CAI'77) pp. 888–893.

von Martial, F. (1992). *Coordinating Plans of Autonomous Agents*, Springer-Verlag.

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

Wilkins, D. E. (1990). Can ai planners solve practical problems?, *Computational Intelligence* **6**(4): 232–246.

Wilkins, D. E. (1994). Comparative 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.

Yang, Q. & Tenenberg, J. (1990). Abtweak: Abstracting a nonlinear, least commitment planner, *Proceedings of the Eighth National Conference on Artificial Intelligence* pp. 204–209.

Zhou, M. C. & DiCesare, F. (1993). *Petri Net Synthesis for Discrete Even Control of Manufacturing Systems*, Kluwer, Boston.

List of Figures