Using Machine Planning to Design Manufacturing Processes*

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

Abstract

In our current work, we form a manufacturing system using four matrices representing the job sequences and resources required to assemble various products. We convert traditional manufacturing assembly trees into plan operators which our machine planner uses to generate two matrices describing job sequencing. With a combination of ideas from analogical and hierarchical planning, our planner uses a novel approach to improve planning efficiency. The remaining two matrices describe resource usage; we form them by combining an initial assumption of dedicated resources with explicit resource assignment. In addition, we can succinctly incorporate alternate plans into a single matrix set.

Introduction

While machine planners have been used to manage a brewery (Wilkins 1990), they are not routinely used in manufacturing systems. Since machine planners and manufacturing systems usually have different representations to describe the ordering constraints between operations or the resources needed to execute an operation, integrating the two can be a challenge. We have integrated a machine planner with a matrix-based controller designed to control manufacturing systems. Our planner uses HTN operators derived from one or more manufacturing assembly trees to form plans to assemble a given part. The plan is converted into two job-sequencing matrices; we then assign resources to the plan steps to form two resource matrices. Our controller uses these four matrices, with real-time information on the plant status to control the execution of our plan. Thus, we use manufacturing information to encode constraints into an operator format usable by our planner, and we convert the completed plan into a matrix format usable by our manufacturing cell.

After describing the four matrices needed by our controller, we demonstrate our method of converting assembly trees into HTN plan operators. We show how flowlines, assembly operations, and job-shop choices can be accurately represented using HTN operators and included in our plans. Next, we show how to convert our planner's output into F_v and S_v matrices and how resource information can be encoded to form the F_r and S_r matrices. We summarize the execution of our controller. Our system incorporates multiple plans into a single set of matrices; we give a brief example of this process. We conclude by presenting some ideas on incorporating hierarchical planning ideas into an analogical planner to further improving the efficiency of search.

*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

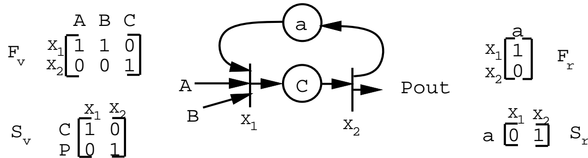


Figure 1: Sample Task Matrices

Dynamic Supervisory Controller for Manufacturing Systems

Our manufacturing system uses four task matrices corresponding to a Petri-Net in which some entries (in the F_v and S_v matrices) represent tasks and other entries (in the F_r and S_r matrices) represent resources. Tokens in resource entries represent the availability of one or more instances of that resource; tokens in job entries represent the completion of tasks. Firing a transition represents the cessation of one activity and initiation of another activity, with corresponding changes in the resources used.

Figure 1 shows a simple system in both Petri-net and matrix form. Two incoming parts, A and B, can be assembled via some process C into an outgoing part, P_{out}. This assembly step makes use of a resource "a".

In matrix terms, a 1 in location (i, j) of F_v means that transition X_i can not fire until action A_j completes. More than one 1 in the same row of F_v indicates an assembly operation. For our example, processes relating to both A and B must complete before transition X_1 can fire. More than one 1 in the same column of F_v indicates the start of a job-shop choice; that is, the dispatcher has a choice of which operation to perform on the part.

A 1 in location (i, j) of S_v means that when transition X_j fires, action A_i is initiated. More than one 1 in the same row of S_v indicates the completion of a job-shop choice; that is, two different sequences of operations have produced the same subassembly. For our system, S_v will never have more than a single 1 in the same column. This would represent the need to have two actions begin at the same time; we represent this by considering the two processes to comprise a single action.

F_r records which resources are needed for our tasks. A 1 in in location (i, j) of F_r means that transition X_i can not fire until resource R_j has been secured. If more than one 1 appears in the same column of F_r, then that resource is being shared by more than one job and thus has the potential to deadlock. Our controller currently only allows for one resource per job, so F_r will not have more than a single 1 in a given row.

S_r identifies when resources are released; a 1 in location (i, j) of S_r indicates that firing transition X_j will free resource R_i. Shared resources will be released by more than one transition and will thus have more than one 1 in their row of S_r. More than one 1 in the same column of S_r indicates that a single transition will release multiple resources; currently this only occurs during assembly operations.

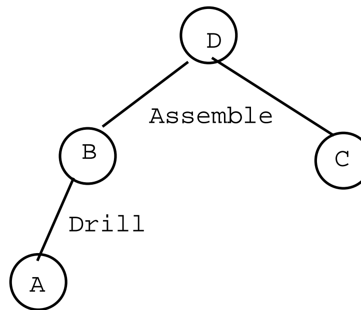


Figure 2: Sample Assembly Tree

Forming Manufacturing Plans

Manufacturing assembly trees describe how, after constructing various subassemblies, a manufacturing cell can construct a completed product. HTN plan operators describe how, after completing various subgoals, a planner can achieve some desired goal. In this section we show how assembly trees can be represented as HTN operators; the plans formed from these operators can represent manufacturing concepts including flow-lines, assembly operations, and job-shop choices.

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 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.

Hierarchical Task Network Planning

Hierarchical task network planning provides the planning system with task schemas as well as traditional operator descriptions (Wilkins 1984). Task schemas provide a method of grouping operators together to form higher-level operations and allow the domain to explicitly represent known ordering constraints. Thus, HTN planners can reduce search compared to conventional planners.

Converting Assembly Trees Into HTN Operators

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 *et al.* 1992) to implement the operators described in this paper.

Figure 3 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 4 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.

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) become 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

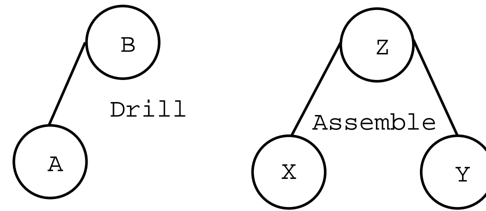


Figure 3: Sample Assembly Trees

```
(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 4: Operator Descriptions

incoming parts, can be converted into HTN operators with no subgoals. For example, if part A of Figure 3 is not constructed locally, the HTN operator shown in Figure 5 will "assemble" it without forming subgoals. Figure 6 summarizes our method of converting assembly trees into plan 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 4 and Figure 5, 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 7 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

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

Figure 5: Handling Product-Ins

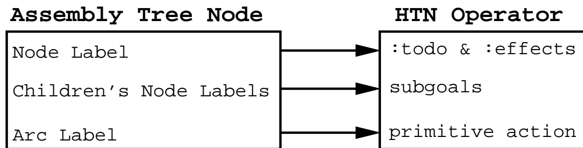


Figure 6: Converting Assembly Trees into HTN Operators

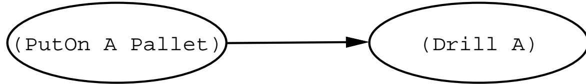


Figure 7: Plan to Assemble B

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 later, plan operators can easily represent alternate means of constructing a particular part.

Job Shops

There is no standard method of representing general job-shop choices as assembly trees. Figure 8a 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 (Figure 8b); unlike assembly steps, the HTN operators for job-shops work on the same part. The resulting plan (Figure 8c) thus splits into two different sections and then rejoins. The plan says that part A 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 job-shop scheduling choices that are difficult to represent in assembly trees. We will later show 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; our controller uses the matrices to manage real-time dispatching and resource assignment.

Converting Plans into Matrices

In this section, we describe how to form the F_v and S_v matrices from our plan. Initially, we assume that each operation has a dedicated resource; by identifying which operations must share resources, we form F_r and S_r.

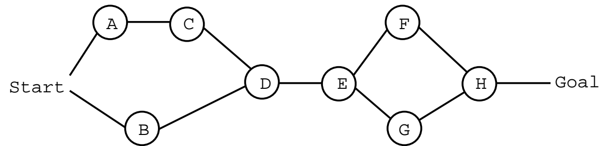


Figure 9: A Sample Plan

Matrix Representation of Job Sequences

Consider the plan shown in Figure 9, which includes assembly and routing alternatives. 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. This plan can be converted into the Petri-Net shown in Figure 10, 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. The Petri-Net corresponds to the two matrices shown in Figure 11.

Resource Usage and Generic Resources

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 $\hat{R}_j$. Under this assumption, our initial resource matrices can be quickly computed:

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

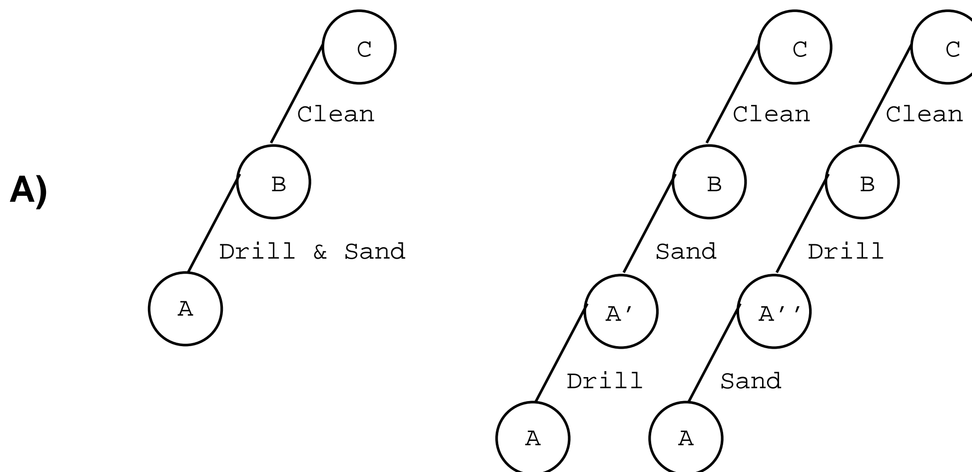
$\hat{S}_r = {F_v}^T$, with the product-in row(s) removed.

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 resource R_j is performing the functions of more than one dedicated resource and is thus being shared among two or more actions. We can use the resource assignment matrix to easily adjust our generic resource usage matrices into actual resource usage matrices:

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

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)

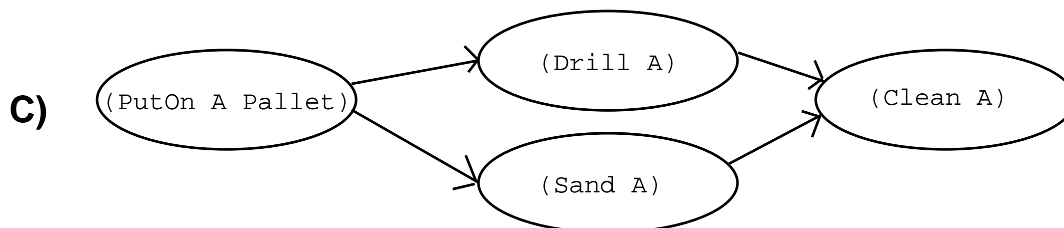


Figure 8: Representing Job-Shops

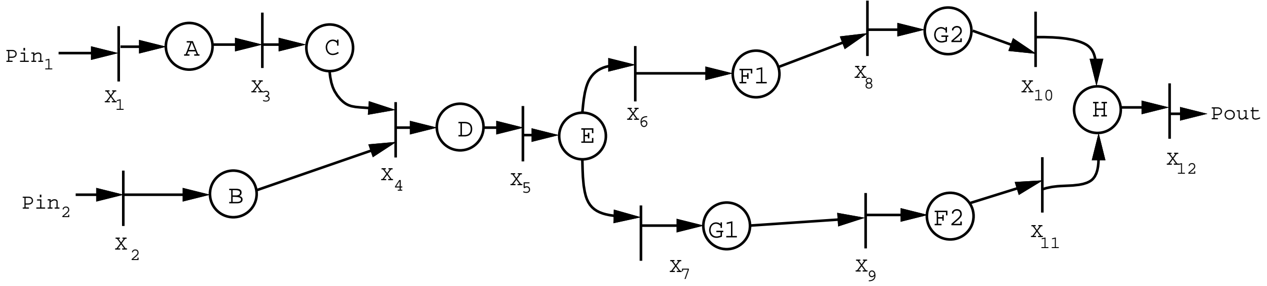


Figure 10: Petri Net Representation of Plan

$F_v =$

	$P_{in\,A}$	$P_{in\,B}$	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 11: F_v and S_v Matrices corresponding to the Petri-Net in Figure 10

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

One problem can result from our notation. Suppose we use a single actual resource *ade* to replace the generic resources $\hat{a}$, $\hat{d}$, and $\hat{e}$. 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:

$T_s = F_r \ \& \ S_r^T$

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

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

Assuming that one resource *ade* is used for the generic resources $\hat{a}$, $\hat{d}$, and $\hat{e}$, and that the single resource f is used for both $\hat{f1}$ and $\hat{f2}$, our completed F_r and S_r matrices are shown in Figure 12.

Plan Execution

Once we have formed our four Plan Task Matrices (F_v, F_r, S_v, and S_r), we can use a rule-based supervisory controller to perform detailed sequencing and routing of jobs and to dispatch shared resources. Our supervisory controller is shown in Figure 13, and is described in detail in (Tacconi & Lewis 1997) and (Lewis *et al.* 1995).

The heart of our controller is the "Matrix Controller State Equation":

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

This equation uses information on incoming parts, completed jobs, and available resources to determine which tasks the system should begin. The x vector corresponds to a Petri-Net firing vector; it determines which transitions in the Petri-Net will be taken.

In some cases, the system may have a choice between two incompatible transitions. For example, when we incorporate multiple alternatives into a single matrix set, we introduce job-shop choices into our F_v and S_v matrices. We use a separate dispatching unit to dynamically resolve the choices; the dispatcher provides an input U_D to the main supervisory controller.

Once we determine x, we can easily form control signals for which jobs to start and stop and which resources to acquire and release; these are given to the work cell.

All of our equations use *or/and algebra*, where "+" denotes logical *or* and "×" denotes logical *and*. The over-bar in our state equation represents logical negation (meaning, for example, that completed jobs are represented with a 0).

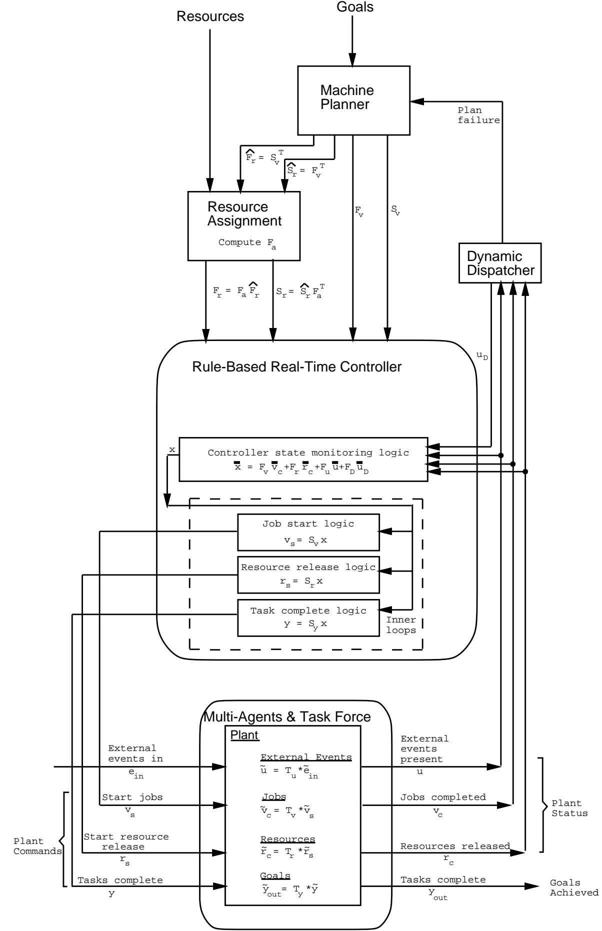


Figure 13: Matrix-based decision-making supervisory controller

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

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

Figure 12: Final S_r and F_r Matrices

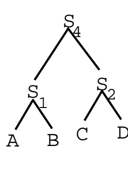


Figure 14: Alternate Assembly Trees

Managing Multiple Plans

Machine planners offer a powerful advantage compared to conventional assembly trees in that they can easily represent alternate methods of constructing a part. In (Gračanin, Srinivasan, & Valavanis 1994), Gračanin uses a parameterized Petri net to represent the alternate means of assembling an S_4 (shown in Figure 14). Our system can incorporate the alternatives into a matrix notation, which is computationally easier to manipulate.

The nodes of the assembly tree can be encoded into HTN operators; thus our planner can find more than one method of constructing an S_4 part. Our polynomial-time algorithm (described in detail in (Harris, Lewis, & Cook 1998)) begins by forming separate F_v and S_v pairs for each solution. It forms a mapping between places in the second plan and (possibly new) places in the first plan; based on this mapping, it determines which transitions from the second plan are novel and adds equivalent transitions to the first plan. Thus, the first set of F_v and S_v matrices will incorporate alternatives from both plans. Once the planner has formed these matrices, we can assign resources and form F_r and S_r as described previously.

Improving Planning Efficiency

Our system attempts to maintain planning efficiency by separating resource handling issues from job sequencing issues. In addition, incorporating known ordering constraints into HTN operators helps reduce the amount of search performed by the planner. However, our planner may still have efficiency problems especially if parts can be constructed in many different ways. In this section, we briefly examine analogical planning and present novel ideas on reducing the cost of analogical match by borrowing ideas from hierarchical (non-HTN) planning.

Analogical Planning

In manufacturing operations, typically a few plans are used repeatedly. Analogical planning is well suited for this class of use; an analogical planner stores complete plans in a case file. When the planner encounters a new problem, it looks for a "similar" case in its case file. The retrieved plan can be attempted as is for identical problems or *adapted* for a new problem. If a case must be extensively adapted or if no case is suitable for the new problem, many analogical planners will dynamically add new cases to the case file.

Traditionally, analogical planners used the case file to store a complete plan, with little or no information on the plan's causal structure. Recent work has proposed an alternative "derivational analogy" in which the planner stores the choices made during the original planning episode and can more easily determine whether the same choices are applicable to the new situation. Derivational analogy has been used with both total-order planners (Veloso 1992) and partial-order planners (Ihrig & Kambhampati 1994).

Hierarchical Matching

Analogy works well if the goal states and initial states are relatively small (even though the plans themselves may be quite complex). However, if most problems have several goals the planner must simultaneously achieve, then deciding which case is

most relevant can take time. If our problem mentions n objects, and each stored plan has k objects which must be matched to the new problem, finding the optimal mapping between the two plans may take up to $\binom{n}{k}$ steps (Hanks & Weld 1995).

We believe that search techniques from hierarchical planning (not to be confused with HTN planning) can reduce the cost of analogical match. Hierarchical planners use a divide-and-conquer approach to problem solving by first focusing on the most difficult or most important goals. Once the planner has a skeletal solution for the crucial goals, it refines the plan and begins to consider less crucial details. Hierarchical planning attempts to reduce a single tree of depth d into L mostly independent trees each of which has a depth of only d/L.

We believe that these same ideas can reduce the cost of performing analogical matching. Specifically, instead of trying to match n new objects with k old objects, a hierarchical matcher will only match the most crucial n/L new objects with the old k/L crucial objects. Under the assumption that the most difficult portions of the plan are considered first, the planner should prefer cases which completely solve the abstract portions of the plan to one which partially solves the abstract portion but completely matches objects for less crucial details. In other words, the planner can prune bad cases based solely on the quality of match of the most abstract objects. The planner next considers how to extend its abstract match to account for less important parts of the problem.

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, and the output from the planner can be converted into a set of matrices executable by a matrix-based controller. In addition, we can incorporate multiple alternate plans into single set of matrices. Our current research focuses on improving the planning phase by using hierarchical and analogical planning.

Our controller acts as an interface between discrete-time, abstract planning activities and continuous-time concrete execution units. Our planner provides a sequence of actions which must be performed to accomplish a goal. The dispatcher and matrix-based controller handle real-time scheduling based on available resources and current status. The agents perform their assigned tasks and report on real-time events. By incorporating alternatives into a single matrix set, our planner provides more generality than an unintelligent control system; if a partial failure prevents one alternative from succeeding, the dispatcher can quickly switch to an alternate plan.

Our future work focuses on two areas. First, we would like to quantify the space savings resulting from combining hierarchical and analogical planning ideas. Second, we would like to extend our controller to handle Materials Requirement Planning and Manufacturing Resource Planning. Our controller can then reason about deadlines and the lead time required for various operations.

References

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

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

Gračanin, D.; Srinivasan, P.; and 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.

Hanks, S., and Weld, D. 1995. A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research* 319–360.

Harris, B.; Lewis, F.; and Cook, D. 1998. Machine planning for manufacturing: Dynamic resource allocation and on-line supervisory control. *Journal of Intelligent Manufacturing* To Appear.

Ihrig, L., and Kambhampati, S. 1994. Derivation replay for partial-order planning. *Proceedings of the 12th National Conference on Artificial Intelligence* 992–997.

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

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

Veloso, M. 1992. *Learning by Analogical Reasoning in General Problem Solving.* Ph.D. Dissertation, Carnegie Mellon.

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.

Wolter, J.; Chakrabarty, S.; and Tsao, J. 1992. Methods of knowledge representation for assembly planning. *Proceedings of NSF Design and Manufacturing Systems Conference* 463–468.