A Structure-Oriented Approach to Assembly Sequence Planning

Sugato Chakrabarty and Jan Wolter

Technical Report 95-006
Department of Computer Science
Texas A&M University

Jan. 1995

Abstract

An assembly sequence planner is a system which, given a description of an assembly to be manufactured, identifies subassemblies and generates high-level plans for the construction of the each subassembly, including the final assembly. Previous research has shown that feasible plans can be efficiently produced for large assemblies, but the generation of good plans has met with less success. Not only are there difficulties in defining "good plans," but all published methods for finding good plans have computational complexities that make them unlikely to be practical for large assemblies. This paper shows that representing an assembly as a hierarchy of assembly structures can overcome many of these difficulties. A planner is described with uses the structure hierarchy both as a framework for structure-dependent definitions of a good plan, and as a tool for finding good plans more rapidly by using high-level expert advice, by reusing subplans for repeated substructures, and by not fully optimizing the plan. Analytical and experimental results are presented to demonstrate the effectiveness of the approach.

This research was supported by the National Science Foundation Presidential Young Investigator grant number DDM-905788.

1 Introduction

An assembly sequence plan is a high-level plan for constructing a product from its component parts. It specifies which sets of parts form subassemblies, the order in which parts and subassemblies are to be inserted into each subassembly, and the nominal trajectories along which those insertions are to be performed. Implicit in such a plan are high-level specifications for the fixtures and and manipulators that will be needed to carry out the plan.

An assembly sequence plan is not a complete, ready-to-execute production plan. Specific tools must still be chosen, and specific fixtures designed. Task plans for actually performing each assembly operation must be produced. However, it is the assembly sequence plan that provides the framework in which all these lower-level plans are generated, and promises that, taken together, they will accomplish the overall goal: the efficient and economical construction of the product.

In the last decade, a variety of automated systems have been designed to generate such assembly sequence plans. While excellent progress has been made in developing methods to quickly find a geometrically feasible plan for a product, there have been both definitional and computational problems with finding "good" plans.

The definitional problems of choosing good plans arise from the range and complexity of the issues that must be considered. Previous researchers have considered goodness criteria such as the attachment of partial assemblies and subassemblies[14], the number of tools and tool changes[11, 12], the number of workpiece reorientations[14, 27], the complexity of fixtures[27], the degree of parallelism permitted[9], the number of subassemblies[9], and the difficulty of the operations[12, 14, 27]. Typically numerical evaluation functions are created for each goodness criterion, and some weighted combination of them is used to select a plan.

Though these evaluation functions are extremely useful, they only skim the surface of the issues that production engineers commonly consider when planning the manufacture of complex assemblies. For example, in the assembly of an automobile there are safety considerations (*e.g.* don't connect the battery until after you have finished wiring the rest of the electrical system) and considerations involving the distributions of internal stresses created during assembly (*e.g.* when installing a wheel, tighten the bolts in a star-shaped pattern), as well as many others. Creating satisfactory numerical evaluation functions that can evaluate arbitrary plans for arbitrary products with respect to criteria as complex as these would be extremely difficult. But plans produced in ignorance of such criteria would have little value.

The second problem that confronts designers of assembly sequence planners is computational. Given some criteria for goodness, it is necessary to to search for valid plans that satisfy those criteria. But the number of possible plans for even fairly simple assemblies is extremely large[22], and given the complexity of realistic criteria, it is often difficult to produce good search heuristics. Thus optimizing algorithms typically run in exponential time and, even with very simple criteria, few are capable of solving problems with more than a couple dozen parts. There seems little hope that the approaches exemplified by these systems could be directly extended to solve large problems with realistic criteria.

This paper describes a new approach to assembly sequence planning that demonstrates significant advances in both the ability to specify complex, real-world criteria, and in the ability to rapidly find plans that are good with respect to those criteria. The basis for both advances is a view of an assembly as a hierarchical collection of standard structures. A simple planner (HAP, the Hierarchical Assembly Planner) has been implemented based on these concepts, and a variety of theoretical and experimental results have been obtained which demonstrate the promise of this approach.

2 Background

Clearly any valid assembly plan must ensure that no intersections occur between the parts as they are being assembled. Some early systems accomplished this by sweeping parts along their insertion trajectories and checking for intersections with other parts[8, 19]. However this method proves computationally expensive without careful buffering and reuse of geometric query results[7].

Capturing geometric information in symbolic constraint languages has led to faster planners. Two forms of constraints have been used. Insertion constraint languages consist of constraints on which parts block which insertion trajectories of other parts (*e.g.*, "if part p_1 is inserted with trajectory θ, it is blocked by part p_2")[23]. It has been demonstrated that these can be efficiently generated and used to rapidly produce geometrically valid plans for large assemblies[20].

The alternative is to use mating constraint languages, which are composed of constraints on the order in which pairs of parts may be joined (*e.g.*, "parts p_1 and p_2 must be joined before parts p_3 and p_4 are joined")[25]. DeFazio and Whitney[1, 5, 6] showed how to generate such constraints in an interactive process. Huang and Lee generate them automatically from insertion constraints[12]. Finding plans that satisfy these constraints, however, remains comparatively difficult.

The methods which have been developed for selecting a plan that is good with respect to some non-trivial criteria from the set of geometrically feasible plans have been less satisfactory. Many systems have been constructed which define quality metrics and then search for the best plan with respect to those metrics. Wolter, for example, rated plans by the number of insertion directions used, the stability of intermediate assemblies, and the manipulation complexity of the assembles and used an opportunistic branch-and-bound search method to find the best plan[23, 28]. Homem de Mello rated plans by their flexibility and parallelism and used an AO* search to find the best[9]. To search efficiently, these systems require admissible heuristic functions for each criterion. Since good heuristic functions are often difficult to find, this makes the addition of criteria difficult. Even with the small sets of criteria used in those systems, the time to search grows exponentially in some cases. Though Wolter's system was reasonably fast for some typical assemblies, there are some classes of simple assemblies, such as one in which a pair of parts is held together by n bolts, where run times grows exponentially with the number of parts.

De Fazio and Whitney[1] demonstrated a different approach to plan selection. They generate all possible plans, display them in a compacted form, and then allow the engineer to edit out the bad plans. Automated tools are provided to eliminate several classes of bad plans, but the ultimate judgement of which plan is best is left to the engineer. This avoids the problem of having to quantify the quality of plans and is probably a more appropriate way to choose plans. However, the number of plans presented to the user is potentially extremely large[24], and, even with some automated filters, the problem of editing the plan set down becomes overwhelming for even modest sized assemblies. Though the final choice of the plan should certainly be left to the engineer, systems are needed that can generate smaller sets of better plans for the engineer to choose from.

Pu has described a case-based approach to assembly planning[18]. In case-based approaches, a data-base of previously solved assembly planning problems is constructed, so new problems can be solved if they consist of subproblems similar to those in the case data-base. In such systems, no explicit goodness functions are used. Instead, knowledge about the best way to solve assembly planning problems is implicit in the case-base since only good plans would be put in the case-base. It is far from clear, however, that modifications of good subplans will always combine into good overall plans.

These three approaches toward finding high-quality plans each have distinct strengths and weaknesses. All are valuable. What is needed is an effective way to build a system where general evaluation functions, specific case-based advice, and expert user input can be integrated.

3 Approach

This paper describes a new approach to assembly sequence planning which provides greater flexibility in specifying quality criteria and greater efficiency in finding good plans for many common assemblies. The success of approach is based on four key ideas:

- The input to the algorithm is a hierarchical decomposition of the assembly into structures and substructures.
- The algorithm presumes the existence of a structure library, containing information about preferred ways to build common structures.
- The algorithm plans bottom-up, choosing plans for low-level structures and then merging them to create plans for higher-level structures.
- The algorithm uses evaluation functions to select plans, but does not normally attempt to fully optimize them.

3.1 Structure Hierarchy

Many previous assembly planners have used the concept of *liaisons* between pairs of parts to describe the significant relationships between the parts of an assembly. A liaison, as defined by Bourjault and popularized by DeFazio and Whitney, most often refers to a contact, but can in general refer to any relationship between two parts that is deemed significant by the engineer[2, 5].

This notion of a *liaison* can be generalized by defining an *assembly structure* to be a significant relationship involving some set of two or more parts. This includes not only pair-wise relationships such as liaisons, but also standardized relationships among larger sets of parts. For example, a nut and bolt holding together a series of parts is a very common structure. Exactly what sets of parts in an assembly form structures must ultimately be determined by the user, but typically structures will be either relationships that occur in many assemblies (such as the bolt structure), or that occur

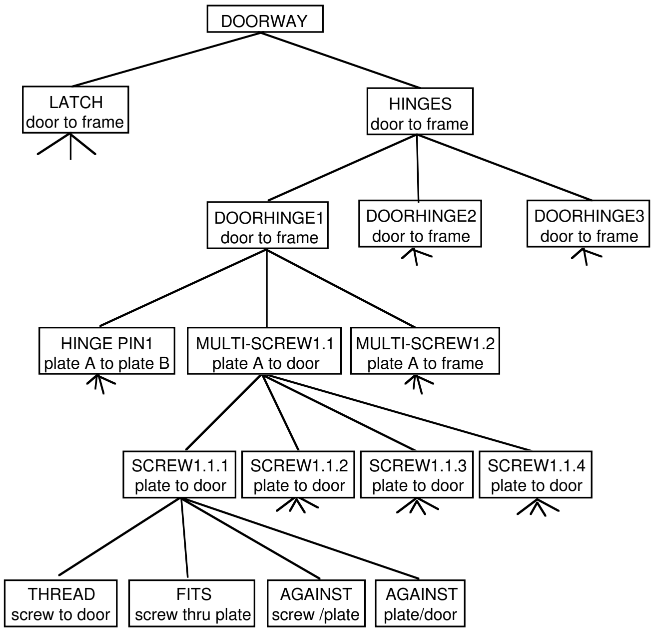

Figure 1: A portion of a structure hierarchy for a ordinary doorway hung on three hinges each of which is attached by sets of four screws to both the frame and the door.

multiply in the same assembly (such as a file-drawer structure in a filing-cabinet), or that serve to perform some key subfunction in the assembly (such as the electrical system in an automobile).

Naturally, some structures will contain other structures, as an automobile engine will contain many bolts. Thus, assembly structures may naturally be arranged into a hierarchy, with the large, high-level structures containing smaller substructures. For example, figure 1 shows an portion of a structure hierarchy for a doorway. The doorway structure is composed of a latch structure and hinge structure. The hinge is actually composed of three separate doorhinge structures, each of which consists of a hinge-pin-connection structure and two multiple-screw-attachment structures. The multiple-screw-attachment structures would be composed of four simple-screw structures, and the simple-screws would be composed of a variety of different types of contact structures, including planar "against" contacts, cylindrical "fits" contacts and threaded contacts. Note that traditional liaison graph descriptions of assemblies represent only the bottom level of this hierarchy.

It is important to note that such hierarchies are not unique. Different ways of decomposing the assembly may be motivated by different views of the assembly. For example, the doorway might have alternatively been decomposed into the fixed-parts substructure (the frame with its hinge plates screwed to it), the moving-parts substructure (the door with its hinge plates screwed to it), and the axis substructure (the three hinge pins).

There is no requirement that the structure hierarchy be a tree. In general it could have a lattice structure. Thus it would be possible to add the alternative decomposition just described into the tree shown in figure 1 by adding the fixed-parts substructure as an additional child of the doorway structure. Structures like "MULTISCREW1.1" would then be children of the fixed-part substructure as well as their doorhinge structures. For simplicity, however, this paper will generally assume hierarchies are trees, although the methods extend easily to lattices. We will also assume that whenever a structure is decomposed, each of its parts will appear in at least one of its child structures.

It is important to note that the structure hierarchy does not give a geometric description of the assembly, and is not intended as a substitute for a geometric model. It describes only the semantic structure of the assembly. To produce correct plans, additional geometric information is required, primarily in the form of knowledge about which part motions intersect with which other parts.

Views of assemblies similar to this have received some study for use in function-based design systems[26], and for function-based approaches to manufacturing cost estimation[13], but the only work which has been done on studying their applicability to computer-aided process-planning tasks was a paper describing a previous version of the HAP planner[4].

3.2 Structure Library

Assembly structures can be classified by type. While some structure types may be unique to a particular assembly, many appear repeatedly in a wide variety of assemblies. Descriptions of such common structures should be stored in a *structure library.*

One type of information that may be stored for a common structure type would be a description of its substructures. Having this would mean that when the user inputs the structure hierarchy for a new assembly, he need not describe those substructures. Thus if the structure library contains descriptions of a doorhinge structure, it would only be necessary to specify the hierarchy down to that point. The rest would be pulled in from the library. In fact, if a company builds many products with doorways, it is likely that it will have the entire doorway as a standard structure in its structure library. Thus, although the complete structure hierarchy for a given assembly may be very large, generating it would not be as burdensome as it might initially appear if a structure library well-stocked with appropriate structures is available.

The other key type of information that is stored in the structure library is advice about how to build structures of that type. For most structures, there are common, preferred ways to build them. For example, there are standard procedures for hanging a door. Certainly the planner could discover its own plan by reasoning about the low-level interactions among the parts of the doorway, but not only does this take a great deal of computation, but the resulting plan is almost certain to be inferior to the one used by expert door-hangers. The knowledge about the "goodness" of a plans captured by the plan evaluation functions will always be incomplete and can never match years of manufacturing experience with common structures. Thus, the addition of assembly advice to the structure libraries will have a dual impact—it will allow us to plan faster, and it will allow us to find better plans.

Ideally the structure types in the library would be arranged in an ISA-hierarchy, where more specific types of structures, such as screws, would be identified as instances of more general types, such as fasteners. This would allow us to give both general advice applicable to broad classes of structures, and more specialized advice for more specific structures. This would allows us to represent "goodness" information that is not quite general enough to be easily representable as an general evaluation function, but still applies to fairly broad classes of assembly structures. For example, an electrical circuit structure might have advice suggesting that the power source be attached last. This could be applied to tell us to finish all the wiring before connecting the battery in a car or before plugging in a lamp.

The HAP system described in this paper, however, supports only a limited form of very specific advice. This takes the form of a list of suggested plans for building the structure.

Every structure in HAP must have either a list of suggested plans, a complete decomposition into substructures, or both. Structures without substructures are called *primitive structures*. The most common primitive structures are those simple two-part contacts corresponding to liaisons. For HAP to be guaranteed to find a plan if one exists, the lists of suggested plans for primitive structures must include all possible plans. This is generally not difficult to achieve, since most primitive plans involve very few parts, so there are very few distinct ways to build them. If a structure has both substructures and suggested plans, then HAP will try first to use the suggested plans, and then, only if they are for some reason unusable, attempt to generate its own plans.

3.3 Bottom-Up Planning

The majority of previous assembly planners plan backwards in time, by first deciding which assembly operation is to be done last, and then continuing to work backwards from there until finally the first operation is specified. This can be viewed, somewhat inaccurately, as generating a disassembly sequence by repeatedly separating the assembly into two subsets, and then reversing the procedure it to achieve an assembly sequence. The planners described by Homem de Mello and Sanderson[9, 10, 19], by Lee[14], and by Wilson[21] are typical of this type approach. If the only concern is the geometric feasibility of the assembly (*i.e.*, that parts don't intersect) then this approach has the important advantage that the planner never has to backtrack. This makes very fast non-optimizing planners possible.

But this advantage vanishes when the goal is to optimize with respect to some reasonably complex goodness function. Backtracking becomes necessary, and because it is very difficult to decide in isolation which operation should be done last, it becomes difficult to create good heuristics functions to guide the search away from blind alleys.

The HAP system does not make decisions in any fixed temporal sequence at all. Instead, it plans bottom-up in the structure hierarchy. First plans are generated for the lowest-level structures, and then they are merged to get plans for higher-level structures. It is possible to ensure that the plans make good sense locally, among the other parts involved in the same substructure, before broadening the view to higher-level structures. Working from local structures up to higher-level structures would appear to be a more natural order for making decisions than working from the last operation back to the first operation. If this is true, less backtracking might be expected, since early decisions might be made on a sounder basis.

The need for backtracking, however, is certainly not eliminated. In order to ensure that the planner satisfies the completeness requirement, which, in a nutshell, states that if a valid plan exists then a valid plan can be generated by the planner, a backtracking strategy is adopted, where all possibilities can be explored, if necessary.

3.4 Sub-Optimization

It cannot generally be presumed that merging a set of plans, each of which was a good plan for some substructure, will always give a good plan for the higher-level structure. Thus, as the planner works up each level of the structure hierarchy, it will consider several different methods of combining lower-level plans to get a plan for the current structure. To choose among these alternatives, it uses general purpose numerical evaluation functions to rate the alternatives. These are similar to the evaluation functions used in previous planners. HAP currently uses an evaluation based on a weighted combination of three criteria: attachment, directionality, and manipulability.

It would be possible to do a full optimizing search to generate the best possible way to combine the plans from the child structures into a plan for the parent structure. But it is not at all clear that this would be desirable. First, such searches are very expensive, typically requiring exponential time. Second, the values returned by the evaluation functions do not correspond closely enough to real manufacturing costs to make optimizing them to the last decimal place very meaningful.

When HAP generates plans for a substructure, it generally returns more than one candidate plan, but it does not, at first, return all possible plans. The maximum number of plans produced is determined by the *planning window size*, W, a user-settable parameter. Further plans will be generated for the child structure only if none of the first W plans could be used to generate any valid plan for the parent. If any are usable, then the best of them will be passed upward without further inspection of child plans.

Clearly this method does not necessarily generate optimal plans for a structure. If the best plan for a structure involves a child plan which exists, but is not in first W plans considered for that child, then that best plan will not be found. In other words, even though what are thought to be the best W plans for each child have been found, it still is not clear that the best plan for the overall structure can always be merged from them. Increasing W will increase the chances of finding the optimum, but will also slow down the system. Increasing W to infinity would convert the system to a fully optimizing system.

4 Plan Generation

The heart of the HAP planning method is the idea of merging plans for child structures to generate plans for the parent structure. This section will first describe the representation used for plans, and then describe the two basic ways to merge plans. Not all pairs of plans can be merged. Some merges are logically invalid, producing results that do not satisfy the definition of a logically valid plan. Other merges are logically valid, but are geometrically infeasible because they would cause parts to intersect each other. A simple method for integrating geometric constraint checking into the planner will be presented. Finally it will be proven that the merge operations described are, with some slight restrictions, correct and complete.

4.1 Representation of Plans

Plans to build a product can be given at different levels of abstraction. The greater the detail provided, the less abstract the plan is. Assembly plan representations generally provide only a partial description of the assembly task and thus many possible assembly processes could be considered valid executions of that plan. The choice of which details should be decided at the assembly planning stage, and should thus be included in the plan representation, is a critical one.

If too little detail is included in the plans, then they may become difficult to evaluate. Homem de Mello and Sanderson's assembly trees contained just enough information to determine geometric feasibility[9]. But since the planner did not decide which set of parts was held and which set of parts were moved in mating operations, criteria such as fixture complexity and directionality could not easily be included.

If too much detail is included in the plans, then the planning process may become slow because many plans that are essentially equivalent must be treated as different alternatives by the planner. Wolter's XAP/1 planner always generated plans in which the insertion operations were totally ordered, even when the ordering information made no difference to any of the the criteria used to evaluate plans. Thus, when given two plates held together by a n nuts and bolts, it required exponential time because it was effectively searching all permutations of the order of insertion of the bolts in hopes of finding one that was better than the others[23]. De Fazio and Whitney's planner also always generated totally ordered plans, thus increasing the difficulty of selecting the best plan because many essentially identical plans appeared as different alternatives[1].

Therefore it is important to try to define plans at a level of abstraction which is detailed enough so that the quality of the plans with respect to a certain set of criteria can be assessed, but is not so detailed as to make the planning task extremely slow. Clearly, this choice depends critically on the criteria set, and some care must be taken to avoid selecting a set of criteria so large that a fully detailed plan would be needed.

Before defining the plan representation, some basic terminology needs to be established. A *part* is a discrete object available to the assembler as in input. The set of all parts will be denoted by $\mathcal{P}$. An *assembly* or *subassembly* is a set of two or more parts which are placed into their final relative goal positions in the course of the manufacturing process and then, if it is not the final assembly, inserted as a unit in another assembly. Each assembly or subassembly is assumed to be constructed in its own fixture. The term *component* will be used to refer to the parts and subassemblies that are separately inserted to construct a particular assembly.

Let Θ be the set of trajectories applied to parts by the plans the structure library gives for structures in the assembly. This would include motions, among many others, normal to planar contacts and parallel to the axis of cylindrical contacts. Note that any trajectory given for any part p_1 can potentially be applied to any other part p_2. Most commonly this happens if parts p_1 and parts p_2 are in a subassembly together and so are moved as a unit. In general, trajectories are best thought of as classes of motions rather than specific motions.

The plan representation used by HAP exactly identifies all subassemblies. A subassembly is a set of parts built in one fixture and then (unless it is the final assembly) inserted as a unit into another assembly. Knowing this is critical for criteria relating fixture design and layout, including

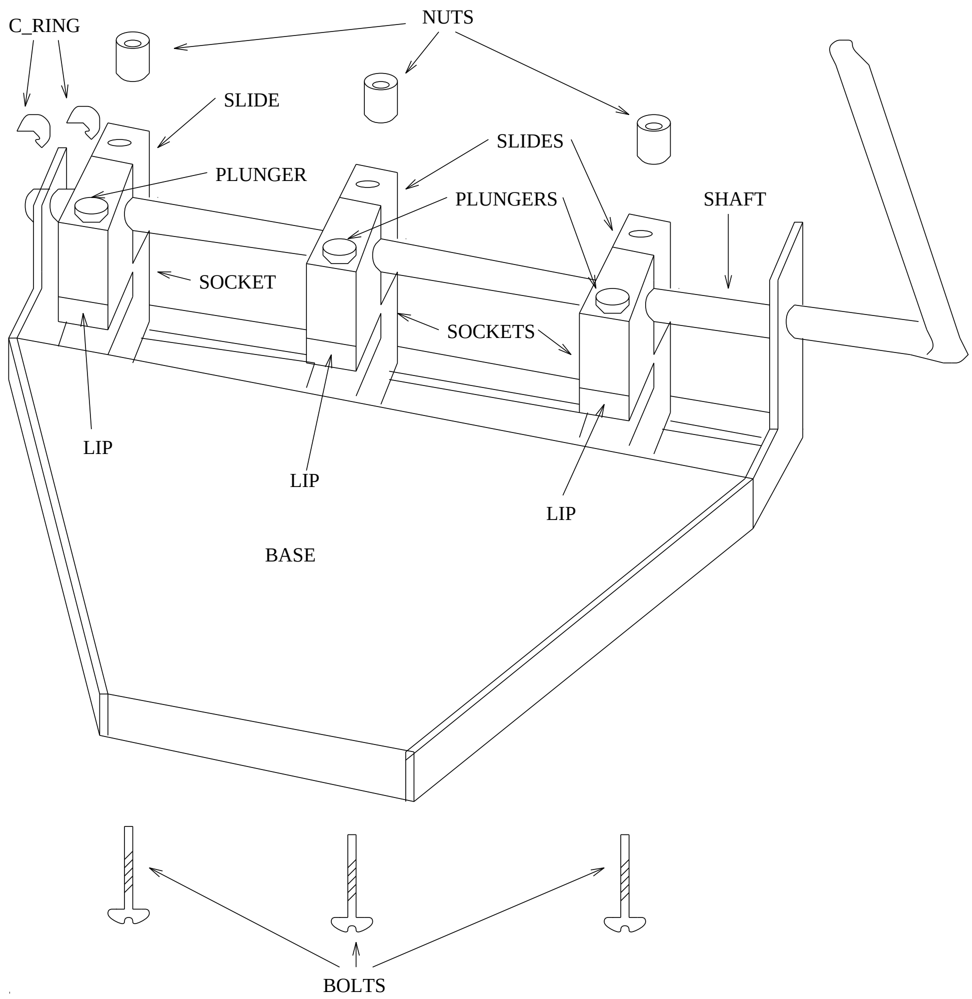

Figure 2: A hole-punch assembly.

the number of different directions parts are inserted from within any given fixture, and for evaluating the stability of subassemblies that will be handled.

Information about the sequence in which operations are performed is sometimes vitally important, and sometimes completely irrelevant. HAP plans are partially ordered, so as much sequencing information can be included as is necessary, without any more being required. HAP also allows some flexibility on the designation of insertion trajectories, since either a unique trajectory can be specified, or the trajectory can be left unspecified.

Definition 1 *A* partially-ordered insertion sequence *(POIS) to build an assembly $P \subseteq \mathcal{P}$ describes a top-level plan to construct that assembly. It is represented by a three-tuple $I = \langle \mathbf{C}, \Rightarrow, \mathbf{T} \rangle$, where the set $\mathbf{C} = \{C_1, C_2, ..., C_n\}$ is a partition of P into a set of two or more components each a set containing one or more parts; the relation $\Rightarrow \subseteq \mathbf{C} \times \mathbf{C}$ is the transitive closure of a partial ordering*

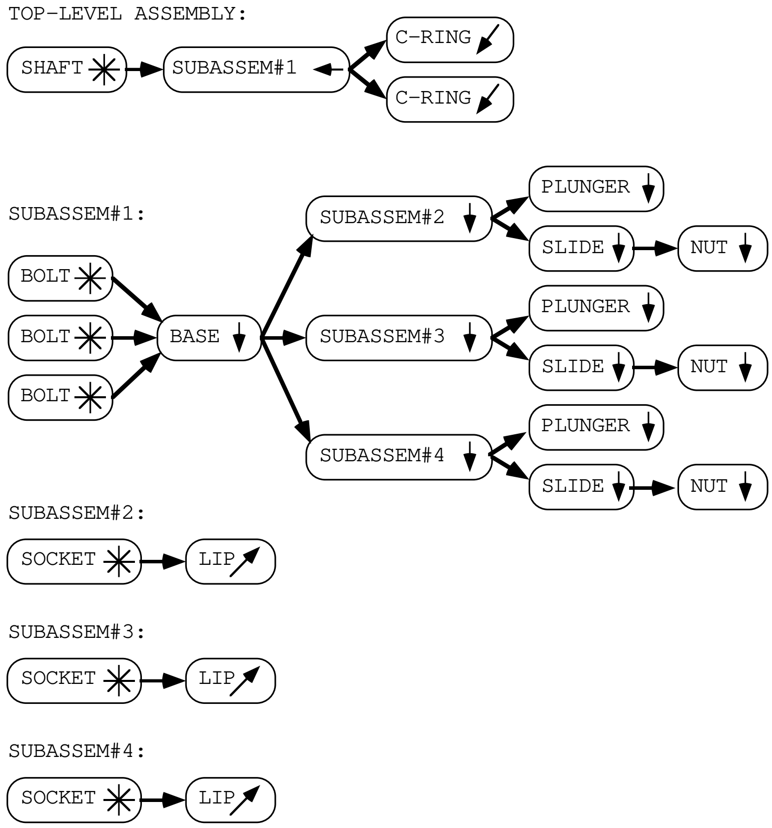

Figure 3: A POIS hierarchy representing a possible plan for building the hole-punch assembly shown in figure 2. Each node of the graph represents an operation inserting a part or subassembly along some trajectory. A star indicates that (almost) any trajectory may be used. The nuts are actually inserted from above with a screwing trajectory, distinct from the other from-above trajectories.

such that $C_i \Rightarrow C_j$ *means component* C_i *must be inserted before component* C_j*; and* $\mathbf{T} : \mathbf{C} \rightarrow \Theta^+$ *is a function assigning a non-empty set of possible trajectories to each component.*

A *POIS hierarchy* to build an assembly P describes not only the top-level POIS plan to build P, but also describes how to build each subassembly. It is represented by a finite set of POIS plans such that there exists exactly one POIS plan in the set which builds P and if there is a POIS plan in set which includes a subassembly C, then there is exactly one POIS plan in the set which builds C.

Figure 2 shows the example of an assembly which can be constructed by following the plan shown in figure 3. Note that although only the basis is drawn in the figure, the sequencing relation of a POIS plan includes the full transitive closure of the relation. A POIS can represent more than one sequence to build a subassembly. A POIS merely denotes a set of possible sequences that are valid in that they are geometrically feasible. Thus, in the top level of the plan shown, the C-RING parts may be inserted in either order.

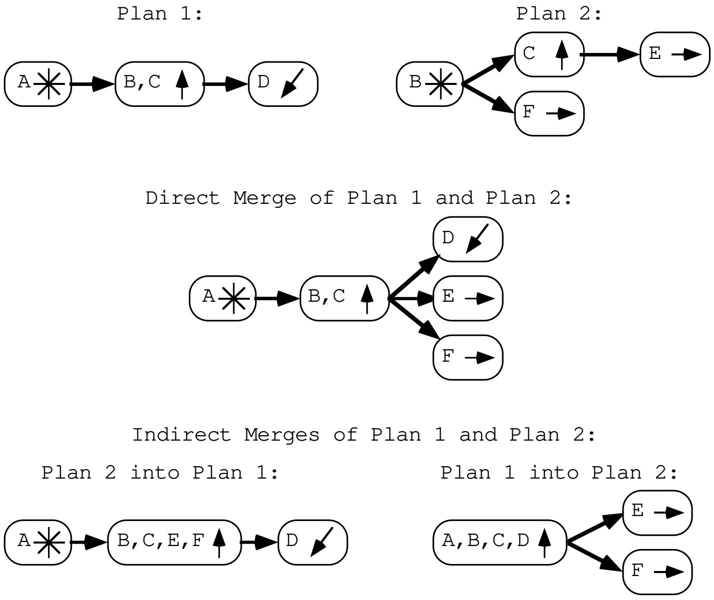

Figure 4: Examples of successful direct and indirect merges of two plans. Trajectories for parts are indicated as in figure 3.

Since POIS plans can represent more than one plan, it is possible for one POIS plan to include another. Plan $I = \langle \mathbf{C}, \Rightarrow, \mathbf{T} \rangle$ is defined to *include* plan $I' = \langle \mathbf{C}', \Rightarrow', \mathbf{T}' \rangle$ if and only if $\mathbf{C} = \mathbf{C}'$; $\Rightarrow \subseteq \Rightarrow'$; and $\mathbf{T}(C) \supseteq \mathbf{T}'(C)$ for all $C \in \mathbf{C}$. That is, the included plan has the same components, no fewer sequencing constraints, and no more proposed trajectories.

The HAP planner first finds a partially-ordered insertion sequence to build the final assembly. After this is finished, it proceeds to recursively find plans for all subassemblies required by this plan. Most of the discussion in this paper will focus on the production of the top-level POIS plan, since the same method is used to to produce all other plans in the POIS hierarchy. Thus, the word "plan" will generally be used to refer to a single POIS plan, rather than to the complete POIS hierarchy.

It is important to note that the POIS hierarchy of the plans and the structure hierarchy of the input data structure are distinct. There is no necessary correspondence between the levels of the structure hierarchy and the levels of the POIS hierarchy.

4.2 Merging Plans

Combining plans for two different part sets to create a plan for the union of the part sets is called merging. Two kinds of merges are possible: *direct* and *indirect*. Indirect merges involve introducing a new subassembly to the plan. Direct merges do not, though they may combine existing subassemblies.

Let there be two POIS plans $I' = \langle \mathbf{C}', \Rightarrow', \mathbf{T}' \rangle$ and $I'' = \langle \mathbf{C}'', \Rightarrow'', \mathbf{T}'' \rangle$. If there are components

$C'_i \in \mathbf{C}'$ and $C''_j \in \mathbf{C}''$ which share at least one part (*i.e.*, $C'_i \cup C''_j \neq \emptyset$), then those components are called *overlapping* and the part is called a *shared part.*

Definition 2 *The* direct merge *of two POIS plans $I' = \langle \mathbf{C}', \Rightarrow', \mathbf{T}' \rangle$ and $I'' = \langle \mathbf{C}'', \Rightarrow'', \mathbf{T}'' \rangle$. is a POIS plan $I = \langle \mathbf{C}, \Rightarrow, \mathbf{T} \rangle$. Here the component partition, $\mathbf{C}$, is the sum of the original two partitions, $\mathbf{C}' + \mathbf{C}''$. This is constructed by concatenating the two partitions and then unioning together any sets of components which overlap[15]. The sequencing relation, $\Rightarrow$, is defined so that $C_i \Rightarrow C_j$ if and only if plan I' contains components $C'_i \subseteq \mathbf{C}'$ and $C'_j \subseteq \mathbf{C}'$ such that $C'_i \Rightarrow' C'_j$ or plan I'' contained components $C''_i \subseteq \mathbf{C}''$ and $C''_j \subseteq \mathbf{C}''$ such that $C''_i \Rightarrow'' C''_j$. Similarly, the trajectory assignment function $\mathbf{T}$ is defined so that $\mathbf{T}(C)$ is the intersection of the trajectory sets of all subsets of C from both I' and I''. That is*

$$\mathbf{T}(C) = \left(\bigcap_{C' \subseteq C, C' \in \mathbf{C}'} \mathbf{T}'(C') \right) \cap \left(\bigcap_{C'' \subseteq C, C'' \in \mathbf{C}''} \mathbf{T}''(C'') \right)$$

An example of a direct merge is shown in figure 4. In this example, component $\{B, C\}$ from plan 1 overlaps components $\{B\}$ and $\{C\}$ from plan 2, so all of these are grouped together in the merged plan.

Clearly not every pair of plans can be directly merged. Three conditions must be satisfied for a merge to be logically valid. First, $\mathbf{C}$ must contain more than one element, so there should not be chains of overlaps tying all components together. Second, the $\Rightarrow$ relation must be acyclic. And third, for the trajectory assignment function $\mathbf{T}$ to give a non-empty set of trajectories, all sets of overlapping components must share at least one trajectory.

Definition 3 *The* indirect merge *of the two POIS plan $I'' = \langle \mathbf{C}'', \Rightarrow'', \mathbf{T}'' \rangle$ into POIS plan $I' = \langle \mathbf{C}', \Rightarrow', \mathbf{T}' \rangle$ is a POIS plan $I = \langle \mathbf{C}, \Rightarrow, \mathbf{T} \rangle$ where I'' is simply created as a subassembly of I'. Thus the component partition, $\mathbf{C}$, is the sum of the component partition of plan I' with the partset of plan I'', that is $\mathbf{C}' + \{\mathrm{PartSet}(I'')\}$. Thus the entire part set of plan I'' forms a new subassembly, with any components from $\mathbf{C}''$ that overlap it unioned in. The sequencing relation, $\Rightarrow$, is defined so that $C_i \Rightarrow C_j$ if and only if plan I' contained components $C'_i \subseteq \mathbf{C}$ and $C'_j \subseteq \mathbf{C}$ such that $C'_i \Rightarrow_1 C'_j$. Similarly, the trajectory assignment function $\mathbf{T}$ is defined so that $\mathbf{T}(C)$ is the intersection of the trajectory sets of all subsets of C from I'. That is*

$$\mathbf{T}(C) = \left(\bigcap_{C' \subseteq C, C' \in \mathbf{C}'} \mathbf{T}'(C') \right)$$

Examples of indirect merges are shown at the bottom of figure 4. However, we are concerned at this point only with the top level plans. Note that merging plan I' into plan I'' does not give the same result as merging plan I'' into plan I'. Each of the possible resulting plans have a subassembly that could have been built with the original POIS of the plan that was merge into the other. As far as the top-level POIS plan is concerned, it does not matter which plan for a particular part set is indirectly merged in. Only the contents of the part set constructed by the plan is significant in an indirect merge.

In the unusual case where PartSet(I'') doesn't overlap any components of I', then the new plan will be just like I' with the component PartSet(I'') added. The trajectory set for this component will be Θ, the set of all trajectories, and no new sequencing relations will be added.

The conditions for an indirect merge to be logically valid are similar to those for the direct merge case. The resulting plan must contain at least two components, so the I' must have included at least one component containing no parts in I''. The $\Rightarrow$ relation must be acyclic, and the components of the I' which overlapped the part set of I'' must all share at least one trajectory so that $\mathbf{T}(C)$ will never be an empty set.

4.3 Representation of Structure Hierarchy

For simplicity, the structure hierarchy for an assembly will be represented in this paper by a collection of structures $\mathcal{H} = \{H_1, H_2, \ldots, H_m\}$ where each structure is represented by the set of parts $H_i \subseteq \mathcal{P}$ involved in that structure. For each structure H_i we will assume that there is a corresponding set of suggested POIS plans $\mathbf{I}_i$ to build it. This set may be empty. In actuality, each structure has a type, which is used to access the list of suggested plans from the structure library and multiple distinct structures may involve the same set of parts, but that level of detail is unnecessary for this paper.

The structures in $\mathcal{H}$ are naturally arranged into a partially ordered hierarchy by the subset relation. The subsets of a structure will be called its descendent structures, and the immediate descendents are called its child structures. For $\mathcal{H}$ to be a valid structure hierarchy, the following conditions must hold:

- The set of all parts $\mathcal{P}$ is in $\mathcal{H}$. Thus, there is one structure, the root of the hierarchy, that includes all other structures and represents the entire assembly.
- If any structure $H_i \in \mathcal{H}$ has children, then the union of sets of parts in those children is H_i. Thus, though the children will usually *not* be disjoint sets, they do cover the parent set.
- If any structure $H_i \in \mathcal{H}$ does not have children, then its suggested plan set, $\mathbf{I}_i$, is non-empty. Thus, all leaves, which represent primitive structures, do have suggested plans.

To simplify the discussion, this paper will also presume that the hierarchy is a tree, not a general lattice. This is not a requirement for the HAP system however and the extension is straight-forward.

Note that the suggested plans stored for each structure are POIS plans which can express only top-level plans for building the structure. Thus they identify any subassemblies to be used to construct the structure, but do not give advice on how to build the subassemblies. This is a deliberate feature. Suggested plans for building the subassemblies should be attached instead to a structure whose part set corresponds to the set of parts in the subassembly. The choice of the best plan to build a subassembly is usually quite independent from the choice of how that subassembly will be inserted into a higher-level assembly. If there are three ways subassembly C can be used in building an assembly A, and there are five ways to build C, then it is better to list them separately than to list the fifteen combinations that could be used to build A.

To generate a top-level POIS plan from a structure hierarchy, we can first select a *structure basis*, $\mathbf{H}_B \subset \mathcal{H}$ from the hierarchy. Every path from the root to a leaf should pass through at least one basis structure, and every basis structure must be one for which there are suggested plans. Then we can select a *plan basis*, $\mathbf{I}_B$, by selecting one suggested plan from each basis structure. Finally we decide for each plan in the plan basis whether it is to be merged into the parent plan directly or indirectly. To merge a set of plans, we start with a null plan and merge in each of the plans in the set. The order in which the merges are done makes no difference to the result. Later we will describe a detailed algorithm for doing this.

Theoretically it is possible to merge in all plans from the basis indirectly, in which case we would get a top-level plan with no sequencing constraints. This could be valid in some cases. Since only a limited set of trajectories is considered it might be possible to insert components in any order without interference. For most realistic structure hierarchies, however, this would not lead to a logically valid plan, because there would be shared parts tying together the substructures so that the top-level plan would have just one component.

4.4 Geometric Constraints

The merge operations described in the previous section do not check for the geometric feasibility of the insertion operations performed. Since the structure hierarchy contains no specific geometric information about the shapes of the parts, additional information in the form of assembly insertion constraints is required.

Assembly insertion constraints have been used by many assembly planners, including Wolter[23], Huang and Lee[12], and Wilson[21]. They are usually seen as taking the form:

If part p_1 is inserted along trajectory θ, then p_1 must be inserted before p_2.

Such constraints most commonly arise when moving part p_1 along path θ would cause it to intersect with p_2.

Rather than introduce a separate mechanism to enforce insertion constraints, they are integrated into the merging algorithm. As a preprocessing step, each part in $\mathcal{P}$ would be tested against all the rest using every trajectory in the proposed trajectory set Θ. If moving part p_1 along trajectory θ is blocked by part p_2, then a primitive structure called a *blocking primitive structure* with part set $\{p_1, p_2\}$ is added to the hierarchy. The list of suggested plans for the blocking primitive structure would allow any plan except inserting part p_1 by trajectory θ after part p_2, or inserting part p_2 by trajectory $-\theta$ after part p_1.

When this blocking primitive structure is added into the hierarchy, it normally becomes a child of the deepest structures containing both p_1 and p_2. If any ancestors of the blocking primitive have non-empty lists of suggested plans, then the plans for the blocking relation should be direct merged with each of those plans, and any plans which become logically invalid as a result of this should be eliminated.

Blocking primitives are thereafter treated like any other primitive structure in the hierarchy. When they are merged in during the normal course of plan generation, they will cause all geometrically infeasible plans to be eliminated. This will be proven in the following section.

4.5 Correctness of Merges

This section will show that constructing plans by merging subplans leads to correct plans. A plan generation algorithm is considered *correct* if any plan it generates for any possible subassembly is correct. A plan to build an assembly is correct if any sequence of part insertions and trajectories that satisfies the POIS hierarchy representing the plan results in the product being built, without collisions between parts. Since lower-level plans are generated by the same algorithm, it will suffice to show that all top-level POIS plans produced are correct.

Theorem 1 *Let $\mathcal{H}$ be a structure hierarchy that includes all blocking relations as described in the previous section. Let I be a logically valid plan produced by merging some plan basis $\mathbf{I}_B$ taken from a structure basis $\mathbf{H}_B \subset \mathcal{H}$. Then I will be a correct plan.*

Proof: Suppose I is incorrect. Then there must be some part p_1 which collides with another part p_2. But if such a collision is possible, then $\mathcal{H}$ must include a blocking primitive $H' = \{p_1, p_2\}$ for it, since it includes blocking primitives for all possible intersections. Let $\mathbf{I}'$ be the plan set for H'. Either the structure H' itself, or some ancestor of the structure must be in the structure basis $\mathbf{H}_B$, since for every leaf, either the leaf or one of its ancestors must be in any structure basis. Since the plans for blocking primitives are merged into all their ancestors, this means that the plan basis $\mathbf{I}_B$ must include either a plan from $\mathbf{I}'$ or a plan that has already been merged with a plan from $\mathbf{I}'$. Since parts p_1 and p_2 must be inserted separately if they are colliding, we know that the plan must have been directly merged. Yet the trajectory set assigned to a part is the intersection of all the trajectory sets assigned to it in directly merged plans in $\mathbf{I}_B$, and no sequencing relation between two parts in plan I can exist if it is contradicted by any directly merged plan in $\mathbf{I}_B$. Thus, if a collision is possible between parts p_1 and p_2 in plan I, then it must also have been possible in every one of the directly merged plans in the set $\mathbf{I}_B$. But one of those plans must be from the blocking primitive structure, which includes no such plans. Thus, by contradiction, the theorem is proved. $\square$

4.6 Completeness of Merges

A method of plan generation is considered *complete* if every possible geometrically valid plan that results in the product being built can be generated by that method. To show our method is complete, it would be necessary to show that every possible plan for an assembly can be found by doing some combination of direct and indirect merges on some plan basis taken from the hierarchy. In fact, the method falls short of completeness, in that it cannot generate arbitrary subassemblies. Only subassemblies which are unions of overlapping structures can be constructed.

As previously defined, two structures overlap if they share at least one part. A set of structures $\mathbf{H}$ is overlapping if and only if any two structures H_i and H_j in $\mathbf{H}$ are connected by some chain of overlapping structures, all in $\mathbf{H}$. For every subassembly in any plan that can be produced by the HAP planner, there must exist some set of overlapping structures, $\mathbf{H}$, in the input structure hierarchy whose part sets union to the parts in the subassembly. No plan containing a subassembly not of that form can be produced by the planner.

This limitation is not a serious one in practice, since such plans would rarely, if ever, be considered to be good plans. If we had a non-overlapping subassembly, then it would be possible to divide it into two disjoint sets of parts which would be involved in no structure relations with each other. Since normally the set of structures includes all contacting pairs, this would be a single subassembly consisting of two or more disconnected sets of parts. While it is quite common to have a temporarily disconnected set of parts in a fixture (and HAP allows this), inserting such a set as a single unit into a subassembly would be extremely unusual.

To prove completeness, we will need the concept of an induced subplan. Suppose I is a plan and P is a set of parts. Then, I_P, the subplan *induced* from a plan I by a set of parts P, is constructed by deleting from each component of I all parts not in P, and then eliminating any empty components and restricting the sequencing relation and the trajectory assignment function to the remaining set of components.

Lemma 1 *If the POIS plan $I = \langle \mathbf{C}, \Rightarrow, \mathbf{T} \rangle$ represents a correct top-level plan to build an assembly with part set P and P' is a subset of P which includes parts from more than one component of plan I, then the induced subplan $I_{P'} = \langle \mathbf{C}', \Rightarrow', \mathbf{T}' \rangle$ represents a correct top-level plan to build P'.*

Proof: Given that P' is not a subset of just one component of I, it is clear that the plan $I_{P'}$ will be logically valid. We will prove by contradiction that it is also correct. Suppose there is an incorrect sequence included in $I_{P'}$. Then some component $C'_1 \in \mathbf{C}'$ must collide with some component $C'_2 \in \mathbf{C}'$ when moved along some trajectory $\theta \in \mathbf{T}'(C'_1)$. There must be components C_1 and C_2 in the original plan I such that $C'_1 \subseteq C_1$ and $C'_2 \subseteq C_2$ because each element of $\mathbf{C}'$ was derived from an element of $\mathbf{C}$ by deleting parts. The trajectory set for C_1 must include θ. There can be no sequencing relation $C_1 \Rightarrow C_2$ in the original plan I because then there would have been a sequencing relation $C'_1 \Rightarrow' C'_2$ in $I_{P'}$ and the collision would not have been possible. But if C'_1 collides with C'_2, and we can use the same trajectory and sequencing, then the larger components C_1 and C_2 must also collide. This contradicts the assumption that plan I was correct. Thus plan $I_{P'}$ must be correct. $\Box$

Using this lemma, we can prove the following lemma, which shows that all plans without non-overlapping subassemblies can be generated for a structure with two children by merging child plans.

Lemma 2 *Let H be a structure with exactly two child structures, H_1 and H_2, where $H_1 \cup H_2 = H$. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be complete sets of plans without non-overlapping subassemblies for H_1 and H_2 respectively. Then if $I = \langle \mathbf{C}, \Rightarrow, \mathbf{T} \rangle$ is a correct plan for H without non-overlapping subassemblies, it will be possible to create I, or a plan including I, by merging a plan from $\mathbf{I}_1$ with a plan from $\mathbf{I}_2$.*

Proof: Let I_{H_1} and I_{H_2} be the subplans of I induced by part sets H_1 and H_2 respectively. It the plans I_{H_1} and I_{H_2} contain more than one component, then they are both correct (by lemma 1) and contain no non-overlapping subassemblies (since I does not), so they must be in the sets $\mathbf{I}_1$ and $\mathbf{I}_2$ respectively.

We consider four cases.

Case 1: If I_{H_1} and I_{H_2} both contain more than one component, then their direct merge includes the plan I.

Since $H_1 \cup H_2 = H$, every component of I will either be identical to a component of I_{H_1} or I_{H_2}, or be the union of a component of I_{H_1} and a component of I_{H_2}. In the latter case, they must have a shared part since I included no non-overlapping subassemblies. Thus the direct merge of I_{H_1} and I_{H_2} will have the same components as I. The trajectory assignments in the merged plan will be identical to I's trajectory assignments, and the sequencing constraints will be a subset of the original, so the merged plan includes I.

Case 2: If I_{H_1} contains only one component, but I_{H_2} contains more than one, then their indirect merge, with I_{H_1} merged into I_{H_2}, includes the plan I.

Let $C \subseteq \mathbf{C}$ be the component of I that includes the entire set H_1. If H_1 is a proper subset of C, then plan I_{H_2} is identical to I, except that there are fewer parts in component C. Indirectly merging plan I_{H_1} will clearly give I. If $H_1 = C$, then plan I_{H_1} will be missing component C, as well as it's trajectories and sequencing constraints. Indirectly merging plan I_{H_1} will give a version of I which is missing the trajectory restrictions and sequencing constraints on component C. Such a plan includes plan I.

Case 3: If I_{H_1} contains more than one component, but I_{H_2} contains only one, then their indirect merge, with I_{H_2} merged into I_{H_1}, includes the plan I.

This is identical to case 2.

Case 4: If I_{H_1} and I_{H_2} both contain only one component, then indirectly merging both plans into a null plan gives a plan which includes I.

This case can happen only in the unusual case where H_1 and H_2 are disjoint and I contains just two components equal to H_1 and H_2. Indirectly merging both I_{H_1} and I_{H_2} will give a plan identical to I but missing all trajectory restrictions and sequencing constraints on the two components. This clearly includes the plan I.

Thus, in every possible case, plan I, or a plan including plan I, can be merged from substructure plans. □

As has been previously said, a set of n plans can be merged by starting with a null plan and merging in plans one at a time. Each plan can be merged in either directly or indirectly.

Theorem 2 *If structure* $\mathbf{H}$ *has child structures* $H_1, H_2, \ldots, H_N$ *and all ways to build* H_i *are given by the POIS set* $\mathbf{I}_i$*,* $(1 \leq i \leq N$ *and* $N > 1)$*, then every correct plan for building* $\mathbf{H}$ *which does not involve a non-overlapping subassembly can be found by merging plans from* $H_1, H_2, \ldots, H_N$.

The proof of theorem 2 is a simple induction using lemma 2, and need not be stated in detail.

5 Search Strategy

The HAP planner has three methods for finding a plan for a structure. They are attempted in the following order:

1. *Similarity:* If the planner has previously found a plan for another instance of this type of structure, reuse that plan.

2. *Advice:* If the structure library gives a list of suggested plans for this structure type, use plans from that list.

3. *Search:* If the structure has child structures, find plans for each of them and merge them into a plan for this structure.

The system, in general, needs to be able to produce many plans for a structure, since it must find one that can be merged with the other structure plans.

If a plan for a similar structure has been generated, then that plan will be tried first (if similarity is enabled). Reusing plans not only speeds up the planning process, but also leads to the production of better plans. It is generally desirable that similar structures be built in the same way in a plan, since this generally requires the ability to perform a smaller variety of operations. Thus, although incorporating the reuse of similar plans into the planner is simple, it is an important feature of the HAP planner.

If similarity does not give rise to a useful plan, the planner next returns any suggested plans for the structure. The use of this form of advice to find better plans faster by including good ways to build standard structures has already been discussed.

Finally, if neither of the previous methods give an adequate plan, a search is undertaken by first making recursive calls to generate plans for each substructure, and then searching for the best way to combine them. Since this phase has been proven complete, it is capable of generating all possible plans, including, among others, any plans found in the first two stages. No checking for this is done, since there are usually only a few redundant plans and small degrees of redundancy are less costly than extra checks.

The following subsections will give a brief overview of the criteria used by the search to select plans, and the search methodology applied. Further details of both may be found in [3].

5.1 Criteria

HAP can use suggested plans to capture knowledge about good ways to build specific structures, but what is good locally may not be as good globally. To capture more global quality criteria, HAP uses numerical evaluation functions similar to those used in previous optimizing planners. Given a plan for building an assembly, these give a numerical rating for the plan.

Currently HAP includes implementations of four criteria functions. *Directionality* counts the number of different insertion directions used to insert parts into a subassembly and favors plans in which operations are done from as few directions as possible. *Manipulability* rates the difficulty of each operation and the manipulability of each part or subassembly, and favors plans in which the more difficult operations are done with the more easily manipulated parts. These first two criteria are essentially identical to those in Wolter's XAP/1 planner[23]. *Attachment* measures how well the subassemblies that are moved hold themselves together, and favors forming subassemblies only from the more stable subsets of parts. Finally, *Similarity* measures how many different plans are used for substructures, and prefers plans where substructures of the same type use the same plans. This last criterion receives some special treatment in the planner as discussed above.

The overall cost function that the planner tries to minimize is a weighted sum of these ratings, with the relative weights selected by the user.

5.2 Search Algorithm

In HAP, plans are generated for the children of a structure and then merged to generate plans for the parent. One way to do this would be to generate exactly one plan for each child structure. This has the advantage that not much time would be spent finding plans for the children, and, if these plans can be merged, then a plan for the parent is found quickly. However the plans may not be mergable, and even if they are, the resulting plan may not be the best overall plan.

In order to find the best possible plan or plans, one could go to the other extreme and get all possible plans for the child structures and then try to search them for the plan basis that merges into the best overall plan. However, the children may have very large numbers of plans and the time to find the best combination may be enormous.

As a compromise between these strategies, HAP recursively generates a user-specified number (called W, the planning window size) of best plans for each child, and then performs an optimizing search for the best way to combine them. If any successful merges are found, then the best W of the resulting plans are passed upward. Only if no mergable combination is found will further child plans are generated for the children, W at a time. Thus, this strategy will always find a plan if one exists, but in choosing the best plan for the structure it will normally only consider a few of the best plans from each child structure. This strategy is not guaranteed to give an optimal plan unless W is made very large, but experience indicates that it can provide a reasonable compromise between plan quality and run time.

Once we have the sets of child plans for each child structure, HAP must be able to find the best way to merge them. To do this, it is necessary to select (1) whether the plan for the child structure will be merged into the parent plan directly or indirectly, and (2) if a direct merge is to be done, then which of the child's plans will be merged in. Thus, if each of n children has W possible plans, then there are $(W+1)^n$ possible ways to form a parent plan, since each of the n children may use either a direct merge of one of its W plans or an indirect merge.

To select the best parent plan with regard to the installed criteria, a simple best-first search is done[16, 17]. HAP decides how to merge in the children one at a time, so that child one is decided at the root of the search tree, child two at the next level, and so forth. If search heuristics were provided for the criteria an A* search would have been possible. However adequate performance was achieved by the test system using the simpler blind search.

If the search finds any plan, up to W of the best are returned. If it finds no valid plan, then the system generates more plans for the children, and tries again. The completeness of the planner is ensured by the fact that it can try all possible combinations of child plans if need be. Based on our previous theorems, it is easy to see that this algorithm is both correct and complete.

Theorem 3 *Assuming that all blocking relations have been included in the structure hierarchy,* *then the HAP planning algorithm described above is correct.*

Proof: Since the algorithm finds a plan basis and merges all the plans in it, this follows directly from theorem 1. □

Theorem 4 *Assuming that all primitive structures have complete lists of plans then the HAP planning algorithm described above is complete for plans without non-overlapping subassemblies.*

Proof: For each structure, the planner can generate and test all possible ways to merge the plans for the child structure. Thus, by theorem 2, it can generate all possible plans with only overlapping subassemblies for the structure. Recursively, this means that it can find all such plans for the root assembly. Thus the planning approach is complete. □

It is important to note that this algorithm finds only the top-level plan for a structure. To generate the full POIS hierarchy, the same algorithm is applied for any and all subassemblies in the top-level POIS plan.

6 Results

This section presents a theoretical complexity analysis of the planner as well as a variety of experimental results. Both types of results are presented much more completely in [3], but they will be summarized here. The HAP planner was implemented in C and all run times are CPU seconds on a Sun 4/60.

6.1 Effect of Structure Hierarchy on Run-time

The form of the structure hierarchy has a significant influence on the performance of the planner. This section presents both analytical and experimental studies of this relationship.

6.1.1 Analytical Results

Before implementing the HAP planner, an analysis was done of the expected time to find a plan with this algorithm. Although many simplifying assumptions were made, the results of the analysis clarify some important characteristics of the effect of the structure hierarchy on the planner's performance, which were later borne out by experimental results.

In this analysis it is assumed that the structure hierarchy is very regular. Every non-leaf structure has exactly n children, and all leaves are at level l. Each primitive leaf structure has a list of t suggested plans. No higher-level structures have any suggested plans at all.

The behavior of the planner is also somewhat simplified in that no automatic reuse of similar plans is considered. The planner's window-size is set to W so that initially only W plans are passed up from each child structure, and the planner will, in the end, produce only W plans. Finally, it is

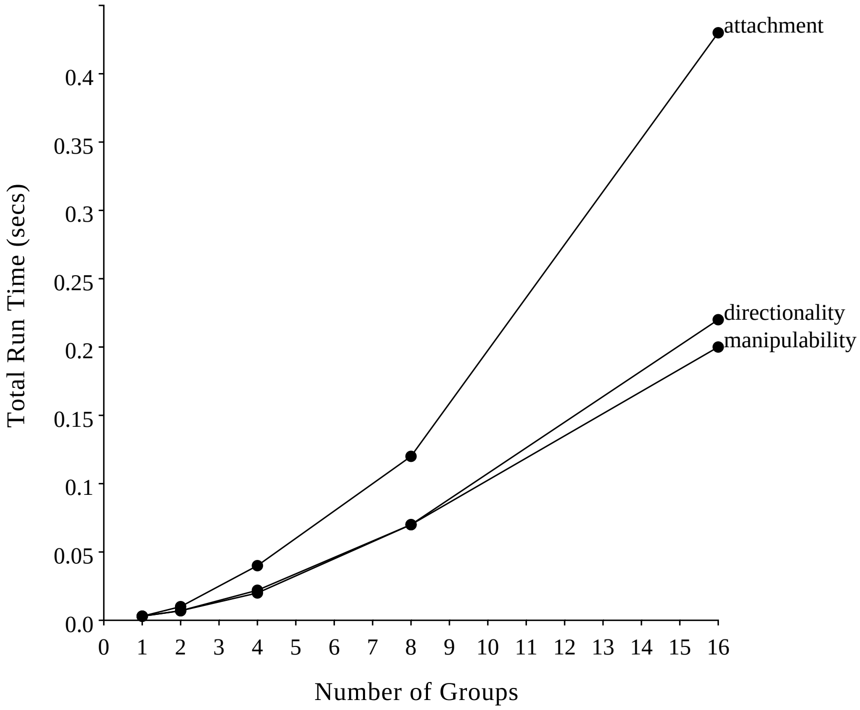

Figure 5: Plot of time vs. number of primitive structures showing that the time to find a plan grows polynomially with increasing number of primitive structures when the number of levels increase, keeping the number of child nodes that a non-primitive node has the same. The three lines show the run-times when the three criteria are the only ones considered by the planner.

assumed that the probability that any given set of n plans from the n children of a structure can be merged is a constant p.

It can be shown that under these assumptions, the order of the expected time to generate W top-level plans for a structure with branching-factor n and depth l is

$$T(n,l) \quad \in \quad O\left(W \cdot z(n)^l + n^l (2W)^n \frac{1-p}{z(n)^{l-1}}\right)$$
$$\text{where } z(n) = \tfrac{n-1}{1-(1-p)^{t/W}} + \tfrac{1}{p}$$

A complete derivation of this is given in [3], but is not included here because of space constraints.

The concern is how the run time is effected by the size of the input. The total number of primitive structures in the tree is $N = n^l$. The total size of the input hierarchy is also in $O(N)$. The parameters p, W and t do not depend on the problem size, and so can be treated as constants.

The effect of increasing the input size, N, on the expected run time, $T(N)$ depends very largely on how the structure of the tree changes when as N increases. Suppose the branching-factor, n, is held constant. Then $l = \log_n N$ as N increases, and $T(N)$ will be a polynomial function in N. This suggests that in families of structures where the number of substructures within any given structure

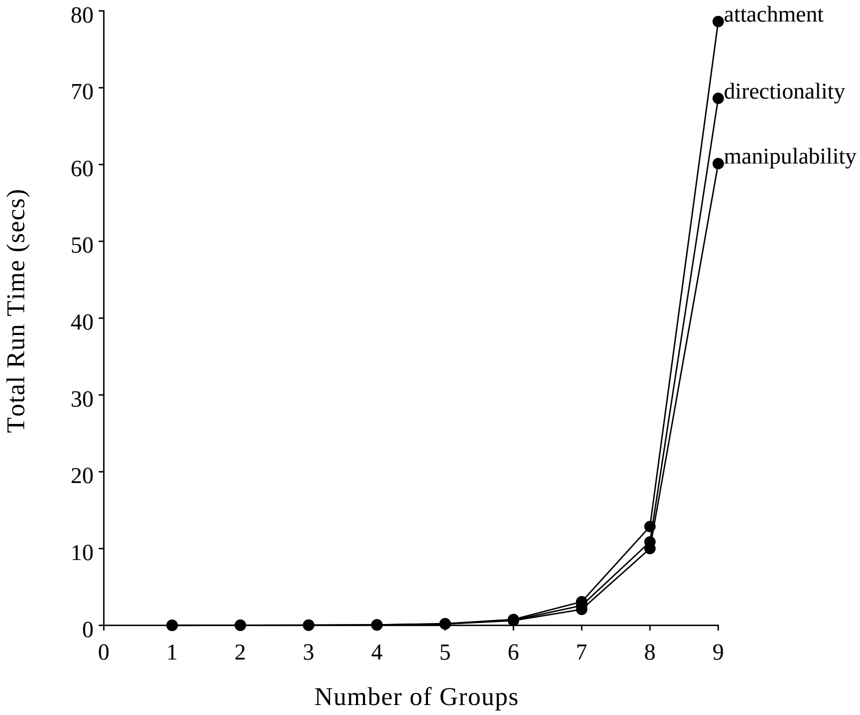

Figure 6: Plot of time vs. number of primitive structures showing that the time to find a plan grows exponentially with increasing number of primitive structures when the number of children that a non-primitive node has increases, keeping the number of levels the same. The three lines show the run-times when the three criteria are the only ones considered by the planner.

remains roughly constant as the complexity of the structure increases, the expected runtime of the planner will be polynomial. The number of times that the top-level planner is run to get a complete POIS hierarchy is bounded by the number of primitive structures in the original structure. Thus it follows that the time to find an overall plan remains polynomial.

On the other hand, suppose the number of levels l in the hierarchy remains constant, and as N increases, the branching factor increases instead. Then the expected run time will be exponential.

The reason for this is that the component of the algorithm which causes exponential expected work to be done is the best-first search for the best way to merge a set of substructure plans to give a plan for a structure. Thus, the expected run time is effectively exponential in the size of the largest structure, rather than exponential in the size of the assembly. The worst case time is exponential in the size of the assembly, but the probability of that is very low.

6.1.2 Experimental Results

To back up these theoretical results, a set of experiments was contrived using an assembly that simply consists of $N + 1$ boxes in a horizontal line. Each box has a face/face contact with its

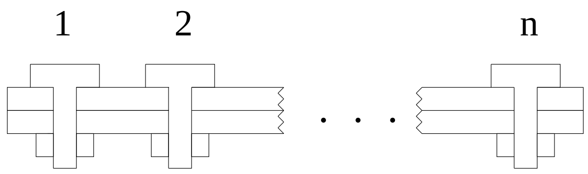

Figure 7: Cross section of an assembly with two plates held by n pairs of nuts and bolts.

neighbors, so there are N primitive contact structures. Parts may simply be removed by lifting them straight up, in which case no part blocks any other, or by sliding them off the ends. This simple assembly was chosen because it is easy to increase the number of parts and any of variety of structure hierarchies can be used, making it easy to simulate the assumptions in the analysis.

As in the analysis, similarity checking in the planner was disabled. Separate runs were done with each of the three criteria functions.

In the first experiment, the hierarchy representing this assembly was varied by keeping the branching factor, n, fixed at 2, while increasing the number of levels, l, in the hierarchy. In this case the analysis predicted that the time to generate a plan would grow polynomially with respect to the number of primitive structures. The run times plotted in figure 5 show that this is in fact the case.

In the second experiment, the hierarchy representing this assembly was varied by keeping the depth of the tree, l, fixed at one, while increasing the number of children of the root node, n. In this case the analysis predicted that the expected time to generate a plan would grow exponentially with respect to the number of primitive structures. The run times plotted in figure 6 show that this is in fact the case.

In this simple assembly, the same plans were found for the assemblies regardless of which hierarchy was used. This would not however be true in general. Since optimization is done only within each structure, breaking the assembly up into smaller structures will lead to less effective optimization. Thus, if the number of substructures in each structure is kept low, the planner will plan faster, but the optimization will be less complete.

6.2 Effect of Similarity

The previous section suggests that the most difficult structures for the HAP planner are those which have large numbers of child structures. In real assemblies, structures with very large numbers of children usually contain many repetitions of the same type of substructure, rather than many different structures. Usually when there are many different structures, they can be naturally classed into logical subgroups and intermediate structure levels result. In this section we study the behavior of the HAP planner on structures with many repeated substructures, and demonstrate that the use of similarity, which was disabled in the previous experiment, can drastically improve performance in this case.

The assemblies used consist of a pair of plates held together by N nut-and-bolt structures.

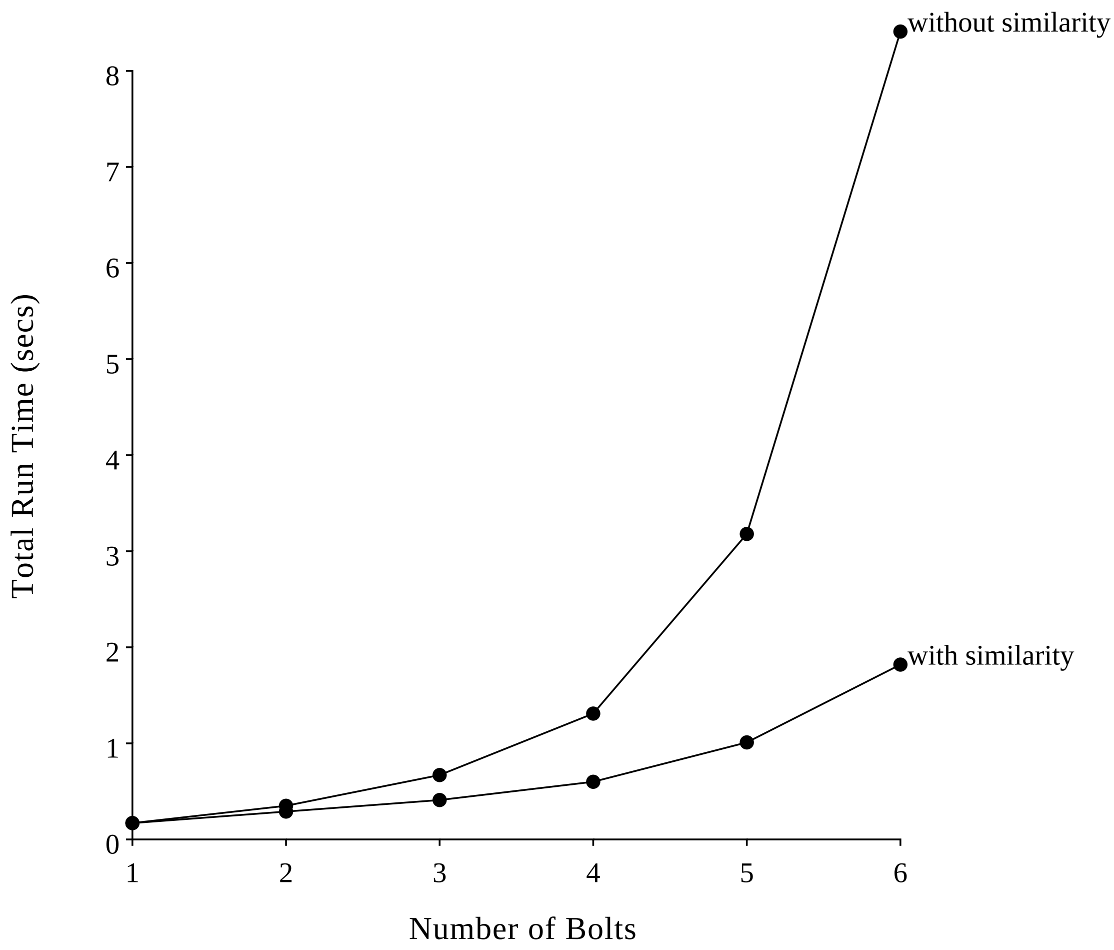

Figure 8: Plot of run time vs. number of nut-bolt structures for a collection of assemblies such as shown in figure 7. When similarity is not enable, run times grow exponentially with the number of structures, but with similarity enabled, polynomial growth results.

An instance of this is shown in the figure in figure 7. The hierarchy consists of a root structure containing N nut-and-bolt structures, each of which consists of 6 simple contacts. No suggested plans are given at the nut-and-bolt level, only at the leaf level. A planning window size of $W = 3$ was used in this example. The manipulability and attachment criteria were used in addition to similarity, but directionality was disabled (enabling it so drastically reduces the search space that the planner runs in linear time on this particular example, which is atypical).

The graph in figure 8 compares the run times with and without similarity. Since the number of levels in the hierarchy does not change as the number of parts is increased, we see exponential growth in the runtimes without similarity, as predicted in the previous section.

When similarity is turned on, the performance of the planner improves significantly. In this case plans do not have to be generated separately for each nut-bolt pair. After the first, the same set of plans is simply reused. This, however, produces a relatively small speed-up in this example, because the similar structures are small ones.

Window Size	Time	Plan
W	(secs)	Rating
2	38.32	78
3	41.50	78
4	51.50	60
5	62.60	60

Table 1: Effect of varying the planning window size on the run time and the rating of the best plan found for the hole-punch assembly shown in figure 2. Only the manipulablity criterion was used.

The more significant part of the speed-up arises because the number of plans that the best-first search will explore is drastically reduced. Since the weight on the similarity criterion was large, the best-first search found a solution without having to expand out the portions of the search tree where different plans were used for different substructures. Note, however, that there is still a possibility of doing exponential work, because different combinations of subassemblies must be considered, however for many combinations of criteria subassemblies are sufficiently unattractive so that only plans without subassemblies will be fully considered. In such cases polynomial time is expected.

Though the speed-up gained by the use of similarity is less dramatic than than in the previous section, it is important because its influence is greatest exactly in the types of structures that cause the greatest difficult to the planner otherwise.

6.3 Effect of Planning Window Size

The planning window size, W, is a user selected-parameter that determines how many plans are initially passed up from a child structure to its parent structure. Since the optimization in the parent structure is only done among these best W plans from each child structure, it is clear that the choice of W will have an important effect both on the run-time and on the quality of the final plan.

In this experiment a more realistic hole-punch assembly was used. The assembly is sketched in figure 2 and a structure hierarchy for the assembly is shown in figure 9. No suggested plans were included for plans above the primitive level, but similarity was enabled. Only the manipulability criterion was turned on.

Table 1 shows the results. Increasing W does, in fact, lead to better plans.

6.4 Effect of Criteria

If the planner is using its criteria effectively, different plans should be generated when the weights of the criteria are selected differently. In this experiment several runs were made of the hole-punch example, shown in figures 2 and 9, with different weights on the criteria. On all the runs shown, the criteria weights were simply set to 0 or 1, enabling or disabling them in various combinations. Similarity was enabled, but no suggested plans were given for non-primitive structures. A fairly

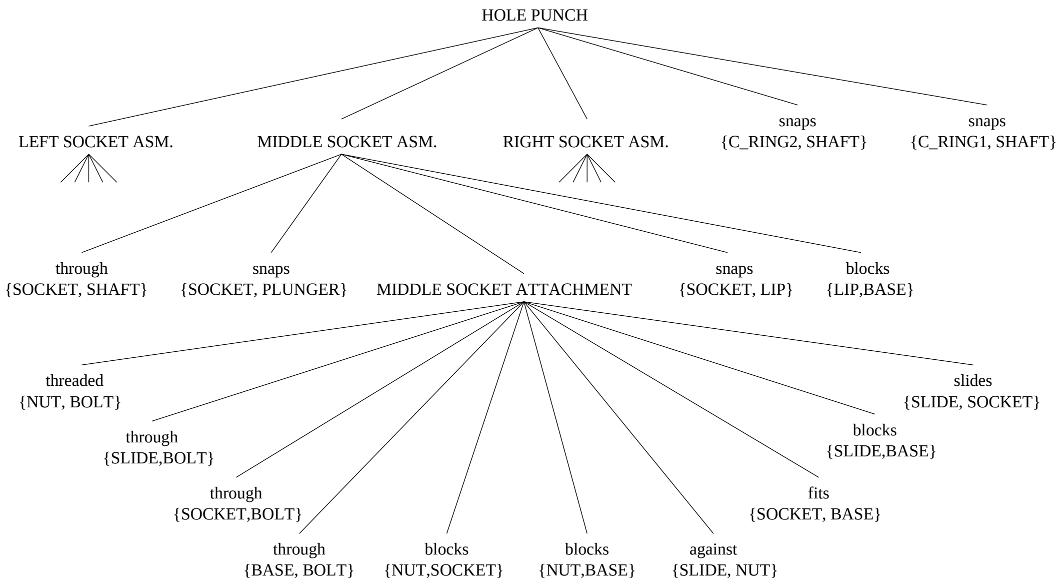

Figure 9: A structure hierarchy for the hole-punch in figure 2.

small planner window size, $W = 2$, was used thus limiting the scope of the optimization done.

Table 2 summarizes the results of this set of runs. Detailed descriptions of each plan can be found in [3]. Every plan included 3 subassemblies in which the LIP parts were fitted into the SOCKETS. Since the BASE blocks the insertion of the LIPS, these subassemblies could only be avoided if the BASE were inserted into three SOCKETS from below after their LIPS had been inserted. This would be strongly discouraged by manipulability because it discourages doing insertions with heavy parts like BASE, and none of the other criteria would favor such a plan. Though manipulability general discourages subassemblies, since subassemblies are usually harder to handle than individual parts, the BASE is much harder to handle than any of the SOCKET/LIP subassemblies. In the case where manipulability is the only criteria, these are the only subassemblies created.

When directionality is the only criterion a plan is generated with many subassemblies, each of which has only one assembly direction. The plan is very unsatisfactory in other ways however. For example, the top level plan consists of first placing a subassembly consisting of all the parts except the three BOLTS, and then screwing in the three BOLTS. Since the bolts hold everything else together, this is almost impossible to do. If attachment is turned on in addition to directionality, the resulting plan still has only one direction in each subassembly, but it has many fewer subassemblies and all of them are more stable, with the bolts being inserted much earlier.

When only attachment is used, a plan with many subassemblies was produced, but each subassembly was attached. The LIPs were assumed to snap into the SOCKETS, so those three subassemblies are attached. The next three higher-level subassemblies built each nut/bolt structure, starting with the NUT and ending with the BOLT. In the next level subassembly the PLUNGERs were placed (these were assumed to lock in place) and finally the SHAFT and C-RINGS were inserted.

Criteria Enabled			Time (secs)	Number of Subassemblies	Plan Description
		D	5.32	6	All subassemblies monodirectional.
	A		17.82	8	Each subassembly fastened.
M			36.45	3	Only subassemblies are {SOCKET,LIP}.
M	A		37.78	3	Same as above.
	A	D	14.92	5	All subassemblies monodirectional and all fairly well fastened.
M		D	36.50	4	All subassemblies fastened. One subassembly uses 2 directions. This plan shown in figure 3.
M	A	D	38.54	4	Same as above.

Table 2: A table showing the results when different combinations of the manipulability (M), attachment (A), and directionality (D) criteria are enabled for the hole-punch depicted in figure 2. Times to generate each plan are given, together with the number of subassemblies in each plan (not including the top-level assembly), and a brief description of the plan.

If all four criteria are turned on, a subassembly is built monodirectionally containing the BASE, and all the SOCKET assemblies and their BOLTs and NUTs. The top-level assembly starts with the SHAFT, then inserts the subassembly on the shaft, and then inserts the C-RINGs. This is the plan depicted in figure 3. This is very stable, and only the top-level plan uses two directions, but manipulability is somewhat compromised.

Clearly for really good plans, additional criteria would be required, but it is evident that the planner is effective in choosing plans in response to the selected criteria, even with a small planning window size. The performance throughout all tests was reasonably good, taking less than 40 cpu seconds in all cases.

7 Conclusions

This paper has outlined an approach to finding good plans for assemblies using a hierarchical representation of the structural relationships among the parts of the assembly. Suggested plans may be listed for common structures, thus enabling the planner to incorporate case-specific knowledge. A bottom-up merging strategy for finding a plan for the overall structure is described. Quality metrics are used to guide the planner toward better plans, but full optimization is not attempted.

It has been demonstrated that the use of a full structure hierarchy can aid planning in several ways. It is important to recognize, however, that the user's choice of an structure hierarchy effects the performance of the planner. The application of suggested plans for higher-level structures and the reuse of plans for similar structures is only possible if those structures have been properly identified. Doing so is clearly advantageous since it not only leads to faster plans, but it also generates plans faster.

Other aspects of the choice of input structures are less clear-cut. This paper demonstrated that if the sizes of structures are kept constant, the planner will run in polynomial expected time. It would certainly be possible to contrive structures that achieve this, but this will also lead to lower quality plans. In this paper we have assumed that the assembly will be broken down into structures that seem natural, or have functional significance rather than structures that are specifically contrived to serve the planner. Much more study on these structures is needed.

The structure library is another area where further research is needed. We would like to be able to represent a richer range of advice than simple lists of suggested plans. This should include more general advice and situation-dependent advice. Since such advice would not give the complete plan for the structure, the system would still have to generate plans for child structures, but that planning would be guided by the advice propagated top-down. Also it would be useful to be able to adapt plans from one structure to a slightly different structure, allowing the system to reuse plans more widely.

Several aspects of the planning strategy deserve further study. The current planner merges only the top-level plans, and then constructs plans for subassemblies separately. Instead one could consider generalizing the merge operations so that full POIS plan hierarchies could be merged. A variety of other improvements would be worthy of investigation, including adding search heuristics so the A* search algorithm can be used instead of a simple best-first search.

Though there remains much to be done before a truly practical planner can be built, HAP demonstrates effectively that it is possible to construct a planner that integrates knowledge about how to build specific common structures into the framework of a general purpose, correct and complete search-based planner, and that doing so can significantly improve the quality of a planner.

References

[1] Daniel F. Baldwin, Thomas E. Abell, Man-Cheung Max Lui, Thomas L. De Fazio, and Daniel E. Whitney. An integrated computer aid for generating and evaluating assembly sequences for mechanical products. *IEEE J. of Robotics and Automation*, 7(1):78–94, February 1991.

[2] A. Bourjault. *Contribution a une approche méthodologique de l'assemblage automatisé: Elaboration automatique des séquences opératiores.* PhD thesis, L'Université de Franche-Comté, November 1984.

[3] Sugato Chakrabarty. *A Hierarchical Assembly Planning System.* PhD thesis, Texas A&M University, Department of Computer Science, December 1994.

[4] Sugato Chakrabarty and Jan Wolter. A hierarchical approach to assembly planning. In *IEEE Intl. Conf. on Robotics and Automation*, pages 258–263, May 1994.

[5] Thomas L. De Fazio and Daniel E. Whitney. Simplified generation of all mechanical assembly sequences. *IEEE J. of Robotics and Automation*, RA-3(6):640–658, December 1987.

[6] Thomas L. De Fazio and Daniel E. Whitney. Correction to "simplified generation of all mechanical assembly sequences". *IEEE J. of Robotics and Automation*, RA-4(6):705–708, December 1988.

[7] Richard Hoffman. Automated assembly planning for b-rep products. In *IEEE Intl. Conf. on Systems Engineering*, pages 391–394, August 1990.

[8] Richard L. Hoffman. Automated assembly in a CSG domain. In *IEEE Intl. Conf. on Robotics and Automation*, pages 210–215, May 1989.

[9] Luiz Homem de Mello and Arthur Sanderson. A correct and complete algorithm for the generations of mechanical assembly sequences. *IEEE Transactions on Robotics and Automation*, 7(2):228–240, April 1991.

[10] Luiz S. Homem de Mello and Arthur C. Sanderson. A correct and complete algorithm for the generation of mechanical assembly sequences. In *IEEE Intl. Conf. on Robotics and Automation*, pages 56–61, May 1989.

[11] Kai-I Huang. *Assembly Sequence Planning for Constrained Environments*. PhD thesis, Texas A&M University, Department of Industrial Engineering, May 1991.

[12] Y. F. Huang and C. S. George Lee. A framework of knowledge-based assembly planning. In *IEEE Intl. Conf. on Robotics and Automation*, pages 599–604, April 1991.

[13] Ehud Kroll. Towards using cost estimation to guide concurrent design processes. In D. Dutta, editor, *Concurrent Engineering*, volume PED-59, pages 281–293. ASME, New York, 1992.

[14] Sukhan Lee. Backward assembly planning with assembly cost analysis. In *IEEE International Conference on Robotics and Automation*, pages 2382–2391, May 1992.

[15] C.L. Liu. *Elements of Discrete Mathmatics*. McGraw Hill, New York, 1985.

[16] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, 1980.

[17] Judea Pearl. *Heuristics : Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley, Reading, Massachusetts, 1984.

[18] Pearl Pu. An assembly sequence generation algorithm using case-based search techniques. In *IEEE International Conference on Robotics and Automation*, pages 2425–2430, May 1992.

[19] Arthur C. Sanderson, Luiz S. Homem de Mello, and Hui Zhang. Assembly sequence planning. *AI Magazine*, 11(1):62–81, Spring 1990.

[20] Randall H. Wilson. *On Geometric Assembly Planning*. PhD thesis, Stanford University, Department of Computer Science, March 1992.

[21] Randall H. Wilson and Jean-Francois Rit. Maintaining geometric dependencies in an assembly planner. In *IEEE Intl. Conf. on Robotics and Automation*, May 1990.

[22] Jan Wolter. A combinatorial analysis of enumerative data structures for assembly planning. In *IEEE International Conference on Robotics and Automation*, pages 611–618, April 1991.

[23] Jan Wolter. On the automatic generation of assembly plans. In Luiz S. Homem de Mello and Sukhan Lee, editors, *Computer-Aided Assembly Planning*, pages 263–288. Kluwer Academic Publishers, Boston, 1991.

[24] Jan Wolter. A combinatorial analysis of enumerative data structures for assembly planning. *Journal of Design and Manufacturing*, 2(2):93–104, June 1992.

[25] Jan Wolter, Sugato Chakrabarty, and Jungfu Tsao. Mating constraint languages for assembly sequence planning. Technical Report 91-038, Texas A&M University, Computer Science Dept., September 1991.

[26] Jan Wolter and Periannan Chandrasekaran. A concept for a constraint-based representation of functional and geometric design knowledge. In *ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 409–418, June 1991.

[27] Jan Wolter and Periannan Chandrasekaran. Knowledge representation for design of mechanical assemblies. In *7th Annual NSF Conference on Design and Manufacturing Systems Research*, pages 633–640, January 1991.

[28] Jan Dithmar Wolter. *On the Automatic Generation of Plans for Mechanical Assembly.* PhD thesis, University of Michigan, Department of Computer, Information and Control Engineering, September 1988.