Submitted to 1997 *IEEE International Conference on Robotics and Automation*

Constraint-Based Interactive Assembly Planning*

Rondall E. Jones Randall H. Wilson Terri L. Calton

Intelligent Systems and Robotics Center
Sandia National Laboratories
Albuquerque, NM 87185-1008

Abstract

The constraints on assembly plans vary depending on the product, assembly facility, assembly volume, and many other factors. This paper describes the principles and implementation of a framework that supports a wide variety of user-specified constraints for interactive assembly planning. Constraints from many sources can be expressed on a sequencing level, specifying orders and conditions on part mating operations in a number of ways. All constraints are implemented as *filters* that either accept or reject assembly operations proposed by the planner. For efficiency, some constraints are supplemented with special-purpose modifications to the planner's algorithms. Replanning is fast enough to enable a natural plan–view–constrain–replan cycle that aids in constraint discovery and documentation. We describe an implementation of the framework in a computer-aided assembly planning system and experiments applying the system to several complex assemblies.

1 Introduction

Constraints on assembly plans come from a wide variety of sources. Design requirements, part and tool accessibility, assembly line and workcell layout, requirements of special operations, and even supplier relationships can drive the choice of a feasible or preferred assembly sequence. Computer-aided assembly planning systems promise to help product and assembly system designers to manage this complexity and choose good assembly sequences.

However, there are so many types of assembly constraints that it is impractical for a single program to encode them all. In many cases, the constraints are company- or product-specific. Hence a practical assembly planning system must have the ability to manage assembly constraints in a general way to determine how they impact the choice of assembly sequence.

In this paper we present a framework for representing and reasoning about assembly constraints of many types. The user chooses constraints from a library of standard constraint types, with a simple graphic interface for defining the specifics of the constraint. Constraints are implemented as *filters* that either accept or reject assembly operations proposed by the planner. Any constraint that can be encoded as a filter can be added to the constraint library in a straightforward way. Some constraints are supplemented with special-purpose modifications to the planner's algorithms for greater efficiency.

This constraint framework has been implemented and tested extensively in the Archimedes assembly planning system [6]. In a typical interaction, the system generates and animates a (flawed) plan, the user adds constraints that are not satisfied by the plan, asks for a new plan, and repeats until a satisfactory plan is found. Our users have found this interactive constraint discovery and planning process to be very natural and productive.

In the next section we place our work in the context of previous research on constraints and assembly planning. We then give an overview of our framework in Section 3. Section 4 delves deeper into some of the issues that arise in implementing the constraint system effectively. Section 5 describes our experiences using and testing the resulting planner. Finally, Section 6 concludes and give directions for future work.

2 Previous Work

Our assembly constraint filters are an instance of *generate-and-test*, a standard paradigm in artificial intelligence [7] as well as in assembly planning [3, 4]. Attaching special-purpose methods to some filters is also a well known efficiency technique [7], although the

*This work was supported by Sandia National Laboratories under DOE contract DE-AC04-94AL85000.

methods we use to accomplish this are novel (see Section 4).

Assembly planning systems that allow user-defined constraints have generally been of two types. In the first type (for example [2, 3]), users must specify all constraints before the long process of plan generation begins. In our experience, this is rarely possible: the user finds it very difficult to list *all* constraints on assembly ordering until some possible plans are considered. In the second type of system (e.g. [1]), a large space of plans is first generated, and then undesirable operations and states are pruned interactively by the user. However, the space of plans quickly becomes too large to edit or even generate as assemblies become moderately complex. Our interactive approach to this problem is foreshadowed in a much more limited form by [8].

Previous efforts to incorporate a comprehensive set of user constraints in assembly planners were based on liaison precedence relations (see e.g. [2, 12]). Precedence relations specify logical combinations of part connections that must be established either before or after others. We considered translating all of our constraints into precedence relations, but chose filters instead for reasons of efficiency and simplicity of implementation.

3 Approach

The constraint framework described here was required for the Archimedes assembly planning system [6]. That system takes CAD data for a product as input and automatically determines geometrically feasible sequences of motions to assemble the product from its parts. However, geometry is only a small part of assembly sequencing. In [5] we surveyed constraints that have appeared in the assembly planning literature, in hopes of adding many of those constraints to the Archimedes system.

Our constraint framework has the following qualities:

User Friendly Each constraint can be described simply in terms familiar to the user, has straightforward effects, and combines with others in a very predictable way.

Maintainable Each constraint simply provides a filtering function that disallows some assembly operations. The filters are completely independent, allowing new constraint types to be added easily.

Efficient The filters are coded as procedures, and special-purpose methods can be attached to improve efficiency.

The result is an extensible library of simple but useful constraints that enable a new, highly interactive mode of assembly planning.

3.1 Constraints

We provide a library of constraint types, from which the user can instantiate constraints on the assembly plan. For instance, one type of constraint is called REQ-SUBASSY; it requires that a particular set of parts appear as a subassembly (with no other parts present) at some point in the plan. To instantiate a REQ-SUBASSY constraint, the user selects the parts that must belong to the subassembly. Multiple REQ-SUBASSY constraints can be instantiated, each with a different set of parts.

Constraints are implemented as filters. During planning, each proposed assembly operation is passed to the constraint's *filter function*, which returns `true` or `false` depending on whether the operation satisfies the constraint or not. Only an operation that satisfies all current constraints is feasible. For instance, consider an operation placing subassembly S_1 into subassembly S_2.[1] The filter function of a REQ-SUBASSY constraint with part set P will return `true` if and only if

$$P \subseteq S_1 \vee P \subseteq S_2 \vee (S_1 \cup S_2) \subseteq P \vee [(S_1 \cup S_2) \cap P] = \emptyset$$

In other words, the operation satisfies the constraint if it keeps the parts in P together, if only parts in P are involved, or if no parts in P are involved.

As a standard interface to all constraints, the filter function provides a number of benefits. First and foremost, it makes the implementation of each constraint type independent. Interactions between constraints need not be considered, and each constraint can be implemented in its most straightforward and efficient way. This becomes crucial as the number of constraint types grows. In addition, constraints can vary in the data associated with them, their instantiation routines, various debugging outputs, and so on.

Filter functions are flexible enough that we have been able to implement a large subset of the constraints identified in [5], plus additional ones that our users requested. The flexibility is further demonstrated by the REQ-TOOL constraint, which encodes tool accessibility constraints for various hand and robotic tools [9]

[1] An operation will typically have other specifications, such as a mating trajectory and perhaps an assembly orientation, but these are not relevant to REQ-SUBASSY.

DISCLAIMER

Constraint Name	Scope
REQ-CONNECT*	strategic
REQ-LINEAR*	strategic
REQ-VERTICAL*	strategic
PRH-STATE	tactical
PRH-SUBASSY	tactical
REQ-CLUSTER	tactical
REQ-FASTENER*	tactical
REQ-LINEAR-CLUSTER	tactical
REQ-LINEAR-PARTS	tactical
REQ-ORDER-FIRST	tactical
REQ-ORDER-LAST*	tactical
REQ-ORDER-LIAISON	tactical
REQ-ORDER-PART	tactical
REQ-PART-SPECIAL	tactical
REQ-PATHS-AXIAL	tactical
REQ-STACK	tactical
REQ-SUBASSY*	tactical
REQ-SUBASSY-WHOLE*	tactical
REQ-SUCCESS-PART	tactical
REQ-TOOL*	tactical

Table 1: Constraints implemented in Archimedes. Those marked by * have special-purpose routines for efficiency.

within the framework. Table 1 lists the constraint types currently in the Archimedes system.

While filter functions themselves are usually quite fast, the generate-and-test abstraction can sometimes lead to an inefficient planning process overall. This is particularly true when many dead-ends appear in the search space, or when a large number of assembly operations are generated but few satisfy the constraints. In many cases, special purpose routines can increase efficiency dramatically. The constraint types for which we have implemented such methods are indicated with an asterisk (*) in Table 1. Subsection 4.4 provides more details of these methods.

3.2 Interaction

In experiments with product designers and assembly process engineers, we have found that a high level of interactivity is critical to successful application of an assembly planner. Usually the designer cannot list all the constraints on assembly order at the start of the planning session. However, many of these constraints become "obvious" when the system graphically illustrates a plan that violates them. Seeing a violation, adding a constraint to remove it, and then replanning becomes the main cycle of interaction in the system. In this way, the assembly planner aids constraint discovery and management as well as plan generation and optimization.

Note, however, that placing a new constraint is very different from ruling out a certain operation, as performed in some previous systems such as [1]. While a single operation demonstrates the need for a constraint, placing the constraint usually limits the allowable plans far more than prohibiting a single operation. In the best case (and in many practical cases), the constraint encodes the manufacturing constraint exactly.

This plan–view–constrain–replan cycle requires that replanning be performed at interactive speeds. In the Archimedes system, a first assembly plan for a product can usually be found in a few minutes [6]. However, the most expensive part of planning is ensuring that part insertions are collision-free. By saving collision-detection information, replanning usually requires 10 or 20 seconds for assemblies of up to 100 parts.

There is of course no guarantee that all of the constraints the user has instantiated can be satisfied by a single plan. In this case, the planner fails and enters a "debug" mode that helps the user to determine the cause of the failure. If the constraints are all real, then a problem with the product design may be indicated. In most cases, some constraints can be adjusted to allow planning to succeed. When there are inaccuracies or inconsistencies in the product CAD data, planning can fail before the user has entered any constraints. The debug mode also supports finding such problems, and certain problems can be fixed within Archimedes. Subsection 4.3 provides more details.

After all known manufacturing constraints have been entered, the user can then ask for an optimal plan, according to user-specified costs of certain standard operations. In some cases additional unstated constraints will be violated and discovered as the planner looks through a large space of plans to find the best. In this case the new constraints must be added and the cycle repeats.

4 A Detailed View

4.1 Assembly Planning Approach

The approach taken to assembly planning is obviously critical to the design, implementation, and performance of a user constraint system. It especially affects special-purpose routines for efficiency.

The Archimedes system [6] generates two-handed

monotone assembly sequences in reverse, starting from the more highly constrained, fully assembled state. This a standard technique in assembly planning. The search space is an AND/OR graph of subassembly states and operations to construct them from smaller subassemblies. The planner uses an NDBG of each subassembly [11] to efficiently determine assembly operations that might be performed to construct that subassembly, then checks these operations for geometric collisions, which is essentially a built-in filter. Operations are then checked against the list of user constraints.

The search strategy is carefully tuned to generate a first plan as quickly as possible in the domain of mechanical assembly. This is critical to achieve the desired view–constrain–replan cycle of interaction. An AND/OR version of iterative sampling is performed: during each pass of the algorithm, a single assembly sequence is generated, making random choices of operations to construct each subassembly. The first time any subassembly is visited, only a single operation is generated to construct it, and the subassemblies of that operation are then visited. The strategy spends minimal time reaching a first solution, like a depth-first search, but avoids getting caught by bad early decisions as a depth-first search would. The same algorithm functions as an any-time algorithm to optimize the assembly sequence when the user requests. See [10] for more details.

4.2 User Interface

The user interface is critical to effectiveness and user acceptance of an interactive planning system. The constraints must be easy to understand, define, and manage. In this subsection we describe features of the Archimedes user interface that are important to the success of the constraint system.

Figure 1 shows the main Archimedes user interface. The upper left window shows the program's current status, displays any diagnostic output, and allows pausing or aborting any computation. The upper right window provides graphic output and part/subassembly selection. The bottom window is the main control window with only six functions, which the user invokes in a generally left-to-right order during the planning process.

After loading the CAD data for an assembly and perhaps making some initial adjustments to it (see Subsection 4.3), the user selects "Plan", which brings up the planning dialog shown in Figure 2. The top half of this dialog concerns global choices for the planner, and the bottom half provides management of the current set of constraints.

Constraints are added by clicking on the "Add" button at the bottom, which brings up a sequence of menus and questions that let the user pick a constraint type and specify the particulars of the desired constraint. Each constraint requires the user to select one or more parts of a "controlled" set in the graphic window, such as the parts making up a subassembly. For some constraints additional inputs must be provided, such as a second set of parts required by the constraint or the choice and placement of a tool to be used in assembly. An auxiliary window provides a list of named subassemblies to facilitate selection of larger sets of parts. Each constraint can be given a name and descriptive comment by the user.

Once defined, constraints are listed in the planning dialog. They can be edited using a process very similar to initial definition. They can also be deleted, temporarily suspended, and re-activated. Constraint suspension is a very useful feature that allows the user to consider various scenarios for assembly. Constraints often embody assumptions about the product assembly scheme; by suspending some and replanning, the user can compare the cost of removing the assumption to the possible gains in assembly sequence efficiency that result.

4.3 When Planning Fails

When the product cannot be assembled according to the current set of constraints, the planner fails and enters a "debug" mode that helps the user to determine the cause of the failure. For example, one can request that the planner try to remove a particular part or subassembly (from the subset of parts remaining when the planner failed) in a direction that appears possible. Collisions or constraints that disallow the operation are then posted in the status window. Other options include trying to disassemble every pair of parts in the offending subassembly, or trying to remove any parts along a given trajectory.

Often, the planner will fail without any user-defined constraints. This is sometimes due to limitations in the planner's algorithms, such as an inability to reason about flexible parts such as snapfits and springs. Other times, inaccuracies or inconsistencies in the product CAD data cause the planner to fail. Examples include pressfit parts and threaded parts that are modeled as cylinders too large for their holes. Archimedes includes a set of model adjustment features, or *overrides*, which can be used to correct such problems. These include a function to effectively add threads to cylindrical contacts between parts; to specify that certain part inser-

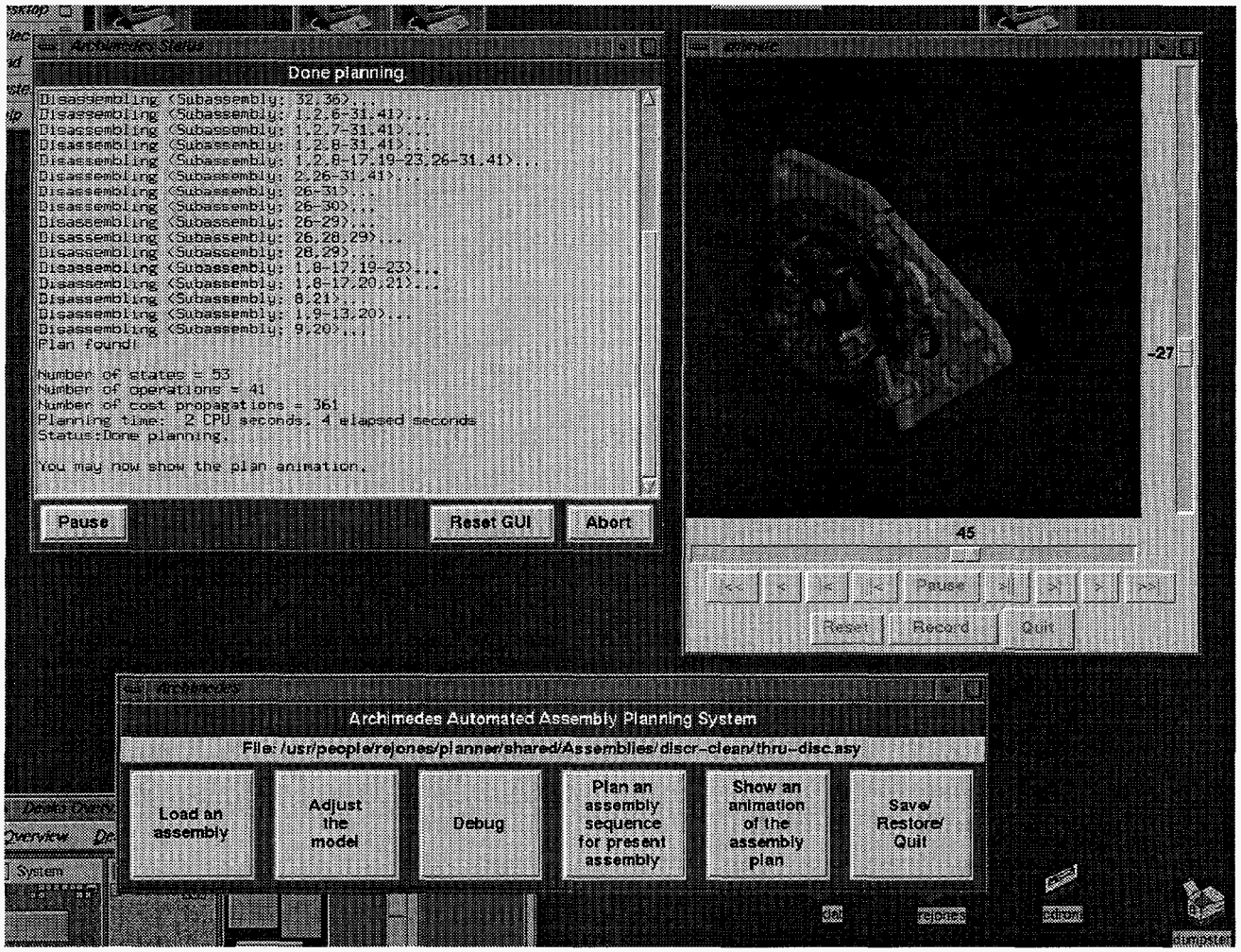

Figure 1: Archimedes user interface

tions are in fact possible, even though collisions occur between the parts; and to delete a part temporarily.

4.4 Efficiency

As mentioned above, the generate-and-test abstraction can sometimes lead to an inefficient planning process, in which case supplemental routines can improve planning efficiency greatly. These routines are very dependent on the internal algorithms of the planner. Because Archimedes plans backward from the assembled state to individual parts, the supplemental routines must be implemented as if they were constraints on *disassembly*, not assembly.

For instance, if the user has created a REQ-SUBASSY constraint with part set P, and parts not in P are present, then P cannot be "split" at that point in the plan. To implement this constraint efficiently, a supplemental routine binds the parts of P together for that stage, not considering any operations that split them. This is accomplished by placing bidirectional arcs between every pair of parts in P in every blocking graph of the subassembly [11].

Supplemental routines must be considered carefully, trading off the added speed against the increase in planner complexity. Three characteristics identify candidates for pre-processing:

1. the constraint either leads to many dead-ends in the search space or rules out a very large proportion of generated operations,

2. an efficient method exists to implement the pre-processing, and

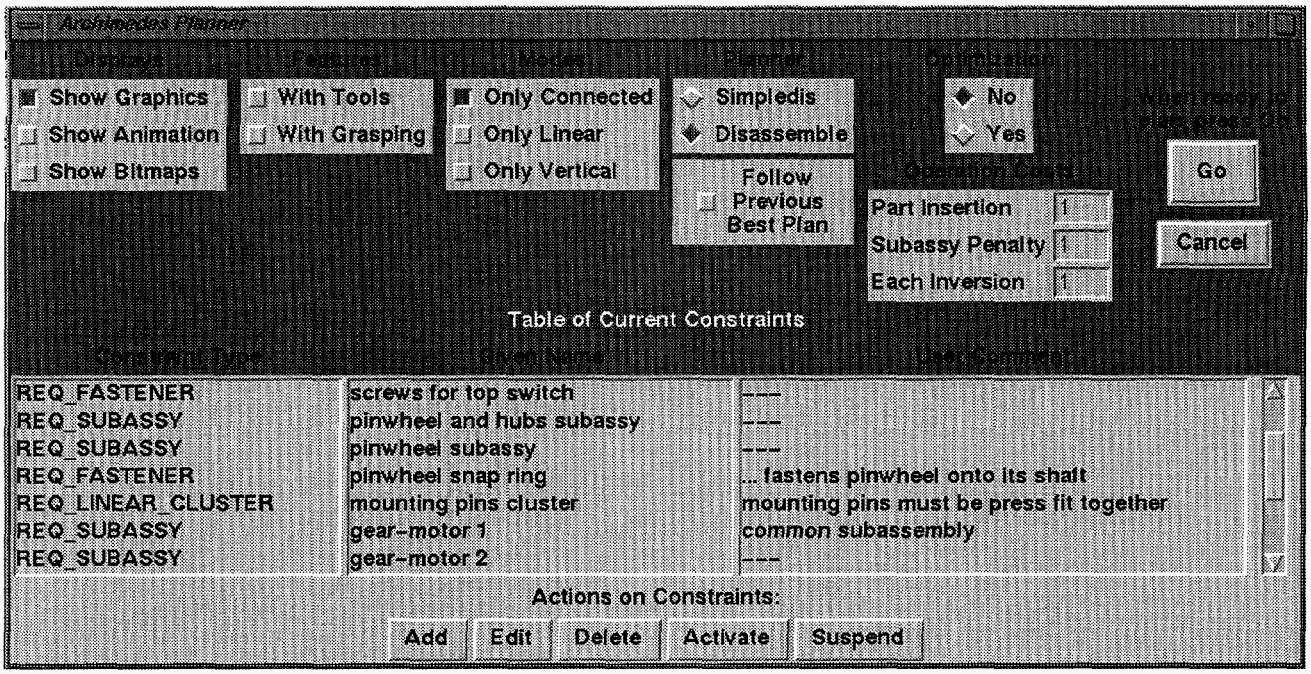

Figure 2: Planning dialog

3. the constraint is used often.

The REQ-FASTENER constraint type is another instructive example. Fasteners are very common in mechanical assemblies. The REQ-FASTENER constraint requires that as soon as one set of parts is joined (the *fastened parts*), then a set of *fastener parts* must immediately be placed. In reverse, this constraint means that as soon as a single fastener part is removed, then all other fasteners must be immediately removed, followed by at least one of the fastened parts. If any of these operations is infeasible, a dead end appears in the search space (in fact many can appear).[2]

The filter function for REQ-FASTENER is straightforward, but its supplemental routines are the most complex we have implemented. The fastener parts are removed from the assembly representation and considered secondary parts. Fasteners are placed back in subassemblies implicitly for collision checking and other calculations. Before generating operations to construct a certain subassembly, the planner determines which fasteners could be placed into it; for those that cannot, the corresponding fastened parts are bound together as for REQ-SUBASSY.

[2]A similar dead end would appear in a forward-planning system.

Note that when a constraint has supplemental routines, the planner still calls the constraint's filter function, which should never return `false`. This double check is very useful to ensure correctness, because the supplemental routines are complex and interact with each other. A supplemental routine is conceivable that would only *reduce* the number of operations rejected by the filter function, but we have not found such a case.

In almost all cases, adding constraints reduces planning time. Though the computation of the filter functions obviously takes time, the time saved by not searching states below an invalid branch outweighs the cost. In fact, constraints can be used to guide the planner to a correct plan for assemblies that would otherwise be intractably large.

4.5 Implementation

Archimedes is implemented in C++, with Tcl/Tk used for the graphic interface and OpenGL® for 3D graphics and animation. The constraints are implemented as a hierarchy of C++ derived classes. Each type of constraint simply overrides the filter function from a base class, along with methods to define the type, name, etc. of the constraint. Each constraint type also has its own data members, such as part sets,

Assembly	Parts	Overrides	Constraints
Sprytron	18	0	5
Door latch	23	28	8
Discriminator	42	18	15
Hughes	472	26	144

Table 2: Example assemblies planned with Archimedes

tool choices and points of application, and so on. Some of the supplemental routines are implemented as constraint class methods; however, most cannot be separated from the planner's algorithms, and are woven directly into the planner implementation.

In [5] we identified two main types of constraints on assembly plans. *Strategic* constraints apply to the entire assembly and its plan, while *tactical* constraints only apply to certain subsets of the parts. Archimedes currently implements strategic constraints as flags for the planner. However, in theory there is no real difference, since tactical constraints can usually be applied to the entire assembly, and strategic constraints can always be limited to a subset of the parts. For instance, we found that the REQ-PATHS-AXIAL constraint, identified in [5] as strategic, was also very useful applied to a subset of the assembly. Implementing it as a tactical constraint allowed both uses, was simpler to implement, and caused no loss in efficiency. We plan to replace all strategic constraints in Archimedes with tactical constraints.

5 Experiments

We have applied the Archimedes planner, extended with the constraint system, to a number of actual assemblies from sources in government and industry. Examples are listed in Table 2.

The **sprytron** is an eighteen part diode-like device, including a surrounding assembly fixture. Most of the parts are symmetric or nearly symmetric about a central axis. The CAD data required no overrides to produce a first plan. However, in the resulting plan some parts that would more naturally be inserted along the axis were inserted from other directions. Adding a REQ-PATHS-AXIAL constraint removed all these unwanted directions, and adding a REQ-ORDER-FIRST placing the fixture first made the plan even better. Three REQ-ORDER-LIAISON constraints were then added to fine tune the assembly plan to specific manufacturing constraints.

The **latch** mechanism involves 23 parts, some of which are very complex. The presence of several snap-fit or riveted fasteners, plus many inaccuracies in the CAD data, required 28 overrides. A good plan was obtained after eight REQ-SUBASSY constraints were added, and one REQ-ORDER-LAST was used.

The **discriminator** is a 42-part clockwork-like mechanism used as a safety device. It is the object partially shown in the animation window in Figure 1. Several parts overlapped in the CAD data, including 12 screws which were modeled larger than their corresponding holes, resulting in 18 model overrides. A plan for the resulting adjusted model was then found with no need for constraints. To make Archimedes place all fasteners in appropriate groups seven REQ-FASTENER constraints were needed, and to match the subassemblies intended by the designer six REQ-SUBASSY constraints were used. In addition, one REQ-LINEAR-CLUSTER was used, and the chassis was requested to be done first with a REQ-ORDER-FIRST.

The **Hughes assembly**,[3] with 472 parts, is to our knowledge the largest assembly that has been processed by any automated assembly planning system. Since Archimedes plans only for straight line assembly motions, and this assembly contained a number of flexible parts (such as cables) that could not use straight line insertions, we overrode 22 part mating situations with specific matings. Four other overrides clarified contacts between parts. A large number of REQ-SUBASSY-WHOLE constraints were used for subassemblies that our Hughes customer was not interested in sequencing. (REQ-SUBASSY-WHOLE is equivalent to REQ-SUBASSY but tells the planner in addition not to bother disassembling the subassembly.) A breakdown of all the constraints used to produce a plan useful to the customer is in Table 3.

The reader may note an approximate 3-to-1 ratio of parts to constraints in Table 2. While our data is clearly insufficient to draw any conclusions, this may be indicative of what we should expect for typical applications.

6 Conclusion

Overall, we have been very pleased with the concept and implementation of constraints as filters. It keeps the code simple, maintainable, and efficient, especially when supplemented with special-purpose routines for

[3]Unfortunately, confidentiality agreements prevent us from revealing more information about the Hughes assembly. We expect to have permission to describe it in more detail, including a figure, in time for the final version of this paper.

Constraint	Count
REQ-SUBASSY-WHOLE	70
REQ-ORDER-LIAISON	48
REQ-CLUSTER	11
REQ-SUBASSY	7
REQ-FASTENER	3
REQ-ORDER-FIRST	2
REQ-ORDER-LAST	2
REQ-PATHS-AXIAL	1

Table 3: Constraints for the Hughes Assembly

certain constraints. A rich variety of useful constraints can be so represented, and we have easily added constraint types when required. The interactive mode of user-guided planning that results from the system has been very effective in our experience.

This paper has only considered the strategic and tactical requirements from [5]. At present, we are working to expand the scope of our algorithms to also include optimization criteria and suggestions. These allow the user to "guide" the planner toward a best plan, or more quickly toward a good plan, respectively.

In addition, we are working on additional methods to speed replanning. One approach is to save the current best plan, and attempt to follow it when planning for a new set of constraints. Since the previous plan is often not feasible, methods must be developed to follow it "as much as possible." In addition to the potential speedup, the user might perceive smaller adjustments in the plan to accommodate constraint changes (rather than the often-radical changes that happen now).

References

[1] D. F. Baldwin, T. E. Abell, M.-C. M. Lui, T. L. De Fazio, and D. E. Whitney. An integrated computer aid for generating and evaluating assembly sequences for mechanical products. *IEEE Trans. on Robotics and Automation*, 7(1):78–94, 1991.

[2] T. L. De Fazio and D. E. Whitney. Simplified generation of all mechanical assembly sequences. *IEEE Journal of Robotics and Automation*, RA-3(6):640–658, 1987. Errata in RA-4(6):705-708.

[3] J. M. Henrioud and A. Bourjault. LEGA: a computer-aided generator of assembly plans. In L. S. Homem de Mello and S. Lee, editors, *Computer-Aided Mechanical Assembly Planning*, pages 191–215. Kluwer, 1991.

[4] L. S. Homem de Mello and A. C. Sanderson. A correct and complete algorithm for the generation of mechanical assembly sequences. *IEEE Trans. on Robotics and Automation*, 7(2):228–240, 1991.

[5] R. E. Jones and R. H. Wilson. A survey of constraints in assembly planning. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 1525–32, 1996.

[6] S. G. Kaufman, R. H. Wilson, R. E. Jones, T. L. Calton, and A. L. Ames. The Archimedes 2 mechanical assembly planning system. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 3361–8, 1996.

[7] N. J. Nilsson. *Principles of Artificial Intelligence.* Springer-Verlag, 1980.

[8] R. H. Wilson. Minimizing user queries in interactive assembly planning. *IEEE Trans. on Robotics and Automation*, 11(2):308–312, 1995.

[9] R. H. Wilson. A framework for geometric reasoning about tools in assembly. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 1837–44, 1996.

[10] R. H. Wilson and G. Laguna. Stochastic search for AND/OR graphs. In preparation, 1996.

[11] R. H. Wilson and J.-C. Latombe. Geometric reasoning about mechanical assembly. *Artificial Intelligence*, 71(2):371–396, 1994.

[12] J. D. Wolter, S. Chakrabarty, and J. Tsao. Mating constraint languages for assembly sequence planning. *IEEE Trans. on Robotics and Automation.* To appear.