ON GEOMETRIC ASSEMBLY PLANNING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Randall H. Wilson
March 1992

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jean-Claude Latombe (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mark R. Cutkosky (Mechanical Engineering)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Leonidas J. Guibas

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jay M. Tenenbaum

Approved for the University Committee on Graduate Studies:

Abstract

This dissertation addresses the problem of generating feasible assembly sequences for a mechanical product from a geometric model of the product. An operation specifies a motion to bring two subassemblies together to make a larger subassembly. An assembly sequence is a sequence of operations that construct the product from the individual parts.

I introduce the non-directional blocking graph, a succinct characterization of the blocking relationships between parts in an assembly. I describe efficient algorithms to identify removable subassemblies by constructing and analyzing the NDBG.

For an assembly A of n parts and m part–part contacts equivalent to k contact points, a subassembly that can translate a small distance from the rest of A can be identified in $O(mk^2)$ time. When rotations are allowed as well, the time bound is $O(mk^5)$. Both algorithms are extended to find connected subassemblies in the same time bounds. All free subassemblies can be identified in output-dependent polynomial time.

Another algorithm based on the NDBG identifies subassemblies that can be completely removed by a single translation. For a polyhedral assembly with v vertices, the algorithm finds a removable subassembly and direction in $O(n^2v^4)$ time. When applied to find the set of translations separating two parts, the algorithm is optimal.

A final method accelerates the generation of linear assembly sequences, in which each operation mates a single part with a subassembly. The results of geometric calculations are stored in logical expressions and later retrieved to answer similar geometric queries. Several types of expressions with increasing descriptive power are given.

An assembly sequencing testbed called GRASP was implemented using the above methods. From a standard three-dimensional model of a product, GRASP finds part contacts and motion constraints, and constructs an AND/OR graph representing a set of geometrically feasible assembly sequences for the product. Experimental results are shown for several complex products.

Acknowledgements

This research could never have been accomplished without the strong guidance and encouragement of my advisor, Professor Jean-Claude Latombe. He helped me focus on the key issues and pushed me to learn more. I hope I can live up to his example.

I thank the other members of my reading committee, Professor Mark Cutkosky, Professor Leo Guibas, and Dr. Marty Tenenbaum, for serving on the committee, for their encouragement, and for many helpful discussions. I look forward to working more with them in the future.

The vibrant atmosphere at the Robotics Laboratory was ideal for doing research in geometry and motion. I am especially grateful to Jean-François Rit, Achim Schweikard, and Toshi Matsui, with whom I worked closely.

Special thanks must go to my parents, Howard and Virginia Wilson, who taught me the value of education and lit the flame of curiosity, and to Cheryl and Brian, who didn't snuff it.

Last but not least, I thank Ramey for being supportive, tolerant, and impelling, and for keeping my life going while I wrote.

Financial support for this work was provided by a fellowship from the National Science Foundation, DARPA contract N00014-88-K-0620 (Office of Naval Research), the Stanford Integrated Manufacturing Association (SIMA), and Digital Equipment Corporation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Product creation has traditionally been separated into at least two stages: first a designer specifies the geometry and physical qualities of the finished product to perform the function needed, then a manufacturing engineer takes over and tries to find a way to manufacture the product. If it is too difficult to manufacture, the product is sent back for redesign, and the cycle repeats. Because the design of the product and the design of the manufacturing process are only loosely coupled, many cycles may be required to find a satisfactory design, while the final artifact usually remains more expensive to build than necessary.

Concurrent design of a product and the process to make it is one way to address this problem. In concurrent design, the manufacturing process is created simultaneously with the product plans, so that constraints arising from manufacturing can be directly incorporated into the design, thereby reducing global iteration. In addition, automated tools support this process at a high level. Human engineers use computer workstations to build a shared product model, which is continually updated and critiqued to give the designers manufacturing and servicing feedback about the design. Some of the agents affecting the developing product model are computer programs, which do process planning, check for consistency, and perform other tasks previously left to human engineers. Examples of concurrent engineering systems under development are the NextCut [22, 23] and DesignWorld [29] systems.

1.1 Assembly Planning

A proficient assembly planner will be an integral part of any concurrent design system. A good assembly process can reduce assembly time, raise quality and reliability, allow greater flexibility in responding to equipment failures, and reduce capital costs for robots and fixtures. Furthermore, many of the manufacturing constraints that influence a design come from the need to assemble constituent parts. For example, the assembly scheme influences the tolerances imposed on individual parts, the shape of mating surfaces, and the design of fasteners.

Assembly planning for production is typically performed by an industrial engineer. Service and repair technicians devise reasonably efficient and error-free disassembly and reassembly plans on the fly. However, an automated assembly planner would have great advantages, especially when part of a concurrent design system:

- Assembly planning is complicated and time-consuming for a human. Beside avoiding the high cost of an engineer's time, an assembly planning program will accelerate the generation of an assembly plan. With the higher planning speed comes faster introduction of manufacturing constraints into the design process, as well as shorter time-to-market.

- Intuition, rules of thumb, and approximate reasoning help human engineers to quickly find good assembly plans, but they also may lead them to overlook the best assembly plan. A computer assembly planner could enumerate all feasible assembly plans to ensure that no better procedure exists. Furthermore, by exploring the set of all plans, or "plan space," a designer could come to a better understanding of the manufacturing aspects of the design.

- Current methods for evaluating the manufacturing aspects of a design, such as the Design for Assembly analysis of Boothroyd and Dewhurst [12], require extensive human judgement, and designers complain that they are tedious to use. A fast, automatic assembly planner would allow a designer to ask "what if" questions, quickly find the consequences of design decisions, and more easily evaluate the manufacturing impact of alternative designs.

- Small batch manufacturing requires that assembly machines, people, and processes switch from one product to another quickly and often. An automated planner could

help to merge the assembly schemes for several products and allow faster changeover. In addition, small batch size only amplifies the importance of low capital costs and fast creation of a manufacturing process.

- A hybrid system, allowing a human to work in tandem with an automated assembly planner, could relieve the engineer from much of the repetitive and tedious work of devising an assembly plan. Meanwhile, the computer would check the human reasoning for consistency. A more competent, efficient, and accurate assembly planning system would result.

An automated assembly planner would have great utility in both concurrent engineering and more mainstream manufacturing planning. In answer to these needs, a body of work has arisen in recent years attempting to systematize and automate the assembly planning process (see below). The concurrent design paradigm imposes the following demands on an assembly planner, although these qualities are to be desired in any such system:

Autonomy Using the assembly planner must require minimal effort from the designer. A human will quickly tire of doing detailed geometric reasoning, or even supplementing the assembly model to help the planner. As a result, the planner should build its model of the target assembly from readily available data, such as a CAD model of the product. Furthermore, the program must include automated geometric and physical reasoning capabilities to allow it to generate assembly plans from just the assembly model given by a designer. Any required human input should be sparse, be useful to the designer, and lack tedium.

Accuracy If an assembly planner returns bad assembly plans or fails to find plans when they exist, it will quickly lose what little trust the human will place in the computer. Thus the employed geometric reasoning methods cannot ignore important details, the search algorithms must be correct and complete, and approximations must be relevant to the assembly planning domain.

Speed If the engineer has to wait a long time to get assembly feedback from the system, it will be used infrequently, and the advantages of concurrent design will be lost. Thus the planner must finish its work in seconds or minutes instead of hours or days. To accomplish this it might need to work incrementally, modifying previous plans instead of generating new ones from scratch.

This dissertation describes progress toward the goal of fulfilling the above demands. Although this work is inspired by the concurrent design framework, the methods described here have broad application to assembly planning in general.

1.2 Assembly Sequencing

Given a complete description of a target assembly and the resources available for its manufacture, *assembly planning* refers to the process of creating a detailed manufacturing plan to create the whole from the separate parts. The assembly planning problem in general includes such problems as work floor scheduling, fixture design and manipulation, feeder and tool selection, and robot path planning. A full treatment of assembly planning is beyond the scope of this work.

This dissertation concerns the subproblem of assembly planning that is commonly called *assembly sequence planning*, or *assembly sequencing*. Assembly sequencing attempts to discern and represent the constraints on assembly plans that emerge strictly from the geometry and structure of the product itself, without considering the influence of the "environment" on the assembly process. The need to grasp parts, influences of fixtures, or movements of any objects other than the components of the assembly are not considered; only the parts and their relative positions are significant. Since the results of assembly sequencing are independent of the technology used to assemble the product, assembly sequencing is well suited to concurrent design, where manufacturing analysis is needed even though the final assembly technology may not be known.

The result of assembly sequencing is a set of assembly operations and constraints on their ordering. Each operation specifies a motion that brings two or more subassemblies together to make a larger subassembly. Any ordering of the operations that obeys the sequence constraints is called an assembly sequence. An assembly plan is created from an assembly sequence by adding details such as fixtures, orientations, and grasping locations, and taking into account the corresponding new constraints. These added constraints might make a particular assembly sequence impossible to execute, but any feasible assembly plan can be generated from its corresponding valid assembly sequence.

I concentrate specifically on *geometric* assembly sequencing, the automatic generation of assembly sequences satisfying geometric constraints. From just a geometric model of the goal assembly and the individual parts, a geometric sequencer computes a set of part

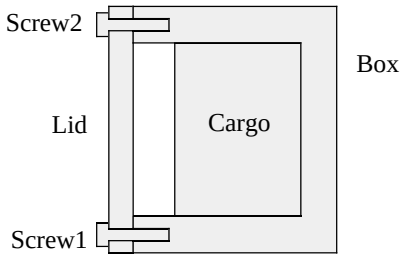

Figure 1.1: A simple assembly

motions to construct the product from the parts, such that no parts collide in the process. Mechanical and physical concerns, such as part tolerances, strains, and clamping forces, are not addressed. A geometric sequencer must identify subassemblies, analyze part contacts, find possible directions of motion, and reason about blocking relationships between parts and subassemblies.

For example, consider the simple crate assembly shown in figure 1.1. The input to the assembly sequencer consists of just the geometric models of the individual parts and their final positions as shown. An assembly sequence for the crate is any set of motions that bring the parts into their final relative positions from separate starting positions. Because the parts have little interaction in their unassembled configuration, each operation in the assembly sequence can be specified in terms of the parts involved and their relative motions. One possible assembly sequence for the crate is as follows:

1. Translate the `cargo` into the `box` from the left.
2. Translate the `lid` into position on the `box` from above.
3. Screw `screw1` into the `lid` and `box` from the left.
4. Screw `screw2` into the `lid` and `box` from the left.

1.3 Previous Work

The state of the art in assembly sequencing consists of a broad array of techniques, each addressing certain aspects of the problem. This work can be roughly divided into Artificial Intelligence approaches to planning, planners specifically designed for assembly sequencing, physical reasoning to validate single assembly operations, and computational geometry approaches. The summary below is an overview of selected work; additional related research

is described where relevant throughout this thesis.

1.3.1 Artificial Intelligence Planning

STRIPS, NOAH, and SIPE are good examples of traditional AI planning systems [28, 57, 62]. The blocks world, an early planning domain in Artificial Intelligence consisting of blocks that can be stacked and unstacked, is a primitive assembly planning domain. NOAH was originally aimed at supplying instructions to a human to repair an air compressor, including disassembly and assembly plans. In typical domains for these planners, actions are quite varied (for instance moving between rooms, grasping objects, recharging batteries), while geometric constraints are expressed in a few simple logical predicates, such as `ON(A, B)` or `IN(pen, Room1)`. In contrast, an assembly planner need only consider a few types of operations (joining two subassemblies, certain fixturing operations), while the constraints arising from geometric models are so complicated that expressing them in a logical notation would be exhausting and inefficient. Hence AI planning techniques seem best suited to handling the non-geometric parts of the assembly sequencing problem, while other methods are used to perform geometric reasoning.

1.3.2 Assembly Sequencing

In the last decade, a number of systems have been targeted specifically at assembly sequencing. These systems differ both in their representation for assembly sequences and in the reasoning techniques they use to identify assembly operations that satisfy geometric and mechanical constraints. Representations for assembly sequences will be discussed in section 2.5. Several important approaches to verifying assembly operations are described below.

Bourjault [14] describes an interactive system for generating the assembly sequences for a product. The method starts with a *liaison graph* of the assembly, which is a graph of connections between the parts. Liaisons usually, but not always, involve contact between the two parts. An assembly sequence corresponds to a particular order in which the liaisons can be established. Geometric reasoning is supplied by a human, who answers carefully constructed, yes-no questions about whether certain liaisons can be established before or after others. From the answers to these questions, the assembly sequences for the product can be inferred.

De Fazio and Whitney [25] drastically reduce the number of questions and answers required to represent the physical knowledge in Bourjault's method. The user answers each question with a logical formula characterizing the situations in which a liaison can be established. Unfortunately, these formulas become complicated and difficult to create accurately for large assemblies, even for a human.

Baldwin [5] later includes simple geometric checks to answer some questions automatically, further reducing the number of questions to the user in the previous two techniques. A human is still used as the final judge of assembly operations.

Homem de Mello and Sanderson [34, 36] use disassembly planning to automatically generate assembly sequences. Disassembly planning computes a way to disassemble the product, then reverses the sequence to produce an assembly sequence. To identify a single feasible operation, the method generates all possible subassemblies and tests the operation removing each subassembly from the rest of the assembly. An operation is tested by a predicate that includes geometric, mechanical, and stability checks. Local freedom is the main geometric test computed in [34]; a subassembly is locally free when it can move a small distance relative to the rest of the assembly considered as a solid. In this generate-and-test approach, a number of candidate operations exponential in the number of parts may be generated before a feasible operation is found, rendering the approach impractical for large assemblies.

Wolter [68, 69] computes linear assembly sequences for a product, in which each operation joins a single part to a subassembly. The sequences are optimized according to certain criteria. However, the input to the method includes possible motions for each part, and sets of other parts that interfere with each motion. No systematic way of generating the possible part motions is given, and the sequencer does no geometric computation.

A number of other approaches to automatic assembly sequencing have been proposed. Lee and Shin [44] describe a number of heuristics to group parts of an assembly into subassemblies, but parts can only move along the major axes, and some operations may not be found. Hoffman [33] generates disassembly sequences involving complicated motions. However, some assembly sequences may be missed, and as in [68] the directions of motion are given as input.

Efficient methods are needed that operate directly from the geometric models of the assembly, yet are guaranteed to find an assembly sequence if one exists.

1.3.3 Motion Planning

An assembly sequencing problem can be seen as a motion planning problem with multiple moving objects. Each part is an independent robot, and a collision-free path must be found for the individual parts to move from an unassembled state to the assembled state. However, the general motion planning problem is known to take time exponential in the number of degrees of freedom; for an assembly of n rigid parts this formulation yields $6n$ degrees of freedom, so using a general path planner in this naive way is clearly impractical.

Motion planning plays a more limited role in determining feasible insertion trajectories for single assembly operations. General motion planning algorithms are described by Latombe [42]. However, many general methods make assumptions that are not compatible with assembly planning, while answering more complex questions than are needed. For instance, some motion planners approximate the shapes of objects or assume the objects do not come in contact, both drawbacks for assembly planning (see for instance [7]). On the other hand, relatively simple geometric techniques can test the feasibility of the large majority of assembly operations accurately (see Chapter 3).

Some motion planning methods have been developed specifically to suit the requirements of assembly planning. Valade [61] finds disassembly trajectories by calculating the interactions between concavities and objects and trying to reduce those interactions, until the parts are separated. In the preimage backchaining approach to fine-motion planning [43, 46], the uncertainty of both sensing and control are modeled explicitly to find guaranteed plans to achieve a goal despite the high relative uncertainties present in assembly operations. Lastly, Pai and Donald [50] present a method for analyzing insertion of flexible parts by modeling them as compliant connections between rigid bodies.

The above methods are useful to verify the feasibility of individual assembly operations. Unfortunately, as in [34] they all require that a large number of assembly operations be generated and then tested; none supply a way to generate only those operations that satisfy the constraints.

1.3.4 Computational Geometry

Research in computational geometry addresses limited cases of geometric assembly sequencing. A survey of methods for finding separating motions for parts in two and three dimensions is given in [60]. In [48] lower bounds on the number of simultaneous translations

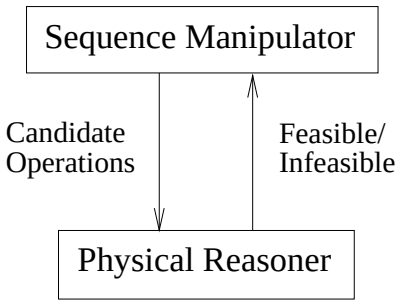

Figure 1.2: An Assembly Sequencing Architecture

necessary for separating objects are derived. Dawson [24] shows that two or more star-shaped objects can always be separated by translating the objects in different directions simultaneously. In addition, it is shown in [24] that for some assemblies of convex polyhedra, no individual parts are removable by a single translation.

Arkin et al. [3] use the concept of a monotone path between obstacles to deduce a removable subassembly and a single extended translation to remove it. However, the parts are limited to polygons in the plane, and the extension to three dimensions is not obvious. Pollack et al. [54] consider sequences of translations to separate polygons. The algorithm is limited to planar assemblies of two parts, but is able to find separating motions consisting of several distinct translations. See [31, 58] for additional special cases of assembly planning problems. Most of these methods are limited to two dimensions or allow only polyhedral parts; in addition, it is unclear how additional constraints, for instance arising from clamping forces or stability, can be included.

1.4 Contribution

Assembly sequences are highly constrained by geometry. However, little of the above research considers the tight interaction between sequencing and geometry explicitly. Some of the approaches assume geometric reasoning finishes before the sequences are generated (such as [25, 68]), while others consider geometric reasoning a black box to test operations (as in [34]). The more powerful geometric techniques are considered tools to test single operations. As a result, the sequence generation and geometric reasoning modules are loosely coupled, as in figure 1.2. Those systems that consider both sides of the problem simplify the reasoning or are inefficient for all but the smallest assemblies.

An approach is needed that allows efficient generation of assembly sequences that satisfy nontrivial geometric constraints. Geometric techniques must be considered and tested within the context of assembly sequencing on real assemblies. This thesis describes the following progress toward the goal of efficient geometric sequencing.

Experimental Testbed I describe a practical approach to generating assembly sequences strictly from a geometric model of the target assembly. This approach has been implemented as an assembly sequencing testbed called GRASP. GRASP is organized into modules that can be replaced individually to test new methods for solving subproblems. Basic modules accomplish geometric calculations using straightforward procedures. By substituting modules incorporating the more sophisticated algorithms given below, these new algorithms can be tested experimentally on real assemblies and compared to results with the basic modules. GRASP has planned assembly sequences for real assemblies of up to 42 parts.

Partitioning for Local Motions I introduce a succinct representation of the blocking relationships between parts in an assembly, called the *non-directional blocking graph* (or NDBG). An efficient algorithm is given to identify subassemblies that are locally free in an assembly, by constructing and analyzing a NDBG for the assembly. Local freedom is a necessary constraint on assembly operations, and in experiments it has proven to be a powerful pruning constraint. Specifically, consider an assembly A of n parts, with m contacts between them that can be described as k point–plane contact constraints. This type of contact includes most contacts in real assemblies. The procedure *PARTITION* determines whether a locally free subassembly of A exists, and finds one such subassembly and a legal motion direction in $O(mk^2)$ time when motions are restricted to translations, and in $O(mk^5)$ time for general rigid motions including rotations. When desired, both subassemblies can be constrained to be connected with the same time bound. The set of all s locally free subassemblies can be found in output-dependent time $O(msk^2)$ for translations and $O(msk^5)$ time for general rigid motions. *PARTITION* performs well in experiments on real assemblies.

Partitioning for Extended Motions I describe a variation of the *PARTITION* procedure that identifies a subassembly that can be completely removed by a single extended translation, for polyhedral assemblies. This procedure uses an extension of the NDBG. When the n parts of the assembly have a total of v vertices, the algorithm identifies a

single removable subassembly in time $O(n^2v^4)$. When applied to find the set of translations separating two parts, the algorithm requires time $O(v^4)$, which is optimal in the worst case.

Re-using Geometric Results Finally, a method is given to re-use geometric results, thereby reducing the geometric computation required to produce linear assembly sequences. Salient information is extracted from each geometric test and stored in a propositional logic expression based on the presence and absence of individual parts. The expressions are stored and later retrieved to answer similar geometric queries. The method can also be used to reduce the number of questions to a human when doing interactive assembly sequencing. Several versions of the expressions with increasing descriptive power have been implemented and tested as plug-in modules for GRASP, resulting in large gains in assembly sequencing speed.

1.5 Outline

The rest of this dissertation is organized as follows:

In Chapter 2, I define the geometric assembly planning and assembly sequencing problems using a configuration space formalization. Within this framework, I further define notions such as assembly operations, subassemblies, and several classes of assembly sequences. Representations of assembly sequences are described and their corresponding classes of sequences identified.

Chapter 3 describes a basic approach to geometric assembly sequencing and its implementation in an assembly sequencing testbed. From the three dimensional geometric models of a product, a contact graph is constructed and used to build an AND/OR graph representing possible assembly sequences. Geometric calculations test assembly operations for feasibility. An interface to an engineer incorporates human expertise when desired. The techniques are implemented in GRASP, a geometric assembly sequencing testbed used to perform experiments using different sequencing methods.

The next three chapters describe methods that can be used instead of the basic procedures to make the assembly sequencing process more efficient. Chapter 4 describes *PARTITION*, an algorithm to find all ways to feasibly decompose an assembly into two subassemblies according to local motion constraints arising from the contacts between parts.

The algorithm allows either translational or general rigid motions, and can impose a connectedness constraint on the subassemblies produced. Experiments on real assemblies are described for an implementation of the algorithm that is well-suited for practical use.

Chapter 5 describes a variation of the *PARTITION* algorithm that identifies a subassembly that can be removed by a single extended translation. The special case of finding separating motions between two polyhedra is shown to be worst-case optimal. An implementation and experiments with randomly generated assemblies are described.

Chapter 6 concentrates on *linear* assembly sequences, where one of the two subassemblies in each assembly task is restricted to be a single part. In this domain I describe *precedence expressions* (or PEs), whereby the geometric reasoner returns a symbolic description of the reasons for each result, allowing re-use of previous results. I give two versions of the method: one in which the PEs are inherited from assembly to subassembly, and one in which they are stored globally. The global PEs are a compact, implicit representation of the set of assembly sequences. Several types of PEs with increasing descriptive power are given. I analyze the theoretical complexity of the methods for limited cases, and show experimental results on real assemblies.

Finally, in Chapter 7 I conclude, stress the limitations and contributions of the work described here, and identify promising directions for future research.

Chapter 2

Geometric Assembly Planning

Assembly tasks are a particular case of manipulation tasks, which consist of a robot moving objects into a desired goal configuration. In this chapter I present a formal geometric definition of the manipulation planning problem, and then specialize it to yield a definition of the assembly planning problem. An assembly plan can then be defined as a solution to an assembly planning problem. A subproblem of assembly planning called the assembly sequencing problem is then described. While an assembly plan is a complete plan to construct a product from its constituent parts given a certain manufacturing environment, assembly sequencing only identifies the constraints arising strictly from the geometry and characteristics of the goal product itself. Thus assembly sequencing is independent of any particular set of assembly fixtures, robots or workers, and tools, but a given assembly sequence may not be feasible in a particular environment. In this framework, I define several subclasses of assembly plans and sequences, and describe some simple results about them. Finally, several representations of assembly sequences are given and their corresponding types of sequences identified.

2.1 Manipulation Planning

Robotic assembly occurs in a workcell similar to that shown in figure 2.1. A robot with the ability to affect other objects causes the objects to move from their initial positions into a goal configuration through a sequence of reaching, grasping, and carrying operations. An important feature of the workcell is that the objects cannot move on their own; they must be manipulated by the robot. Given a robot and other objects in an initial position,

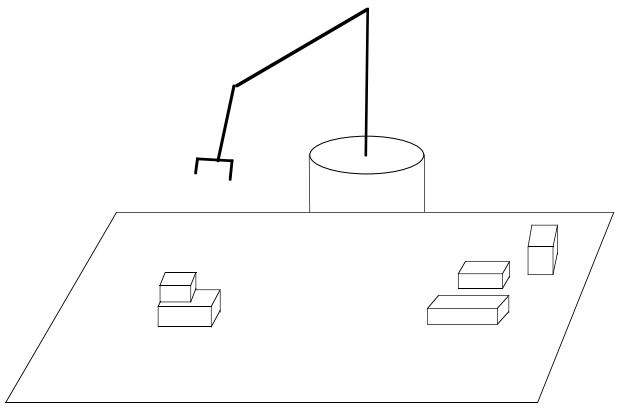

Figure 2.1: A robotic workcell

a manipulation task requires the robot to achieve a specified goal position through legal moves. The following formulation of the manipulation planning problem borrows heavily from [2] and [42].

2.1.1 Configuration Space

Consider a robot with r degrees of freedom, a set $O = \{P_1, \ldots, P_m\}$ of m rigid objects, and s fixed obstacles in a 3-dimensional workspace $\mathcal{W}$. Let

- $\mathcal{R}$ be the r-dimensional configuration space of the robot.
- $\mathcal{P}_i$ be the 6-dimensional configuration space of object P_i, for $1 \leq i \leq m$.
- $\mathcal{P} = \mathcal{P}_1 \times \mathcal{P}_2 \times \cdots \times \mathcal{P}_m$ be the $6n$-dimensional configuration space of all the objects.
- $\mathcal{C} = \mathcal{R} \times \mathcal{P}$ be the configuration space of the whole system. A configuration $q \in \mathcal{C}$ thus specifies the positions of the robot and all the objects.

Attach a reference frame F_i to each object P_i, and define $F_i \setminus F_j$ to be the relative transformation between F_i and F_j. Let $E_i(q)$ give the position of F_i in the world coordinate system in configuration q. Similarly attach a reference frame F_r to the gripper of the robot, and let $E_r(q)$ give the position of F_r in the world coordinate system in configuration q. $R(q)$ and $P_i(q)$ denote the subsets of $\mathcal{W}$ occupied by the robot and object P_i, respectively, in configuration q. $R(q)$ and $P_i(q)$ are bounded three-dimensional manifolds with boundary[1].

[1] A subset M of $\mathcal{W}$ is an m-dimensional *manifold with boundary* if every point $x \in M$ has a neighborhood V such that the set $V \cap M$ is homeomorphic to either an open ball of R^m or a closed half-space of R^m. This restriction rules out some pathological cases of part geometry.

Although this formulation applies only to a single robot, multiple robots could be included with some added complication.

Two objects P_i and P_j *interfere* when their interiors intersect, i.e. in configurations q for which $int(P_i(q)) \cap int(P_j(q)) \neq \emptyset$; they are *in contact* if they do not interfere and their boundaries intersect, or $bound(P_i(q)) \cap bound(P_j(q)) \neq \emptyset$. The same relations are defined similarly for the robot and an object. Then ILLEGAL is the open subset of $\mathcal{C}$ in which an object interferes with an obstacle, another object, or the robot. Let LEGAL $= \mathcal{C} -$ ILLEGAL be the closed set of all non-interfering configurations.

2.1.2 Grasping and Stability

Because the objects cannot move on their own, two constraints must be expressed: the objects only move when moved by the robot, and they must be in stable configurations when not grasped by the robot. Assume that the robot can only move an object by grasping it rigidly; for example, no pushing or dropping actions are allowed. In general, the graspability of any one object might depend on the locations of all the other objects. In a given configuration the robot might be able to grasp and move different sets of objects, including the empty set (no objects), depending on the forces exerted. Let GRASPABLE(q) be the set of all sets of objects that the robot can grasp in configuration q. Since the robot can always choose to grasp no objects, the empty set $\emptyset$ is always in GRASPABLE(q). To affix a set of objects $S \in$ GRASPABLE(q) to the robot gripper, the robot executes the operation GRASP(S). To release the objects, the robot simply executes GRASP($\emptyset$).[2]

Figure 2.2 shows part of a configuration (call it q_1) that illustrates the definition and use of the function GRASPABLE. From its position in q_1, the robot cannot grasp `A` or `B` alone, but it can grip the two of them together by exerting an inward force on its jaws. However, moving {`A`,`B`} will necessarily move object `C`, so the set {`A`,`B`} cannot be included in GRASPABLE(q_1). Hence GRASPABLE(q_1) = $\{\{\mathtt{A}, \mathtt{B}, \mathtt{C}\}, \emptyset\}$.

When an object is not currently being grasped, it must be in a stable position in the environment. However, once again the stability of an object could depend on the locations of all the other objects. For any set of objects S, let $\mathcal{P}_S = \prod_{P_i \in S} \mathcal{P}_i$ be the space of configurations of the objects in S, and let STABLE$_S$ denote the subset of stable, noninterfering configurations of $\mathcal{P}_S$. In other words, STABLE$_S$ is the set of stable configurations for objects

[2] In reality, physical actions are needed to grip an assembly, such as exerting a gripping force or turning on a suction gripper. However, I approximate grasping as an on-off switch.

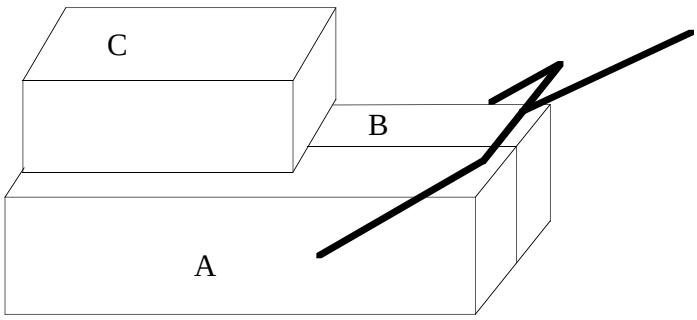

Figure 2.2: Grasping a subassembly

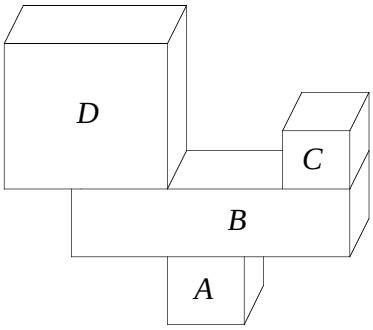

Figure 2.3: An unstable arrangement of objects

S, without the support of any other object. Then let $\pi_S : \mathcal{C} \rightarrow \mathcal{P}_S$ be a function mapping configurations of the workcell into configurations of the objects in S. To grasp a set of parts S, the robot must be in a configuration where the rest of the objects $O \setminus S$ are stable, i.e. a configuration q such that $\pi_{O\setminus S}(q) \in \text{STABLE}_{O\setminus S}$. Note that STABLE_O is the set of object configurations in which all the objects are in stable configurations, so to execute $\text{GRASP}(\emptyset)$ (an ungrasp), the current object configuration $\pi_O(q)$ must be in STABLE_O.

Figure 2.3 shows a partial configuration q_1 of objects to illustrate the sets STABLE_S. With no influence from the robot, object `D` is heavy enough to tip `B` and `C` over; hence $\pi_O(q_1)$ is not in STABLE_O. However, the configuration consisting of blocks `A`, `B`, and `C` is stable. Hence, if the robot were grasping block `D`, the configuration would be stable, so $\pi_{O\setminus\{\texttt{D}\}}(q_1) \in \text{STABLE}_{O\setminus\{\texttt{D}\}}$.

$\text{GRASPABLE}(q)$ contains all the sets of objects that *can* be grasped in a configuration q, while $\text{STABLE}_{O\setminus S}$ gives the configurations of stable objects when set S is grasped. A configuration q is stable if it is in LEGAL and if grasping one of the graspable sets leaves

the ungrasped objects stable. Define STABLE as the set of such configurations:

$$\text{STABLE} = \{q \in \text{LEGAL} \mid \exists S \in \text{GRASPABLE}(q),\ \ \pi_{O\setminus S}(q) \in \text{STABLE}_{O\setminus S}\}$$

In the real world, objects also need to be stable when undergoing motion and when insertion forces are applied; these constraints are not considered here.

The computation of GRASPABLE and STABLE_S will not be discussed in depth here. A great deal of literature addresses grasping issues (see for instance [20, 37, 53]), although few papers consider grasping of possibly-unstable assemblies. Palmer [51] shows that checking the stability of a set of polygonal objects in a vertical plane is NP-hard both with and without friction. Boneschanscher et al. [11] give a stability test for limited types of contact that runs in polynomial time, and Blum et al. [9] describe a numerical stability test whose behavior is hard to characterize. However, it is clear that practical stability tests for "normal" assemblies must be developed before truly autonomous assembly planning can be realized.

We can impose the simple necessary condition on grasp configurations that the union of the grasped objects and the robot must be connected. If the union of the obstacles is a connected set (as in real robotic workcells), then the union of the ungrasped objects and the obstacles must be connected in stable configurations. However these constraints are obviously not sufficient for stability.

2.1.3 Manipulation Paths

Two distinct types of motions can be performed in this system, depending on whether the robot is grasping any objects during the motion. In a *transit path*, the robot moves without affecting the positions of the objects, which must be in a stable arrangement. During a *transfer path*, a subset of the objects is grasped and moves with the robot. The grasped objects stay rigidly attached to the end effector of the robot throughout the transfer path, while the ungrasped objects do not move and must be stable. A *manipulation path* is an alternating sequence of transit and transfer paths, in which the endpoint of one path coincides with the starting point of the following one. Formally,

Definition 2.1 *A* transit path *is a continuous map* $\tau : [0,1] \to \text{STABLE}$ *such that:*

- $\forall s \in [0,1] : \pi_O(\tau(s)) = \pi_O(\tau(0))$ *(the objects do not move).*
- $\pi_O(\tau(0)) \in \text{STABLE}_O$ *(the objects are all stable).*

Definition 2.2 *A* transfer path *is a continuous map* $\tau : [0,1] \rightarrow \text{STABLE}$ *for which there exists a* grasped set *of objects* S *such that:*

- $\forall s \in [0,1] : S \in \text{GRASPABLE}(\tau(s))$ *(the grasped set is graspable).*
- $\forall s \in [0,1] : \pi_{O \setminus S}(\tau(s)) \in \text{STABLE}_{O \setminus S}$ *(the ungrasped objects are stable).*
- $\forall s \in [0,1] : \pi_{O \setminus S}(\tau(s)) = \pi_{O \setminus S}(\tau(0))$ *(the ungrasped objects do not move).*
- *For each* $P_i \in S$*, there exists a constant transformation* T_i *such that* $\forall s \in [0,1] : E_i(\tau(s)) \setminus E_r(\tau(s)) = T_i$ *(the grasped set is rigidly attached to the robot gripper).*

Every transit and transfer path lies in a submanifold of STABLE whose dimension is r.

Definition 2.3 *A* manipulation path *is an finite alternating sequence* $(\tau_1, \tau_2, \ldots, \tau_{2p+1})$ *such that:*

- $\forall j \in [1, 2p] : \tau_j(1) = \tau_{j+1}(0)$ *(the endpoint of one path is the start of the next).*
- $\tau_1, \tau_3, \ldots, \tau_{2p+1}$ *are transit paths.*
- $\tau_2, \tau_4, \ldots, \tau_{2p}$ *are transfer paths.*

At the beginning of every transit path, the robot executes a $\text{GRASP}(\emptyset)$ *operation. At the beginning of every transfer path, the robot grasps the grasped set for that path.*

A manipulation path consists of alternating move and carry actions by the robot. Note that another formulation might allow two transfer paths to follow one another. For instance, in figure 2.4 the robot has been grasping the set {A, B}, and in the configuration shown it executes GRASP({A}) (setting B down on C and D) and continues the downward motion, holding A. A transit path cannot intercede because A is not in a stable position. Such motions will not be considered here, although the extension to handle them is straightforward.

A *manipulation planning* problem is specified by an initial configuration q_I and a set of acceptable goal configurations $Q_G \subset \text{STABLE}$. A solution to the problem is a manipulation path $(\tau_1, ..., \tau_p)$ such that $\tau_1(0) = q_I$ and $\tau_p(1) \in Q_G$.

For examples of manipulation problems, see [2] and [42, chap. 11].

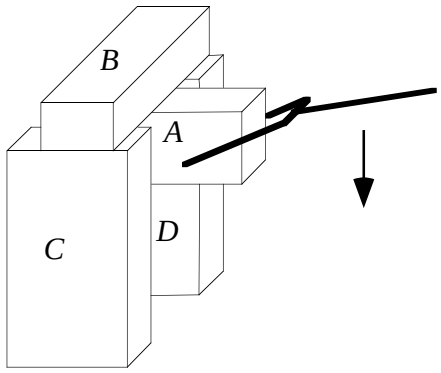

Figure 2.4: A configuration between two transfer paths

2.2 Assembly Planning

An assembly planning problem is an instance of the manipulation planning problem. Some of the objects are distinguished as parts of the goal assembly, while the rest are tools to be used to assemble the parts[3]. The initial configuration presents the parts in an initial, unassembled state along with the other objects. The goal is any configuration in which the product is completely assembled, with no other objects stuck inside it.

The m objects are divided into an assembly A of n *parts*, and $m-n$ *tools*, which include fixtures, clamps, wrenches, etc. For convenience, let the parts be objects $P_1, \ldots, P_n$ and the tools $P_{n+1}, \ldots, P_m$. In their initial position, the parts are unassembled—far enough away to be out of the influence of each other and the tools. To formalize this, let two sets of objects S_1 and S_2 be *separated* in configuration q when there exists a plane such that $\bigcup_{i \in S_1} P_i(q)$ is on one side of the plane, and $\bigcup_{j \in S_2} P_j(q)$ is on the other side of the plane. We allow the objects to touch the separating plane, so that objects in contact can still be separated. The robot is separated from a set of objects when a parallel condition holds.

The initial configuration in an assembly planning problem is a configuration q_I such that the set of all parts is separated from the set of all tools and from the robot in q_I, and furthermore all pairs of parts $P_i, P_j \in A, i \neq j$ are separated in q_I.

In an assembly planning problem, all the parts must be in their final relative positions in a goal configuration. In addition, the other objects, the robot, and the fixed obstacles must not be "stuck" inside the assembled parts. The latter constraint holds when a path

[3]In an extended view of the assembly planning problem, the selection and design of the tools could be included in the planning process. Here I consider only the simpler problem when the tools are specified.

exists to remove the assembly from the rest of the system while moving neither the parts relative to each other nor the robot and tools. I will use part P_1 as a reference to define the positions of the rest of the parts in the goal configuration. For each part $P_i, i \in [2, n]$, let T_i^G be a transformation relating the position of part P_i to part P_1 in the final assembly. Then ASSEMBLED is the set of all configurations q in STABLE such that:

- $\forall i \in [2, n], E_i(q) \setminus E_1(q) = T_i^G$ (the parts are in their final relative positions).
- A path exists in LEGAL to rigidly move the assembly from its position in q to a position separated from the robot, tools, and obstacles.

In a real assembly workcell, the tools must also be left in a state that allows them to be used to construct the next assembly. This constraint is not formulated here.

An *assembly planning* problem is a manipulation planning problem with n distinguished parts, where the initial configuration satisfies the constraints above and the goal is ASSEMBLED. An *assembly plan* is a solution to an assembly planning problem.

Because it is just a special case, the assembly planning problem can obviously be reduced to the manipulation planning problem, and thus a general manipulation planner could solve an assembly planning problem (if one existed).

2.3 Assembly Sequencing

In this section I describe assembly sequencing, in which the motions of the parts of an assembly are planned only with respect to each other, without considering the abilities of the robot or the effects of the fixtures. Assembly sequencing identifies the constraints on assembly plans arising strictly from the geometry and characteristics of the product itself; it is independent of any particular set of assembly fixtures, robots or workers, assembly orientation, and tools. As a result, the existence of a feasible assembly sequence for an assembly does not guarantee it can be manufactured, since only a subset of the constraints are taken into account. On the other hand, assembly sequence analysis can be applied early in the design process, before a manufacturing scheme is chosen, and possibly even when the assembly design is not finished. I describe the relation between assembly sequencing and assembly planning, and define several types of assembly sequences.

2.3.1 Definition

Consider an assembly of n rigid parts $A = \{P_1, \ldots, P_n\}$ in a 3-dimensional workspace $\mathcal{W}_A$ with no obstacles. Because there are no obstacles in $\mathcal{W}_A$, only the relative positions of the parts are significant. Let $\mathcal{P} = \mathcal{P}_2 \times \mathcal{P}_3 \times \cdots \times \mathcal{P}_n$ be the $6(n-1)$-dimensional composite configuration space of the parts of A in $\mathcal{W}_A$ with respect to the coordinate frame attached to part P_1. Let $\text{LEGAL}(\mathcal{P})$ be the subset of $\mathcal{P}$ in which the interiors of no two parts intersect. An *assembly path* is a continuous map $\tau : [0,1] \rightarrow \text{LEGAL}(\mathcal{P})$.

Define the part position functions $E_i(q)$ and the *separated* predicate for configurations $q \in \mathcal{P}$ in the same way as for $\mathcal{C}$ in the previous two sections. Then an *unassembled configuration* for an assembly sequencing problem is a configuration in which all parts are pairwise separated; call the set of such configurations Q_I. Because the position of part P_1 is fixed, the parts are all in their relative goal positions for one $q_G \in \mathcal{P}$. The *assembled configuration* $q_G \in \mathcal{P}$ is such that $\forall i \in [2,n], E_i(q_G) = T_i^G$.

An *assembly sequencing* problem is specified by an assembly A of n parts and an assembled configuration q_G. A solution to an assembly sequencing problem consists of an unassembled configuration $q_I \in Q_I$ and an assembly path τ such that $\tau(0) = q_I$ and $\tau(1) = q_G$. A solution to an assembly sequencing problem is called an *assembly sequence*.

Note that the unassembled configuration q_I can be chosen as part of an assembly sequence. In the real world this corresponds to placing the part feeders in the most convenient locations. Formally, since the parts must be pairwise separated by planes, any initial configuration could be transformed to any other by spreading the parts far enough apart and swapping their positions until the desired configuration is reached, then contracting them back in. Hereafter I will consider an assembly sequencing problem to be specified by just the goal assembly A.

2.3.2 Relation to Other Work

The assembly sequencing problem as stated here is purely geometric. Similar problems are called "assembly sequence planning" by Homem de Mello [34] and "assembly planning" by Wolter [68]. Homem de Mello takes a more applied approach to the problem, including stability and "mechanical" as well as geometric constraints on assembly sequences. However, only very weak stability constraints can be applied to an assembly sequence without information about the environment in which it is executed. For instance, many unstable

assemblies are stable in some orientations, in a stabilizing fixture, or when grasped correctly. Requiring that subassemblies be connected is a common constraint associated with stability, but even that can be remedied using a fixture: consider a car body being lowered onto two wheel-axle subassemblies. Homem de Mello's mechanical constraints concern the geometric aspects of fasteners—which are considered individual parts here—and tools, as well as physical constraints such as clamping forces that are difficult to formalize but obviously important for an applied assembly planning program.

Wolter defines the assembly sequencing problem in terms of n workspaces. Each part starts in a separate workspace, and the parts can be transferred between workspaces to assemble them. This more easily formalizes the notion of the parts being separated in their initial positions, but weakens the connection to assembly planning with a robot (below).

The geometric nature of the assembly sequencing problem stated here makes it fully reversible. In other words, an assembly sequence is the reverse of a valid *disassembly sequence* for the same assembly. A number of assembly planners [34, 39, 44, 68], including GRASP, take advantage of this fact by planning for disassembly. Another reason for doing disassembly planning is that the assembled configuration is more tightly specified than the unassembled configuration; this aspect of the state space lends itself to backward planning. In the real world, some assembly operations are not the reverse of disassembly operations, due to mechanical and stability constraints and non-rigid parts such as springs, snap-fit parts, and fluids. However, if a disassembly operation is defined as the reverse of a feasible assembly operation, then disassembly planning is a valid approach even for non-rigid parts. In most of this thesis, assembly planning and disassembly planning will be discussed interchangeably.

The assembly sequencing problem with an arbitrary number of parts has been shown to be PSPACE-hard independently by Natarajan [48] and Wolter [68]. Natarajan [48] shows that the problem remains PSPACE-hard when the parts are limited to a constant number of vertices.

Theorem 2.4 *The assembly sequencing problem is* PSPACE*-hard.*

2.3.3 Relation to Assembly Planning

Given an assembly sequence for a product A and a particular robotic workcell, the robot might or might not be able to produce the relative motions required by the assembly sequence. If the robot can realize the relative motions called for in the assembly sequence, the

corresponding assembly plan is said to *execute* the assembly sequence. In fact, a number of different robot plans might execute the same assembly sequence. From a different perspective, we might consider the set of all robotic workcells that can execute a given assembly sequence. A production engineer might take this view when designing a manufacturing cell for a product.

Formally, consider a robotic workcell for A with a particular robot and fixtures (for a total of m parts and fixtures) having a composite configuration space $\mathcal{C}$. Let $\pi_A : \mathcal{C} \rightarrow \mathcal{P}$ be the function projecting a configuration q of the whole system into the configuration $\pi_A(q)$ of the parts relative to P_1. Then an assembly plan $\tau_\mathcal{C}$ *executes* the assembly sequence $\tau_\mathcal{P}$ of A when there exists a continuous nondecreasing function $\gamma : [0,1] \rightarrow [0,1]$ such that $\gamma(0) = 0$, $\gamma(1) = 1$, and

$$\forall s \in [0,1] : \pi_A(\tau_\mathcal{C}(s)) = \tau_\mathcal{P}(\gamma(s)).$$

In other words, the parts follow the same path relative to each other in both the assembly sequence and the executing assembly plan.

2.4 Types of Assembly Sequences

An assembly sequence τ can be divided into an equivalent list of assembly paths $(\tau_1, \ldots, \tau_m)$ accomplishing the same motions. The τ_i are called *operations*. This representation allows additional restrictions to be placed on the operations τ_i, thereby defining classes of assembly sequences. Several such classes will be considered below, including binary, monotone, linear, and connected assembly sequences. These are categories of sequences that lend themselves both to execution by a robot or human and to automatic generation. Much of the terminology below is taken from Natarajan [48] and Wolter [68].

2.4.1 Number of Hands

Let τ_i be an operation in an assembly sequence for assembly A. A *moved set of* τ_i is a maximal set of parts S such that the relative positions of parts in S stay constant during τ_i. The moved sets of any operation are a partition of the parts of the assembly. An operation τ_i is *m-handed* if there are m moved sets of τ_i. To execute an m-handed operation would require one "hand" to move each set of parts along its trajectory, where the table counts as one hand. An assembly sequence is m-handed if it can be divided into m-handed operations. A robotic workcell with r independent robots can execute $(r+1)$-handed sequences. Natarajan [48]

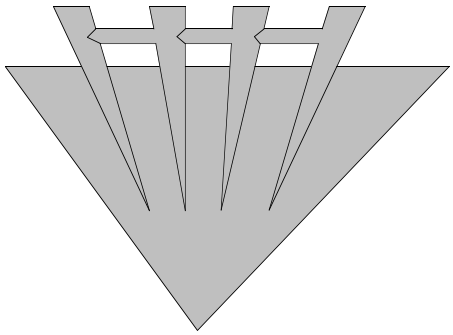

Figure 2.5: An assembly that requires n hands to build [48]

shows that in general n simultaneous motions may be needed to build an assembly of n parts (figure 2.5 shows one such assembly made of star-shaped[4] parts). However, most real products can be built using a much smaller number of hands, usually with just two (a robot and a table). An assembly that requires more than two hands to build is a prime candidate for redesign [12].

A two-handed assembly sequence is also called *binary*. Generating non-binary assembly sequences requires reasoning about the simultaneous relative motions of more than two subassemblies. Because such reasoning is difficult and most real products can be built with two hands, all assembly planners to date have been restricted to binary assembly sequences. Even for a product that cannot be constructed with a binary assembly sequence, maximizing the number of binary assembly operations in the sequence will minimize manufacturing costs.

2.4.2 Monotonicity

The number of hands needed to execute an assembly sequence is only one aspect of its difficulty to generate and execute. Another is the number of intermediate positions that parts may take before they are placed in their relative goal positions. The class of assembly sequences without any intermediate positions is a special case. A *subassembly* in configuration q of a sequence τ is a maximal set of parts S that are in their final relative positions and that stay in those positions until the end of the sequence:

- For all parts $P_i, P_j \in S$ and all configurations q' following q in τ, $E_i(q) \setminus E_j(q) = T_i^G \setminus T_j^G$.

[4] An object S is *star-shaped* if there exists a point $p \in S$ such that the line segment connecting p to every element of S is completely contained in S.

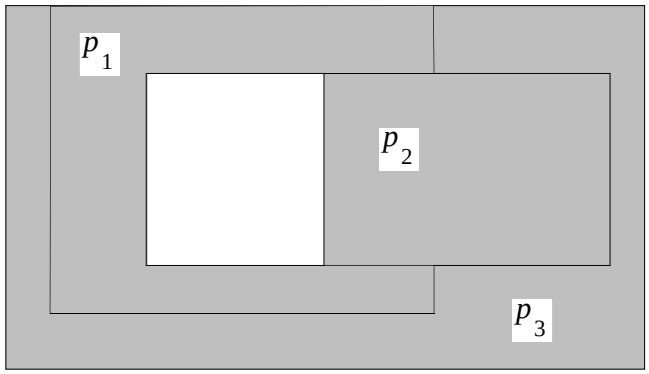

Figure 2.6: An assembly with no monotone binary assembly sequence [68]

Each part is a trivial subassembly at the start of a sequence, and until it becomes part of a larger subassembly.

An assembly sequence is *monotone* if each operation requiring m hands joins m subassemblies to make a larger subassembly. In other words, a monotone sequence consists of operations placing parts into their final positions relative to each other. Formally, for each operation τ_i in a monotone sequence, let τ_i be m-handed. Then there is a set of parts S_i such that

- S_i is a subassembly in configuration $\tau_i(1)$.
- S_i is the union of m subassemblies $S'_1, \ldots, S'_m$ in $\tau_i(0)$.
- All but one of the subassemblies $S'_1, \ldots, S'_m$ are moved sets of τ_i.

Monotone assembly sequences are simpler to compute than nonmonotone sequences, because they do not require the identification of intermediate positions for subassemblies. In a monotone binary sequence, each operation brings exactly two subassemblies together; hence a monotone binary sequence consists of $n - 1$ operations.

Note that the monotonicity of a sequence means little unless the number of hands is stated. Any assembly sequence for an n-part product can be written as a monotone sequence in which all the parts are mated in one long n-handed operation. Consider the latch assembly shown in figure 2.6 from [68]. It could be assembled with a non-monotone binary sequence placing P_2 in P_1, then $\{P_1, P_2\}$ inside P_3, then sliding P_2 right. This same set of motions could be described as a single monotone 3-handed operation.

The relationship between monotonicity and handedness is an interesting one. Let an m-handed assembly be an assembly that can be built with m hands, and an m-handed

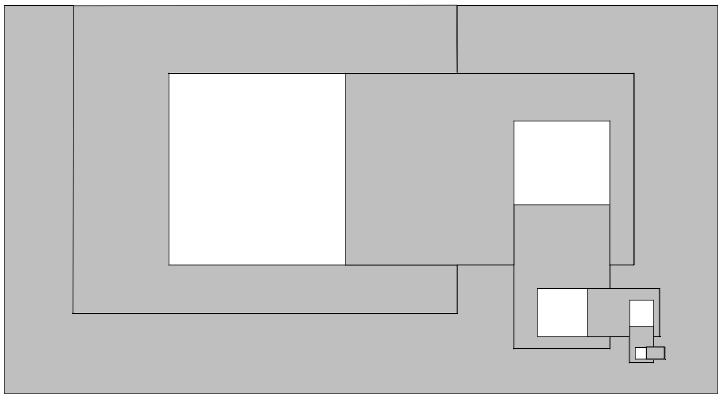

Figure 2.7: An $(m+1)$-part latch assembly

monotonic assembly be an assembly for which there exists an m-handed monotonic assembly sequence. It can be shown that there is no inclusion relation between the set of m-handed monotone assemblies and the set of $(m-1)$-handed assemblies. In other words, for any m, there exists an m-handed monotonic assembly that is not $(m-1)$-handed, and there is an $(m-1)$-handed assembly that is not m-handed monotonic.

Natarajan [48] proved the first part by showing that there exist assemblies with m parts that cannot be assembled with less than m simultaneous motions (figure 2.5).

To prove the second part, consider the extended latch assembly shown in figure 2.7. It consists of m interlocking pegs fitted inside an outer shell. The latch assembly is a 2-handed assembly, since it can be built by inserting the smallest peg into the second smallest, inserting the resulting subassembly into the next largest, and so on, then inserting the pegs together into the hole, and finally latching each peg assembly in reverse order out into its sub-hole.

However, assume an m-handed monotone sequence exists to assemble the latch. No peg can be inserted before the next larger peg, so during some operation τ_i in the sequence the assembly must pass through the configuration q_1 shown in figure 2.8. Every operation in a monotone sequence creates a subassembly, but no part is in its final position relative to any other in q_1. Therefore τ_i must be the first operation in the sequence. Hence τ_i has moved all $m+1$ parts from an unassembled configuration to configuration q_1. Operation τ_i is thus $(m+1)$-handed, which is a contradiction.

Theorem 2.5 *For any m, there is an assembly with $m+1$ parts that is 2-handed but not m-handed monotonic.*

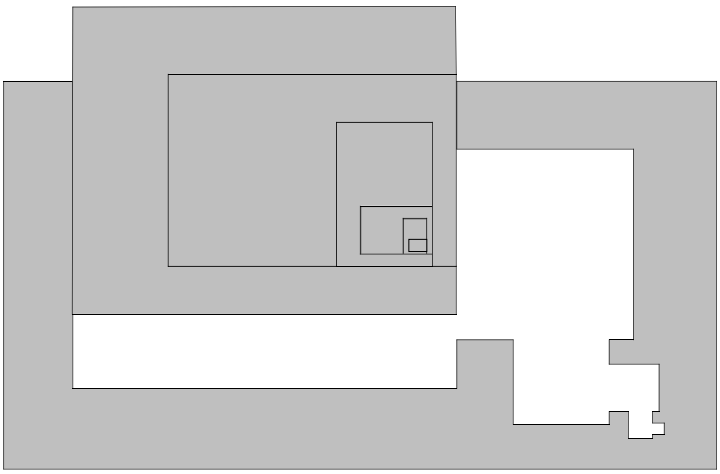

Figure 2.8: An intermediate step in assembling the latch

Many assembly sequencers assume monotonicity, but not all. For instance, Hoffman [33] describes a system that generates disassembly sequences from the boundary representations of the parts. If an assembly cannot be disassembled in a single motion, a subassembly is chosen to move to an intermediate position that might allow disassembly. The intermediate positions are chosen heuristically based on the geometry of the parts and certain features (such as the center of a hole or protrusion) that may be lined up. The method does not always succeed, but it works in many practical cases.

2.4.3 Linearity

Further restrictions on assembly sequences are possible to simplify the assembly sequencing problem. One that is imposed by several assembly planning systems (see [66, 68] and Chapter 6) is linearity. A binary assembly sequence is *linear* if one of the two moved sets of each operation is a single part. Hence a linear monotone sequence consists of $n - 1$ operations, each mating a single part with a subassembly. Figure 2.9 shows a monotone binary assembly with no linear assembly sequence. Under the linear assumption, a disassembly planner need only consider removing single parts, instead of identifying removable subassemblies. This simplifies the planning process considerably and allows additional optimizations, as will be seen in Chapter 6.

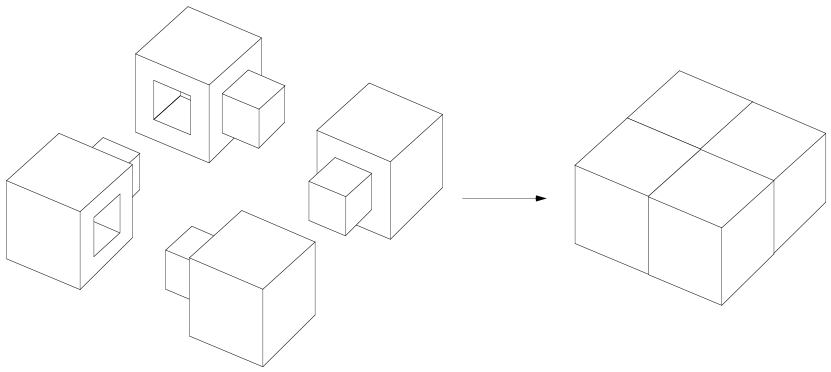

Figure 2.9: An assembly in which no single part can be removed

2.4.4 Connectedness

The restrictions on assembly sequences given above are all based on the type of part motions allowed. In contrast, an often useful constraint is that the subassemblies constructed in a monotone assembly sequence be connected. Wolter [69] calls such assembly sequences *contact-coherent*. Connected subassemblies make sense for executing a sequence, since it is difficult to grasp or maintain the stability of a subassembly when it is not even connected. As noted in section 2.1, fixtures can usually be designed to stabilize even unconnected subassemblies, but connectedness is still a useful heuristic. In addition, enforcing connectedness of subassemblies helps to reduce the combinatorics of non-linear sequence planning. For instance, to find an operation Homem de Mello [34] generates all connected subassemblies and then tests each for removability; if unconnected subassemblies were allowed, this approach would become impractical for much smaller assemblies than when the connected constraint is included. Figure 2.10 shows an assembly that cannot be built with a connected binary assembly sequence.

For each category of assembly sequence described in this section, we can define a corresponding class of assembly plans based on the correspondence between sequences and plans. Thus a monotone binary assembly plan is an assembly plan that executes a monotone binary assembly sequence.

Much of the literature on motion of objects has relevance for assembly planning but cannot all be summarized here. A survey of methods for separating sets in two and three dimensions is given in [60]. For more types of assembly sequences, assemblies that can be

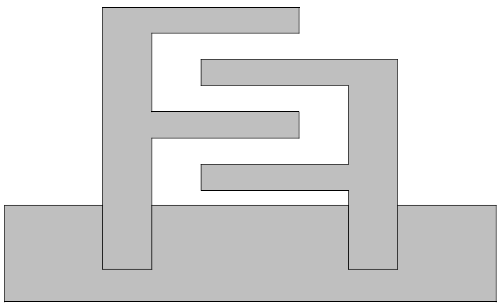

Figure 2.10: An assembly with no connected binary assembly sequence

constructed under various restrictions, and the complexity of certain object-motion problems, see for instance [24, 48, 51, 54, 60, 68].

2.5 Representations of Assembly Sequences

In an assembly sequencing system, the choice of representation for assembly sequences can be crucial. This section defines several equivalence classes of sequences based on part motions, and then describes data structures along with the classes of sequence they can represent.

2.5.1 Equivalence Classes of Assembly Sequences

Representing a sequence as a continuous function from time into a multi-dimensional space of part positions is not adequate. Doing so would give too much detail without making explicit the important events in the sequence. As a result, most assembly sequencers define equivalence classes of sequences based on the order of part mating operations, and then represent equivalence classes of sequences.

Position Equivalence For many purposes the order of mating operations between parts is more important than the exact motions the parts follow in the operations. For instance, several subassemblies might be built by separate manufacturers, then shipped to a common factory for final assembly. For such a product, constructing the subassemblies individually is quite important, while the motions required to build each subassembly can be decided by its respective manufacturer. In such cases equivalence classes of assembly sequences can be considered based on the parts moved in each operation.

Let $\tau = (\tau_1, \ldots, \tau_n)$ and $\tau' = (\tau'_1, \ldots, \tau'_n)$ be two assembly sequences of the same length for an assembly A. Then τ and τ' are *position equivalent* when respective operations leave the assembly in equal states, i.e. when $\tau_i(0) = \tau'_i(0)$ for all $i \in [1, n]$. Two position equivalent nonmonotone sequences may have different motions but they use the same intermediate positions.

Order Equivalence Position equivalent monotone sequences create the same subassemblies in the same relative positions. However, in monotone assembly sequences the relative positions of the subassemblies may not be important either. Define two monotone sequences τ and τ' to be *order equivalent* when respective operations τ_i and τ'_i create the same subassemblies.

Many applications do not distinguish between order equivalent sequences, since they differ only in the part mating motions for individual operations. These mating tactics can sometimes make a large difference in the quality of an assembly sequence, but they only affect the difficulty of single operations, and can be optimized individually or left unspecified to allow adjustment when the full assembly plan is created. As a result, after this chapter I will use the term *assembly sequence* to refer to a class of order equivalent assembly sequences. Most papers on assembly sequencing that consider monotone sequences take this view.

Subassembly Equivalence Finally, in some applications the order in which subassemblies are created is unimportant. If operation τ_i mates subassemblies S_1 and S_2, for instance, the order of construction of S_1 and S_2 makes little difference. S_1 could be built first, or S_2, or the operations accomplishing their construction could be interweaved. Because the sequence is monotone, the parts of S_1 and S_2 do not interact until τ_i. For some products it is necessary to perform a measurement or other operation on one subassembly before constructing another; this was common practice before the 20th century. However, such designs are discouraged in modern manufacturing because they raise the cost of assembly.

Let two monotone assembly sequences be *subassembly equivalent* if they create the same set of subassemblies, possibly in different orders. A one-to-one correspondence can be made between the operations of two subassembly equivalent sequences, in which each pair of operations establish the same subassembly from the same smaller subassemblies.

Other equivalence relations between assembly sequences are possible in practice. For instance, one might consider two operations to be equivalent when their respective part

motions have the same final trajectory. Another useful view is to define part *clusters*, such that two sequences are equivalent when they place parts from a cluster in different orders. For instance, the bolts in a bolt circle rarely need to be placed in a specific order, so it is wasteful to generate all possible orders of assembly. Although the part clusters must be found heuristically or input by a human, this view can reduce the complexity of assembly sequencing in many practical cases (see [10] for example).

The next subsections consider representations for classes of assembly sequences.

2.5.2 State Graphs

State-space graphs [49] can easily be adapted to represent assembly sequences. Salient features of an assembly configuration are identified, and the state of the assembly process is defined in terms of those features. For instance, the features may be just the positions of the parts at the end of an operation. One set of feature values (the *unassembled state*) represents the unassembled configuration, and another set of values (the *assembled state*) represents the assembled configuration. An assembly operation is represented by an arc in the graph from one state to another. Any path through the graph from the unassembled state to the assembled state represents an assembly sequence. The state graph thus represents a space of possible assembly sequences.

A natural set of features for order equivalent assembly sequences was identified by Bourjault [14] and later used by De Fazio and Whitney [25]. They define the state of an assembly based on the liaisons that have been established thus far in an assembly sequence. When a liaison between two parts is established, the two parts are in their final relative position. Thus a set of liaisons that have been established define a set of subassemblies. The state with no liaisons established is the unassembled state, and the state with all liaisons established is the assembled state. An assembly operation mating two (or more) subassemblies is represented by an arc in the graph from one liaison state to another. A path from the unassembled state to the assembled state represents a class of order equivalent sequences.

For example, figure 2.11 shows a liaison diagram for the crate assembly of figure 1.1, and figure 2.12 shows a state graph representing some binary monotone assembly sequences for the crate. Each liaison is established when its corresponding box is filled in. The unassembled state is at the bottom of the figure, and the assembled state at the top. States that do not occur in any assembly sequence are not shown.

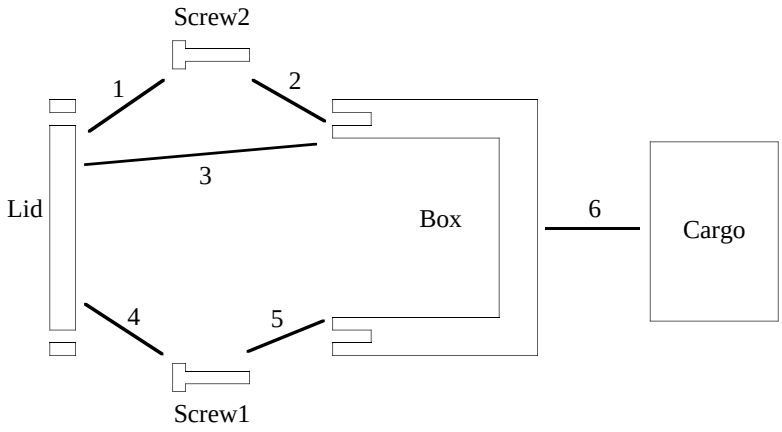

Figure 2.11: A liaison diagram for the crate assembly of figure 1.1

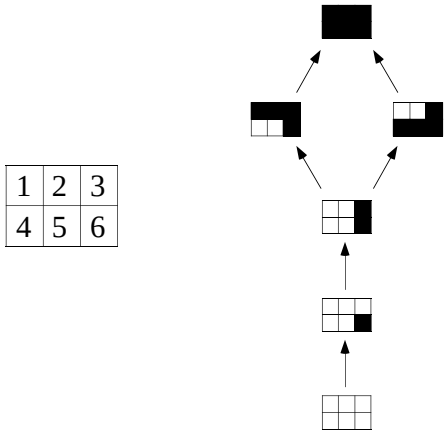

Figure 2.12: A liaison state graph for the crate

State-space graphs are not limited to representing monotone sequences. For instance, an extra feature or liaison can be created for a subassembly that has an intermediate position. This liaison can be established and then broken by a later operation. More generally, each feature can represent the position of a part in the assembly configuration; the resulting state graph represents position equivalent assembly sequence classes.

State graphs can require a large amount of storage in some cases. For instance, in a liaison state graph, each liaison is either established or not in any one state, yielding a maximum of 2^m states for m liaisons. For representations in which assembly state features take continuous values, the state graph is obviously infinite.

2.5.3 AND/OR Graphs

The AND/OR graph is the standard AI tool to represent problems that can be decomposed into subproblems with few or no interactions [49]. Homem de Mello and Sanderson first used AND/OR graphs to represent subassembly equivalent classes of assembly sequences [35]. The state graph differentiates between orders of subassembly construction; the AND/OR graph does not, and consequently is usually more compact than the state graph for the same product.

Each node in the AND/OR graph represents a subassembly that might be constructed in an assembly sequence for the product. An AND-arc represents the operation bringing several child subassemblies together to make the parent, while OR-arcs give different ways of creating the same parent subassembly. In a binary AND/OR graph, each AND-arc specifies two child subassemblies; only binary AND/OR graphs have been used in assembly sequencers to date. The root of the graph is the final assembly and the leaves are subassemblies with only one part in each. Thus each AND-subtree of a full AND/OR graph represents a subassembly equivalant class of assembly sequences. Figure 2.13 shows a binary AND/OR graph for the crate representing the same assembly sequences as the state graph in figure 2.12. Subassemblies that do not occur in any assembly sequence are not shown. GRASP (see the next chapter) adopts binary AND/OR graphs to represent sets of assembly sequences.

In the worst case, an AND/OR graph for an assembly with n parts can have $2^n - 1$ nodes. The number of AND-arcs is $O(3^n)$ for binary assembly sequences [68], and higher for nonbinary sequences. As with state graphs, the worst case happens with an highly unconstrained assembly, such as a printed circuit board and chips that can be placed in any order.

2.5.4 Implicit Representations

A state graph or AND/OR graph might require a very large amount of storage to represent a set of assembly sequences generated by an assembly sequencing program. An alternative is to represent the sequences implicitly by a set of *sequence rules* restricting the operations in a sequence. Depending on the assembly and the expressive power of the rules, the set of rules may be quite compact.

Implicit representations of assembly sequences will not be considered in depth here.

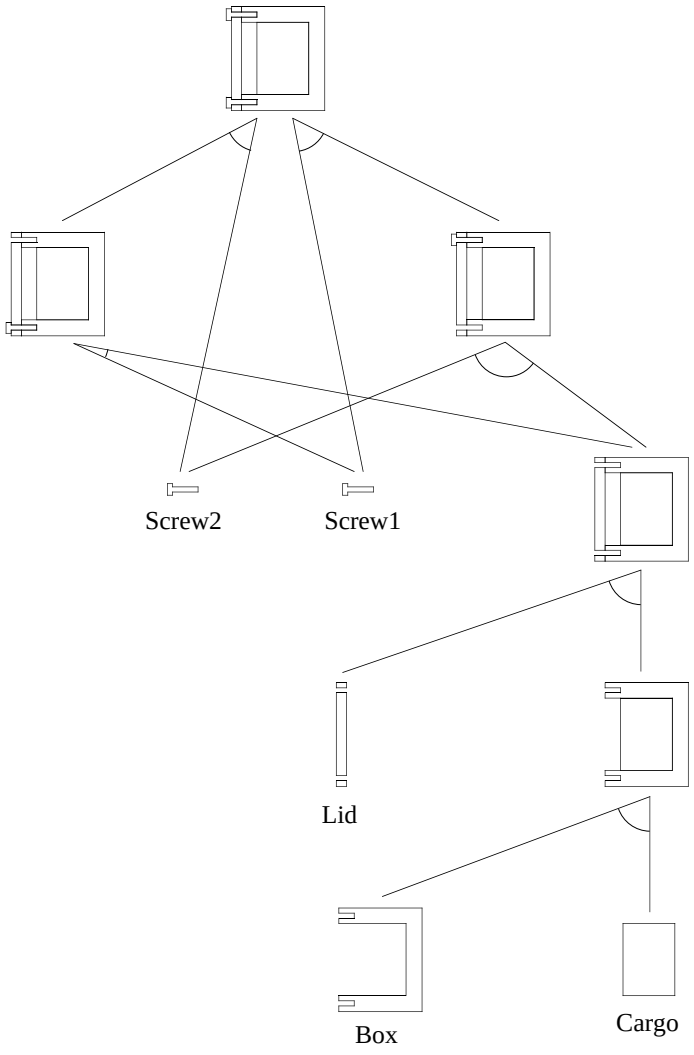

Figure 2.13: A binary AND/OR graph for the crate

Homem de Mello [34] describes several implicit representations and shows how they can be generated from each other and from state graphs and AND/OR graphs. The rules considered there are of three types:

- rules specifying the states of assembly that can follow other states of assembly.
- rules specifying the temporal relationship between establishment of one liaison and states of the assembly, and
- rules specifying the temporal relationship between establishment of one liaison and the establishment of other liaisons.

No methods are given in [34] to generate an implicit representation of a set of assembly sequences without first constructing an explicit representation. However, generating sequence rules by transformation from an explicit representation results in more complicated and more numerous rules than are required. Standard logical simplifications can be applied to reduce the complexity of a rule set, but no systematic methods have been applied in assembly sequencing applications.

A preferable method would be to derive the sequence rules directly from the geometric and other constraints on assembly sequences. Global precedence expressions, described in Chapter 6, are a simple version of sequence rules derived directly from geometric tests. One AND/OR graph representing a set of linear assembly sequences has 1509 nodes and 6190 edges; 34 precedence expressions represent the same set of sequences. However, the set of rules is not always so small, and the entire AND/OR graph must be generated to ensure that the precedence expressions cover every state that might arise in an assembly sequence. This is discussed more in section 6.4.

The non-directional blocking graph of chapters 4 and 5 is also an implicit representation of sets of assembly sequences. As shown there, an NDBG completely defines the set of all assembly sequences using certain types of part trajectories. Although this set of sequences is often of exponential size, the NDBG is of polynomial size, can be constructed in polynomial time, and allows efficient calculation of a sequence satisfying its constraints.

2.6 Assumptions

In the rest of this thesis, the following assumptions hold unless otherwise stated. All assembly sequences are monotone binary, and every subassembly must be connected. In addition,

the parts are modeled as purely geometric objects: the parts are rigid, they have exact geometry, and their positions have no tolerances. This has the disadvantage that a sequencer cannot accurately reason about springs, snap-fit connections, wires, liquids, or other non-rigid parts, or about inaccuracies in sensing, control, and part models that plague real assembly plans. However, known reasoning techniques for such problems are quite limited, and many such problems can be dealt with for practical purposes using special-purpose routines. Furthermore, in monotone assembly sequences for rigid parts a sequencer can represent any subassembly of the product as just a set of parts [34].

Chapter 3

A Basic Assembly Sequencing Approach

This chapter outlines a basic approach to automatic assembly sequencing, considering only the geometric model of the product. An experimental testbed for assembly sequencing called GRASP[1] was implemented following this approach. GRASP is organized in modules to allow easy replacement of individual modules. The geometric techniques described in this chapter are only a starting point, and later chapters give more sophisticated, efficient, and in cases less general modules that have been substituted for basic modules to achieve higher performance. GRASP is a valuable tool for testing alternative methods of assembly sequencing, allowing the methods of the following chapters to be tested on actual assemblies under realistic assumptions about their interactions with the rest of the assembly sequencer.

The basic approach to generating assembly sequences proceeds as follows. The first step is to compute a *connection graph* for the target assembly, detecting the contacts between parts and making them explicit. Then an AND/OR graph is constructed that represents a set of possible assembly sequences for the product, using geometric calculations to check the feasibility of each assembly operation. The geometric calculations used to verify operations include contact analysis to find feasible directions of translation and rotation for subassemblies, checking for interference between parts while moving along a single trajectory, and general path planning when required. The architecture of GRASP is shown in figure 3.1. This basic framework is derived from the work of Homem de Mello [34], although several

[1] GRASP stands for "Geometric Reasoning Assembly Sequence Planner." It has nothing to do with grasp planning.

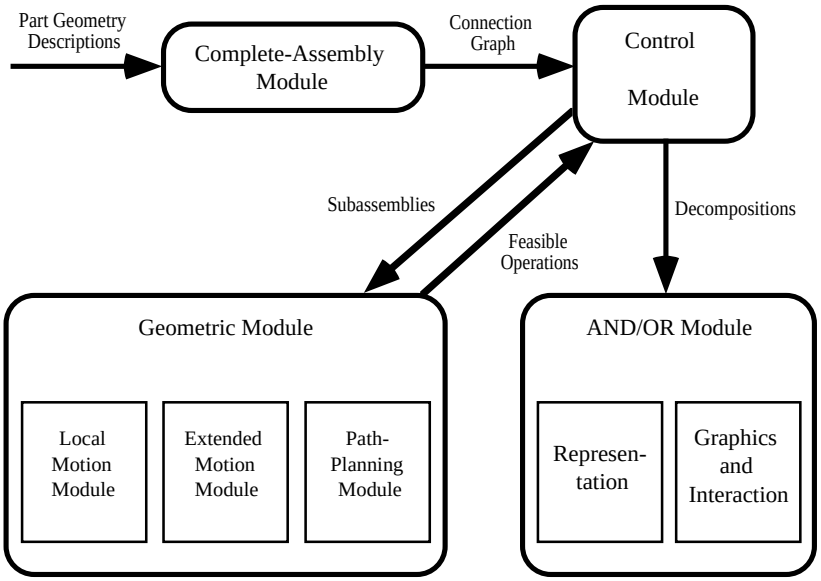

Figure 3.1: The architecture of GRASP

of the geometric techniques are not included there and some important improvements have been made.

3.1 The Assembly Description

The input to assembly sequencing is a description of the product giving the geometry and relative positions of the parts in the product. In addition, the product description includes a *connection graph*, which contains information about the contacts between parts.

3.1.1 Local Motion

The connection graph facilitates computation about small motions of the parts of an assembly. A *local motion* is an arbitrarily small rigid motion of a part, or equivalently, a direction of rigid motion of the part. The local motions of a part at a given position in space form a six-dimensional vector space [13]. For instance, a local motion ΔX can be described as a 6-vector with three degrees of translation and three of rotation:

$$\Delta X = (\dot{x}, \dot{y}, \dot{z}, \dot{\alpha}, \dot{\beta}, \dot{\gamma})$$

where $\dot{\alpha}$, $\dot{\beta}$, and $\dot{\gamma}$ are the rotational components of ΔX around the x, y, and z axes, respectively.

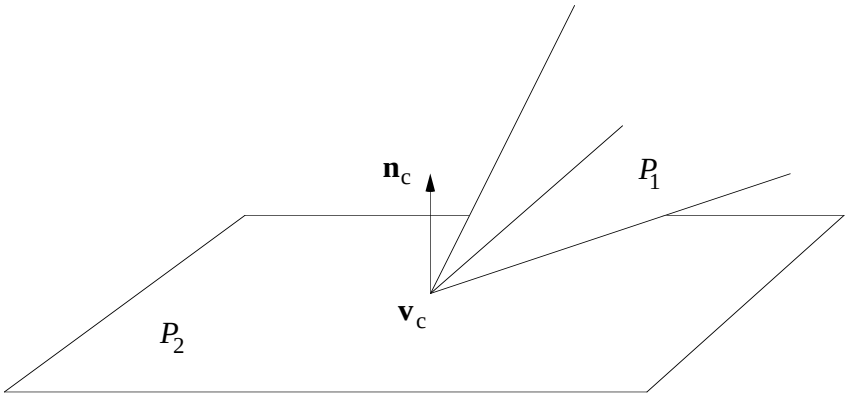

Figure 3.2: A point–plane contact between two polyhedra

The *local freedom* of a part P_1 with respect to a part P_2 is the set of local motions ΔX such that part P_1 can undergo a finite motion in the direction of ΔX without interfering with P_2. The *contact* between two noninterfering parts is the intersection of their boundaries[2]. It is clear that if the contact between P_1 and P_2 is null, then the local freedom of P_1 with respect to P_2 is the full space of local motions.

3.1.2 Contacts

A contact in the connection graph is represented as a finite conjunctive set C of point–plane contact *constraints*. A typical point–plane constraint between two polyhedra is shown in figure 3.2. The constraint c is defined by a vertex of contact v_c and the outward normal of the face n_c. A translation d of v_c will cause P_1 to penetrate P_2 at v_c exactly when $n_c^T d < 0$. The local motion ΔX causes a vertex v_c of P_1 to undergo a translation $d_c = J_c \Delta X$, where J_c is the constant 3×6 Jacobian matrix that relates the differential motion of P_1 to the motion of v_c. Thus ΔX causes P_1 to penetrate P_2 at v_c exactly when $n_c J_c \Delta X < 0$.

The representation C of a contact between parts P_1 and P_2 is interpreted as follows. For each constraint $c \in C$, a local motion ΔX can relate to c in three ways, depending on the relative motion at the contacting point under the motion:

- Motion ΔX *violates* c if and only if $n_c^T J_c \Delta X < 0$. In other words, ΔX violates c exactly when part P_1 undergoing motion ΔX penetrates part P_2 at contact point v_c.

[2]Because I do not consider tolerances on the geometry of the parts, two parts are either in contact or not in contact at any point. When tolerances are considered, the notions of contact and of local motion become more complicated.

- Motion ΔX *breaks* c when $n_c^T J_c \Delta X > 0$. In this case, ΔX causes contact point v_c on P_1 to move away from P_2.
- Motion ΔX *slides on* c when $n_c^T J_c \Delta X = 0$, i.e. when ΔX causes contact point v_c on P_1 to move in a local tangent to part P_2.

When ΔX breaks or slides on constraint c, we say that ΔX *obeys* c. The set of local motions that obey c form a closed half-space bounded by a hyperplane through the origin.

A local motion ΔX obeys the contact C if and only if it obeys all constraints in C. C *describes* the contact between P_1 and P_2 if and only if the set of motions that obey C is equal to the local freedom of P_1 with respect to P_2. In this case the local freedom of P_1 is given by the intersection of the closed half-spaces defined by the constraints $c \in C$.

Most of the typical contacts between parts in industrial assemblies can be described as finite sets of point–plane constraints. I will first consider contacts between polyhedral parts, and then non-polyhedral parts. Note that the point–plane constraints representing a contact need not correspond to actual contact points on the parts; the only requirement is that the local motions that obey the contact be equal to the local freedom between the parts.

3.1.3 Representing Polyhedral Contacts

The following types of contact between polyhedra can be described as sets of point–plane constraints:

plane–point The contact c between a planar face of P_1 with outward normal n_c and vertex v_c of P_2 is given by a constraint at v_c between a point of P_1 and plane of P_2 with outward normal $-n_c$.

face–face A contact between two polygonal planar faces is described by a set of point–plane constraints at the vertices of the convex hull of their contacting surface area. Figure 3.3a shows this case.

nonaligned convex edges Two convex edges touching at a point p are described by a point–plane constraint between p and the plane containing the two edges (figure 3.3b).

edge–face A contact between a convex edge e and a planar face f is described by two point–plane constraints, one at each end of the intersection segment of e and f (figure 3.3c).

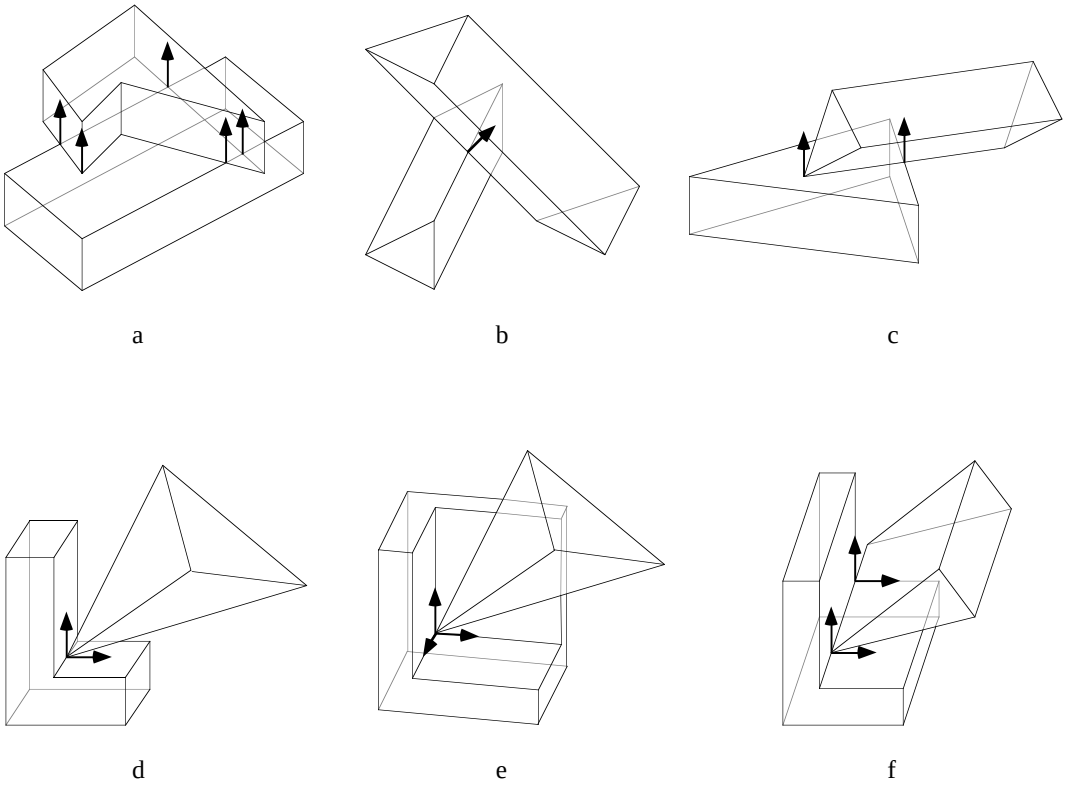

Figure 3.3: Contacts between polyhedra expressed as point-plane contacts

convex vertex–concave edge or vertex If a convex vertex v is in contact with a concave edge or vertex, the constraint on local motion is equivalent to a set of point constraints between v and each of the faces meeting at the edge or vertex (figures 3.3d and 3.3e).

convex edge–concave edge In a similar way, two edge–face contacts suffice to describe the constraint arising from a convex edge contacting a concave edge (figure 3.3f).

A contact between two polyhedra that includes several of the above can be described as a set of constraints C, where C is the union of sets C_i each representing one of the above simple contacts.

The remaining possible contacts between polyhedra are convex vertex–convex vertex, vertex–convex-edge, and aligned-convex-edges contacts (figure 3.4); these contacts cannot be represented using the above scheme. However, such contacts are quite unstable and rarely appear in real assemblies. All but aligned-convex-edges contacts can be treated using finite disjunctions of point–plane constraints [32]. For instance, in figure 3.4a, the motion of

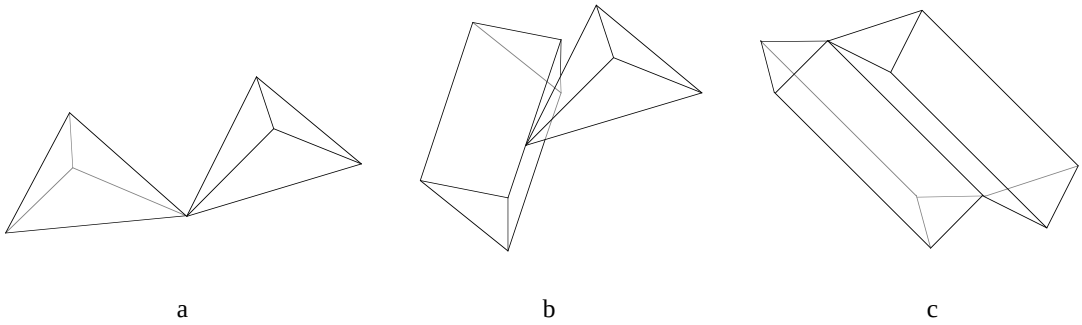

Figure 3.4: Polyhedral contacts not considered here

the contact vertex v_c on P_1 must obey at least one of the point–plane constraints between v_c and the planes of P_2 that meet at v_c.

3.1.4 Representing Nonpolyhedral Contacts

In addition to the above contacts between polyhedra, several common contact types in non-polyhedral assemblies can be expressed in terms of point–plane constraints:

cylinder–face A cylinder contacting a plane in a line segment is equivalent for local motion purposes to an edge–plane contact along the contact line segment (figure 3.5a). Note that although a rolling contact is very different from an edge–plane contact for extended motions, they allow the same local motions.

cylinder–cylinder A round peg in a round hole has the same local freedom as a round peg in a triangular hole. Thus a cylinder–cylinder contact can be described as three cylinder–plane contacts, i.e. six point–plane contacts (figure 3.5b).

threaded cylinders A contact between two threaded cylinders can be expressed as shown in figure 3.5c. A cylinder–cylinder contact is combined with two point–plane contacts that together express the twisting constraint of the threads at a single point. The normals of the two thread contact constraints are opposing and have angle $\alpha = \arctan\frac{1}{p}$ with the axis of the cylinder, where p is the pitch of the threads. In the procedure to calculate local freedom described below, this representation of threaded contacts is never needed because motion is so strongly constrained.

Although products often have complicated, curved surface shapes, the great majority of contacts between parts fall into the cylindrical, planar, and threaded types above. The main

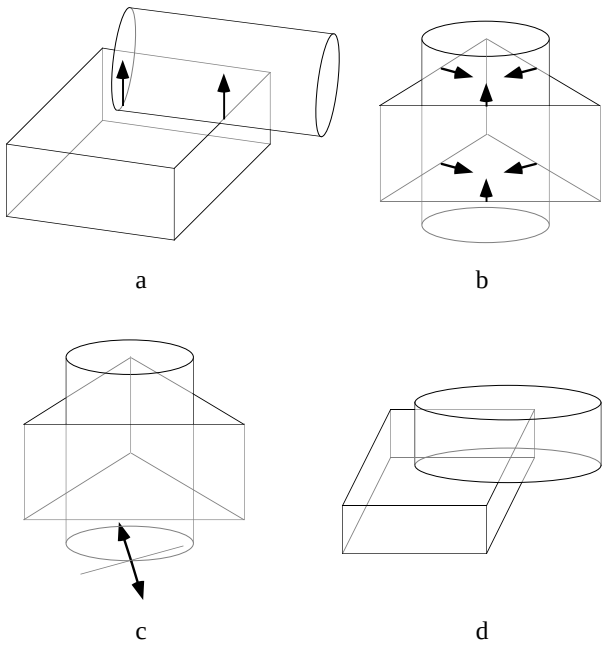

Figure 3.5: Typical non-polyhedral assembly contacts

exception is when the convex hull of the contact area of a face–face contact is not polygonal (see figure 3.5d). Such a contact is equivalent to an infinite number of point–plane contacts around the convex hull. A polygonal approximation of the convex hull allows such a contact to be described with some loss of accuracy.

Non-contacting surfaces of parts are often curved to satisfy requirements such as strength, aerodynamics, and aesthetics. Thus reasoning about curved surfaces is more important when extended motions (instead of local motions) are considered, since then these surfaces may interfere with each other. In such cases approximate methods are often better suited, such as in [33] and in section 3.4.

Finally, figure 3.6 shows another type of contact that cannot be represented. Although the contact seemingly could be represented by two point–plane constraints, the resulting local freedom would allow vertical translation. Any such finite translation will cause a collision. This is an additional reason why local freedom is only a necessary condition on the movability of a part (section 3.2.4).

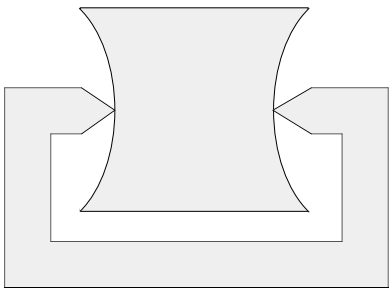

Figure 3.6: A contact that cannot be represented

3.1.5 The Connection Graph

The connection graph of a product is an undirected graph in which a node corresponds to a part of the product and a link connects two parts that are in contact. Each link is associated with a finite set of point–plane constraints that describes the contact between the parts. Formally,

Definition 3.1 *A connection graph is a 4-tuple* $\langle P, C, L, f \rangle$ *where*

- P *is a set of parts.*
- $L \subseteq P \times P$ *is a set of links between parts. There is one link between each pair of parts that are in contact, and no link between uncontacting parts.*
- C *is a set of point–plane contact constraints between parts.*
- $f : C \rightarrow L$ *is a surjective map of constraints onto links. Thus every link between two parts is associated with a set of point–plane constraints. No constraint belongs to more than one link.*

The links of the connection graph are also referred to as *connections* or *liaisons*. Each $c \in C$ represents the constraint on motion of one part with respect to another; thus a contact between parts at a vertex v gives rise to two complementary contact constraints $c_i, c_j \in C$ with opposing constraint vectors. Let $C(A)$ denote the connection graph of an assembly A.

The connection graph is similar to Bourjault's *liaison diagram* and Homem de Mello's *relational model*, both of which are undirected graphs with a node for each part. A liaison diagram is a loosely defined graph that does not contain information about the constraints

on motion of the parts, and in which liaisons sometimes connect parts that are not in contact [14]. The relational model of an assembly describes the connections between parts on a more symbolic level—in terms of mechanical attachments, fasteners, and so on—than does a connection graph [34].

In an integrated environment such as a concurrent design system, the input to an assembly planner will probably include a connection graph, for several reasons. Although computer-aided design tools currently represent best the geometric aspects of a design, some commercial CAD systems support feature-based models, from which the connections can be easily established. In the future, product models will include such extra-geometric information as degrees of freedom, design decisions taken, and the functional requirements of a design [40]. The contacts between parts must be known to perform many kinds of analysis in design, such as tolerancing and stress and kinematic analysis [8, 38]. Furthermore, contact information can often be ambiguous, due to tolerances or small distances between parts, requiring explicit human clarification. However, in many instances the sequencing system will need to supplement the input model of the assembly for planning purposes.

Since no tolerances are assumed on the parts and to make the assembly sequencing testbed as autonomous as possible, GRASP constructs its connection graph of the product automatically from the boundary representations of the individual parts. For techniques to accomplish this and a detailed description of the input format for GRASP, see Appendix A.

3.2 Generating Assembly Sequences

From the connection graph and solid models of the constituent parts of the assembly, an AND/OR graph representing all feasible monotone assembly sequences can be generated.

3.2.1 Building the AND/OR Graph

Figure 3.7 gives the main algorithm to generate the AND/OR graph in disassembly planning. *EXPAND* is called first with the goal assembly, and the algorithm builds the AND/OR graph from the top down. The goal assembly is decomposed into two subassemblies according to geometric constraints, then those subassemblies are recursively decomposed, and so on. A single AND-tree can be found by choosing a single operation at each node, instead of expanding them all.

The entire AND/OR graph of assembly sequences for a product can be very large. A

```
Procedure EXPAND(A)
    unless expanded(A)
        expanded(A) ← true;
        P ← DECOMPOSE(A);
        for each partitioning (S1, S2) ∈ P
            ADD-DECOMPOSITION(A, S1, S2);
            EXPAND(S1);
            EXPAND(S2);
        end; {for}
    end; {unless}
end; {procedure}
```

Figure 3.7: Main algorithm of GRASP

single AND-tree could be built instead of the whole graph. However, without considering alternatives it is difficult to guarantee any sort of optimal sequence, and a single AND-tree leaves very little flexibility to satisfy further constraints in the planning process. During sequencing all the criteria for choosing the best sequence might not be available, and so a number of solutions should be generated to allow later refinement. Another option is to search the implicit AND/OR graph for an optimal AND-tree, using an algorithm such as AO* [49]. To use AO*, a pessimistic heuristic must be chosen, and the heuristic will greatly influence the search results; in addition, AO* also has the inflexibility of generating only one AND-tree. The geometric techniques presented in this thesis are relevant for other modes of searching the assembly problem space with only minor modifications.

3.2.2 Procedure *DECOMPOSE*

A *partitioning* of an assembly A is a pair of non-empty subassemblies (S_1, S_2) such that S_1 and S_2 partition the parts of A. The procedure *DECOMPOSE* executes the physical reasoning necessary to determine a set of *feasible* partitionings of A: partitionings (S_1, S_2) such that S_1 and S_2 can be brought together in one operation to create A. There are several versions of *DECOMPOSE*, depending on the type of assembly sequences being generated and the geometric reasoning to be performed at each step. A basic version is shown in figure 3.8 for reference.

In addition to checking for geometric feasibility of the operation, the basic version ensures that both subassemblies are connected; such a partitioning is called *connected*. As

```
Procedure DECOMPOSE(A)
    D ← CONN-PARTITIONINGS(A);
    feasible-decompositions ← ∅;
    for each decomposition (S1, S2) ∈ D
        if SEPARABLE(S1, S2)
            push((S1, S2), feasible-decompositions);
    end; {for}
    return(feasible-decompositions);
end; {procedure}
```

Figure 3.8: Procedure *DECOMPOSE*, following [34]

noted in section 2.4.4, connected subassemblies are usually easier to handle. In addition, the connectedness constraint reduces the combinatorics of the planning process.

DECOMPOSE follows a generate-and-test approach like *GET-FEASIBLE-DECOMPOSITIONS* in [34]. It calls *CONN-PARTITIONINGS* to generate all connected partitionings of the connection graph $C(A)$ of A, then calls procedure *SEPARABLE* to test each disassembly operation—separating the subassemblies corresponding to the two partitions—for geometric feasibility (see section 3.2.4).

3.2.3 Generating Partitionings

A straightforward way to generate all the connected partitionings of a graph G is to find all the *cut-sets* of G. Any set of edges E whose removal partitions G into exactly two connected components is a cut-set, provided no proper subset of E also disconnects G. There is a one-to-one correspondence between a graph's cut-sets and its connected partitionings. The cut-sets of a graph can be generated from a system of fundamental cut-sets defined by a spanning tree of the graph as described in [26]. However, the straightforward implementation of this method runs in time $\Omega(2^m)$, where m is the number of links in $C(A)$.

A simpler algorithm to generate connected partitionings, proposed by Homem de Mello [34], enumerates connected components S of G and checks whether their complements $A \setminus S$ are connected; if so, $(S, A \setminus S)$ is a connected partitioning. The complexity of this algorithm is $O(2^n)$ in the worst case, where n is the number of parts of A.

Figure 3.9 shows *CONN-PARTITIONINGS*, which finds all the connected partitionings of the connection graph $C(A)$ in time polynomial in the number of such partitionings. Since

```
Procedure CONN-PARTITIONINGS(A)
    D ← ∅;
    for an arbitrary P_0 ∈ A
        S ← {P_0};
    for each P_i ∈ A \ {P_0}
        CONN-SUPERSETS({P_i}, S);
        S ← S ∪ {P_i};
    end; {for}
    return(D);
end; {procedure}

Procedure CONN-SUPERSETS(S_1, S_2)
    CC ← CONNECTED-COMPONENTS(A \ S_1, C(A));
    if |CC| > 1
        if for some C_i ∈ CC,  C_i ⊇ S_2
            S_1 ← S_1 ∪ ⋃_{j≠i} C_j;
        else return;
    end; {if}
    push((S_1, A \ S_1), D);
    for each neighbor P_i of S_1 in C(A) such that P_i ∉ S_2
        CONN-SUPERSETS(S_1 ∪ {P_i}, S_2);
        S_2 ← S_2 ∪ {P_i};
    end; {for}
end; {procedure}
```

Figure 3.9: An algorithm to generate all partitionings of a graph into two connected components

two partitionings (S_1, S_2) and (S_2, S_1) are equivalent, *CONN-PARTITIONINGS* chooses an arbitrary part P_0 and finds only those partitionings with P_0 in the second partition. Each remaining part P_i divides the partitionings of G into those with P_i in the first partition and those with P_i in the second. Part P_i is called a *pivot*. Let $D(S_1, S_2)$ denote the connected partitionings (S_1', S_2') of $C(A)$ such that $S_1' \supseteq S_1$ and $S_2' \supseteq S_2$. The recursive procedure *CONN-SUPERSETS*(S_1, S_2) enumerates $D(S_1, S_2)$.

At each call of *CONN-SUPERSETS*(S_1, S_2), the sets S_1 and S_2 must be disjoint and both non-empty, and furthermore S_1 must be connected. The procedure first computes the connected components CC of $A \setminus S_1$. If $A \setminus S_1$ is connected, then $(S_1, A \setminus S_1)$ is a connected partitioning. If $|CC| > 1$, then the only supersets S_1' of S_1 that will have a connected

complement $A \setminus S_1'$ will include all the CC but one. There are two cases:

- If S_2 intersects more than one of the CC, then all supersets S_2' of S_2 not intersecting S_1 are unconnected. In this case $D(S_1, S_2)$ is empty, so the procedure returns.
- If S_2 is contained in one connected component C_i, then that C_i cannot be added to S_1. Clearly then, all components $C_j, j \neq i$, must be added to S_1, and then $(S_1, A \setminus S_1)$ is a connected partitioning.

After one partitioning (S_1, S_2) is found, then each part P_i connected to S_1 and not in S_2 becomes a new pivot. The pivot P_i divides $D(S_1, S_2)$ into two sets: partitionings containing P_i in S_1' and those with P_i in S_2'. Recursive calls to *CONN-SUPERSETS* enumerate those partitionings. If all neighbors P_i are in S_2, then no superset of S_1 will be connected.

Let A have n parts and m connections between them, and let s be the number of connected partitionings of $C(A)$. A single call of *CONN-SUPERSETS* can be performed in $O(m)$ time, when the recursive calls are not included. There are $n-2$ top-level calls to *CONN-SUPERSETS*; each connected partitioning is followed by at most $n-3$ recursive calls, so it is called $O(ns)$ times. Hence *CONN-PARTITIONINGS* runs in time $O(nms)$.

Procedure *DECOMPOSE* calls procedure *SEPARABLE* to test the assembly operation corresponding to each connected partitioning of A for geometric feasibility.

3.2.4 Procedure *SEPARABLE*

The procedure *SEPARABLE* encompasses the geometric reasoning module of GRASP. The assembly operation mating subassemblies S_1 and S_2 is geometrically feasible if there exists a collision-free path to move S_1 from a position separated from S_2 (as defined in Chapter 2) into its final position relative to S_2, or equivalently, if there exists a path to separate S_1 from S_2. Such a path could be computed by calling a general purpose path planner [42]. However, calling a path planner is in general very expensive.

Because of the cost of calling a path planner, procedure *SEPARABLE* employs much simpler techniques to determine the feasibility of most proposed assembly tasks. Both necessary and sufficient conditions for the feasibility of an assembly task are checked before resorting to path planning:

- Local motion analysis determines whether S_1 is fully constrained by its contacts with S_2. If S_1 has no local motion, it obviously cannot be removed in this step. Local motion analysis prunes many infeasible assembly operations quickly.

```
Procedure SEPARABLE(S1, S2)
    freedom-cone ← LOCAL-FREEDOM(S1, S2);
    motions ← USEFUL-DIRECTIONS(freedom-cone);
    if empty(motions)
        return(false);
    else
        for each direction d ∈ motions
            if VALID-MOTION(S1, S2, d)
                return(true);
        end; {for}
        return(PATH-PLAN(S1, S2));
    end; {else}
end; {procedure}
```

Figure 3.10: Procedure *SEPARABLE*

- If S_1 has a valid local motion, then some simple motion might separate the two subassemblies. For instance, if any single collision-free translation can move one subassembly to infinity, then the operation is feasible. Checking simple extended motions proves many assembly operations feasible while requiring little computation.

In industrial assemblies, these special cases correspond to the vast majority of assembly operations, so the sequencer can achieve much greater efficiency by explicitly checking for them. Where both methods fail, more expensive plan planning methods can be brought to bear.

Figure 3.10 shows procedure *SEPARABLE* as implemented in GRASP. The next sections describe the geometric computation that accomplishes each of the substeps of *SEPARABLE*.

3.3 Local Motion

In an assembly A, the local freedom of a subassembly S of A is the set of directions in which S can move an infinitesimal distance from its current position, given the geometry of $A \setminus S$ considered as a solid. If S can be removed, it can be moved a very small distance; hence, if S has no local motion, it cannot be removed. If S has at least one valid local motion, it is called *locally free* in A. For simplicity I will assume that the reference frames for all parts coincide, so that local motions for all parts are in the same coordinate system.

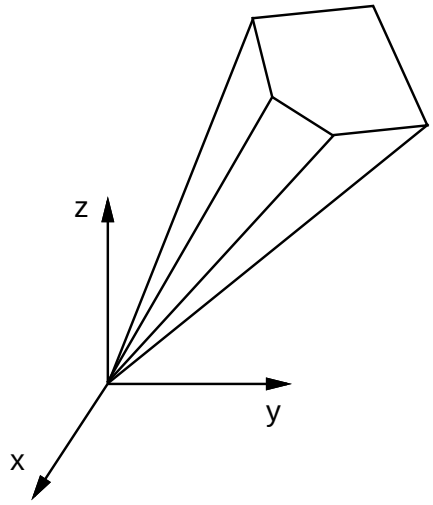

Figure 3.11: A 3D local translational freedom cone

3.3.1 Local Freedom

As described in section 3.1.2, the motions allowed by a set of point-plane contact constraints are the intersection of the motions allowed by each constraint individually. Therefore F, the local freedom of S_1 allowed by a set C of k point–plane constraints, is given by

$$F = \bigcap_{c \in C} \{\Delta X \mid n_c^T J_c \Delta X \geq 0\} \tag{3.1}$$

The set of constraints in (3.1) can be rewritten

$$C_a \Delta X \geq 0 \tag{3.2}$$

where $C_a : k \times 6$ is the matrix whose ith row is $n_{c_i}^T J_{c_i}$. The set of solutions to the inequalities (3.2) is a polyhedral convex cone F whose shape and dimension vary with $r = rank(C_a)$ [30]. The tip of F is a linear subspace of dimension $d = 6 - rank(C_a)$ and the edges of the cone are of dimension $d + 1$. Motions in the tip of F slide on all contacts between S_1 and S_2, while motions in the interior of F break all contacts.

When local motions are restricted to translation, the set of local motions lie in a three dimensional space, and a typical polyhedral convex cone is shown in figure 3.11. For an example in two dimensions, consider the possible local translations for part `A` in figure 3.12. Each edge contact of `A` with part `B` or `C` restricts the local translations of `A` to a half-plane, where the inward normal to the half plane is the inward normal of the contacting edge of `A`.

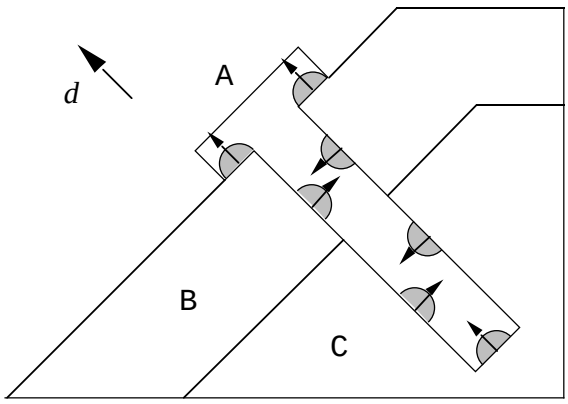

Figure 3.12: Local freedom computation

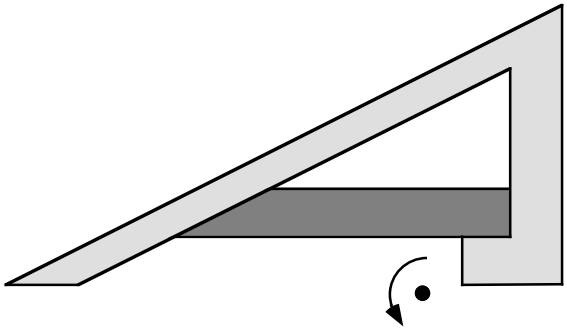

Figure 3.13: A 2D rotation to remove a part that cannot translate

The local translational freedom of A is the intersection of the half-planes of motion, a single ray d. This d is the only direction along which A can translate.

Even if a part is fully constrained for translations, it still might be removable using a combination rotation and translation. Figure 3.13 shows a part in 2D that is fully held in translation but can be freed by a rotation around the point shown, while figure 3.14 gives an example in three dimensions.

3.3.2 Useful Motions

A subassembly S of A is fully constrained when the local freedom cone F of S contains only the origin. However, some local motions do not contribute to removing S from $A \setminus S$:

- A pure rotation around an axis of symmetry for S simply maps S into itself, thus coming no closer to being removed.

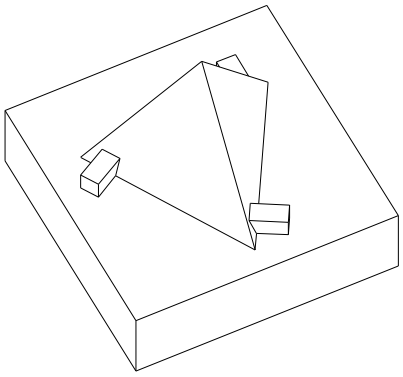

Figure 3.14: A part that can be freed by a twist in 3D

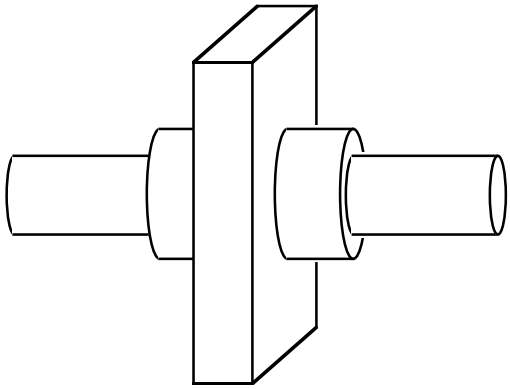

Figure 3.15: The plate can only rotate around an axis of symmetry of the remaining parts

- Similarly, a rotation around an axis of symmetry for $A \setminus S$ does not contribute to disassembly.
- Let C be the contact between S and $A \setminus S$. A rotation around an axis of symmetry for C will not result in any new freedom, so it cannot contribute to disassembly.

For instance, figure 3.15 shows a plate on a shaft with two retainers. The plate's local freedom cone includes only rotations around the axis of symmetry of the rest of the assembly, so the plate is locally free yet impossible to remove. Thus for S to be removable F must include some motions that are not rotations around axes of symmetry for either S, $A \setminus S$, or their contact C. If a line l is an axis of symmetry for all parts $P \in S$, then it is a axis of symmetry for S. However, the converse does not hold; l can be an axis of symmetry for S without being an axis of symmetry for any part in S.

3.3.3 Computing Local Freedom

Given a set of point–plane constraints C between S and $A \setminus S$, finding the 6D local freedom cone F for S is complicated and time-consuming. For instance, the procedure described in [32] takes time $O(k^3)$ for k contacts in some cases. Therefore procedure *LOCAL-FREEDOM* performs several less involved checks to handle the most frequent and simpler cases before resorting to the general case:

- Certain combinations of contacts can be found that constrain S completely; for instance, two cylindrical or threaded contacts with non-parallel axes allow no local motion.

- In many cases a contact allows only a finite set D of removal directions. For instance, a threaded contact allows only two motions, one spiraling in each direction of the axis; two parallel cylindrical contacts allow only translation along their axes. In these cases, each motion in D can be checked for compatibility with all of the other contacts. If every motion in D is incompatible with at least one contact in C, then S cannot move. Otherwise the local freedom of S is the subset of D compatible with the rest of the contacts.

- If neither of the first two cases hold, then the local translational freedom T of S is computed. If a cylindrical contact c is in C, then T includes only translations parallel to the axis of c that are compatible with the other contacts in C. If C contains only planar and cylinder-plane contacts, T is computed by intersecting the half-spaces of freedom given by the planar contacts. With planar contacts T takes the form of a polyhedral convex cone as in figure 3.11. For k contacts, T can be computed in time $\Theta(k \log k)$ [55]; Homem de Mello and Sanderson [36] implemented another method. If T is non-empty then S is locally free.

- If the previous checks fail, the 6D local freedom F of S is computed using a method similar to that in [32]. Figure 3.16 shows two local motions found for the assembly of figure 3.14. The function *USEFUL-DIRECTIONS* then determines whether F contains at least one infinitesimal motion that displaces the subassembly with respect to the rest of the assembly. The symmetries for a subassembly S are the intersection of the symmetries of its parts $P \in S$, found by *COMPLETE-ASSEMBLY* (see Appendix A).

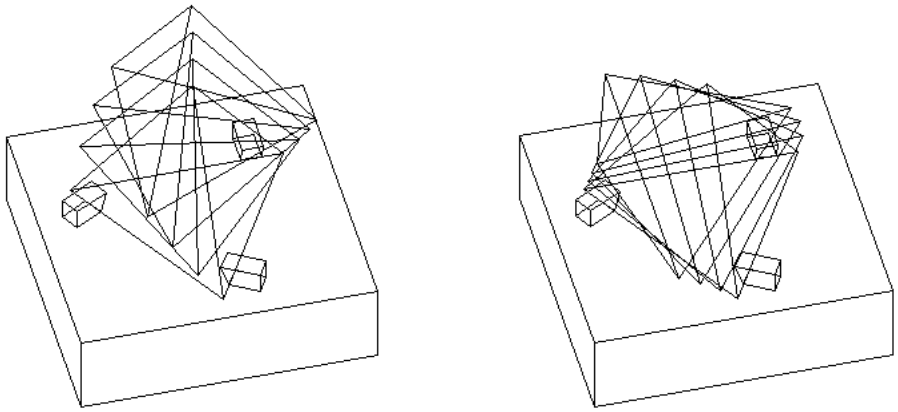

Figure 3.16: Two local motions for the part in figure 3.14

If all of the spanning vectors for S's 6D local freedom cone represent rotations around symmetries of S or of $A \setminus S$, then S cannot be removed.

In normal operation, the special cases encompass so many assembly operations that the general local freedom computation is usually turned off during assembly sequencing using GRASP. When this is done, the 6D local freedom calculation routine always returns a null cone, saving the effort of calculating the 6D local freedom cone for every constrained subassembly. As a result, a few subassemblies are incorrectly found to be constrained in this mode.

3.4 Extended Motion

If a subassembly is locally free, then it might be removable along a simple trajectory, such as a single translation to infinity or the helical motion followed by a screw in a threaded hole. When such a motion is found, the expense of calling a general motion planner can be avoided.

3.4.1 Global Freedom

The globally-valid translations to remove a subassembly S_1 from an assembly A constitute the *global translational freedom* of S_1 with respect to its complement $S_2 = A \setminus S_1$. The global translational freedom G is the set of directions in which S_1 can translate indefinitely without intersecting S_2. Compared to the local freedom cone, which is always convex, the

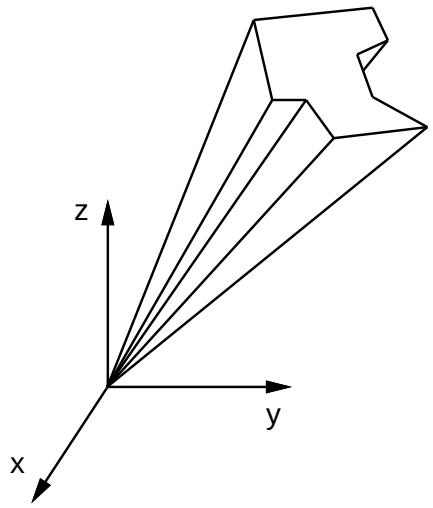

Figure 3.17: A nonconvex cone of removal translations

global freedom cone can be nonconvex (figure 3.17). If all parts in A are polyhedral, the resulting cone will be polyhedral also. Each face of G that is not in common with the local translational freedom cone will arise either from a vertex of S_1 and an edge of S_2, or from an edge of S_1 and a vertex of S_2.

Krishnan and Sanderson [41] find the extended translations possible for one part with respect to another by mapping the set of all unit translations onto a two-dimensional grid, and marking grid elements that correspond to collisions between two polyhedra. Any unmarked elements then represent valid removal translations. However, this method is only accurate to the size of the grid, and cannot be used to find translations involving contacts between the two parts.

A method to calculate G efficiently and accurately is given in Chapter 5. However, for most assembly operations simpler global motion checking procedures will suffice. A locally free subassembly S can be swept in some of the directions in its local freedom cone, from its current position to infinity. If any direction is free of collisions with the rest of the assembly, it constitutes a valid removal path for S. In experiments this method has proven fast and accurate in the vast majority of assembly operations. The directions to sweep are chosen heuristically by the function *USEFUL-DIRECTIONS* based on the shape of the local freedom cone. For instance:

- When the translational freedom cone is a half-space, the normal of the plane facing into the half-space and four perpendicular directions in the bounding plane of the half

space are chosen as sweeping directions.

- For a cone such as the one in figure 3.11, sweeping is performed along vectors parallel to the edges of the cone.
- When a threaded contact exists, the corresponding twisting motion is extended in each direction to infinity.
- When a 6D local freedom cone has been calculated, the axis of motion and the pitch of the rotation about that axis can be extracted for each of the spanning motions of the cone, and a twisting motion generated to infinity along the axis. A motion with no rotational component is a translation. A pure rotation around an axis of symmetry of either S or $A \setminus S$ is not considered.

For each direction of sweep, the function *VALID-MOTION* checks whether the trajectory constitutes a collision-free path to separate S from $A \setminus S$. If one of the chosen trajectories is free from collision with all other parts present in the assembly, it constitutes a valid insertion path for S.

3.4.2 Sweeping

To sweep a subassembly in a direction d, *VALID-MOTION* sweeps the individual parts along d. The problem of collision detection among moving objects has been well studied. Canny [15] gives an algorithm for detecting collisions between polyhedra translating and rotating in three dimensions that runs in $O(n^2 \log n)$ where n is the total complexity of the objects. Although Canny's technique could be used in assembly sequencing, simpler and less powerful methods have been implemented in GRASP.

To sweep a part in translation, the faces of the translating part are compared pairwise with the faces of each possible interfering part to check for collision. If the two faces intersect when projected into the plane perpendicular to the vector of translation, and the face being swept is behind the interfering face at one or more of the points of intersection, then a collision exists and the motion is infeasible.

Rather than sweep a part along a twisting motion, GRASP computes a rotational closure of the object about the axis of the path, and sweeps the resulting cylindrical shape in translation along the axis. Thus for each vertex v of the moving part with distance d to the axis of motion, and for each edge e of a stationary part, if the segment of e above v

with respect to the axis comes within d of the axis, a collision is found. Edges of stationary parts that represent circular curves are replaced by their circular arcs for this purpose, since otherwise Vantage's polyhedral approximation of cylinders would cause incorrect collisions to be detected. For trajectories with relatively small absolute pitch, such as those resulting from threaded contacts, this sweeping computation approximates quite well the actual swept volume of the original twisting path. For larger pitches, however, the calculation is very conservative.

To minimize sweeping computations, GRASP saves the result of each sweep for later retrieval. *PREVIOUS-SWEEPS*[P_1, P_2] is a two-dimensional array of lists of pairs (d, *collides*). When a part P_1 needs to be swept against part P_2 in direction d_1, *PREVIOUS-SWEEPS*[P_1, P_2] is searched for a pair whose first element is d_1. If one exists, *collides* is T if P_1 hits P_2 in direction d_1, and F if not. If no pair (d_1, *collides*) is found, P_1 is swept against P_2 as above, and the result is stored in the table. This technique is called *sweep caching*. Since the same motion of the same part can be attempted in many different subassemblies during the planning process, sweep caching accelerates planning considerably.

3.5 Path Planning

When a subassembly is locally free, but no simple trajectory can be found to extricate it from the assembly, a sequence of translations or a curved path might exist to remove it.

GRASP has a well-defined interface to a path planner, called through the procedure *PATH-PLAN*. *PATH-PLAN*(S_1, S_2) returns one of the following four answers:

Movable S_1 can be separated from S_2 by a path that is not specified.

Path S_1 can be separated from S_2 by a given path.

Not Movable S_1 and S_2 cannot be separated in one operation.

Constrained S_1 and S_2 cannot be separated in one operation, and furthermore two sets of *constraining* parts are identified. The constraining parts are subsets $S_1' \subseteq S_1$, $S_2' \subseteq S_2$, such that S_1' cannot be separated from S_2' by any path.

Thus there are two simple answers, Movable and Not Movable. Path is the same as Movable but gives more information, while Constrained gives more information than Not Movable. Both types of extra information can be used in the planning process to reduce the number

of queries needed to the path planner. Specifically, when a path is returned, it will still be valid as long as no parts interfere with it. When constraining sets are returned, any subassemblies that include those sets are not separable. See Chapter 6 for more details.

Both types of additional information should be retrievable from a path planner, with some modification. Most path planners return a single feasible path as part of their answer if one exists. In addition, an automated path planner can conceivably produce a list of parts that together constrain a subassembly S_1 and S_2. The planner must be augmented to find this set, and the technique used will depend heavily on the path planning technique employed. For instance, if the planner builds an adjacency graph based on a cell decomposition of the configuration space of S_1 [42, chapters 5–6], it may return a list of the parts contributing boundaries to the connected component of free space that contains the starting position of S_1. If the path planner is based on a local exploration of the configuration space [7] it may return the set of parts of S_1 and S_2 that collided during the search process; in this case the list of constraining parts may not be complete, however. Although it will depend on the exact planner used, the set of constraining parts can conceivably be found with little additional computation.

Path planning can be turned on and off by the user. When path planning is turned off, the path planner simply indicates that no motion is feasible. When path planning is on, the current implementation asks a human designer to act as the path planning expert.

The path planning human interface works as follows. A drawing of the two subassemblies S_1 and S_2 is presented in a window, along with a question and a set of buttons showing appropriate answers (see figure 3.18). The parts of S_1 are highlighted in green; S_2 is in white. The message asks the engineer to identify those parts of S_1 and S_2 that prevent S_1 from being separated from S_2. The user can then select and unselect parts of A by clicking on their edges with the mouse pointer; each part turns red as it is selected.

The human interface only allows the following three answers:

Movable S_1 can be separated from S_2. It would be difficult and time-consuming for the user to enter a removal path manually, so it has not been implemented. Hence the answer Path as defined above is not available in the human interface.

Not Movable S_1 and S_2 cannot be separated in one operation. This answer can be selected if the engineer does not want to answer the question in more depth.

Constrained The highlighted parts are the constraining sets returned to the sequencer. If

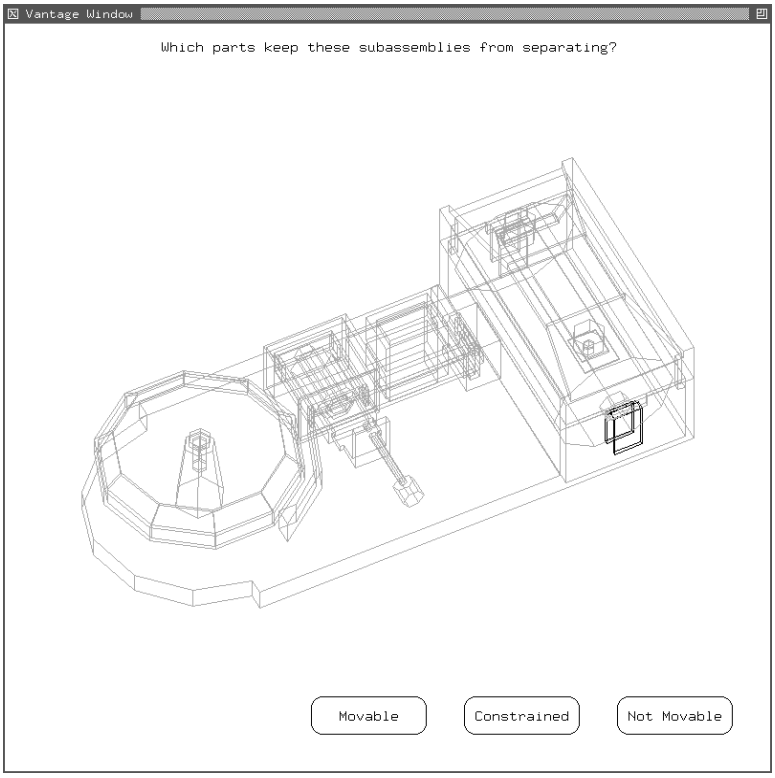

Figure 3.18: GRASP's human path planning interface

S_1' or S_2' are not minimal, sequencing will be correct but slower than otherwise.

The answer returned by the human is the final word on the removability of one subassembly from another.

3.6 Implementation

GRASP is implemented in Allegro Common Lisp under the Xwindows window system. Experimental results are computed on a DEC5000 workstation. A 2D prototype of GRASP was written in the same environment, but it does not include many advanced features of the full system. Important differences will be stated where experimental results from the prototype are given.

Figure 3.19 shows GRASP planning for the assembly of the electric bell (see Appendix B. It has generated the partial AND/OR graph seen in the upper left, and the graphics window shows the current operation it is adding to the graph: placement of the battery into its case.

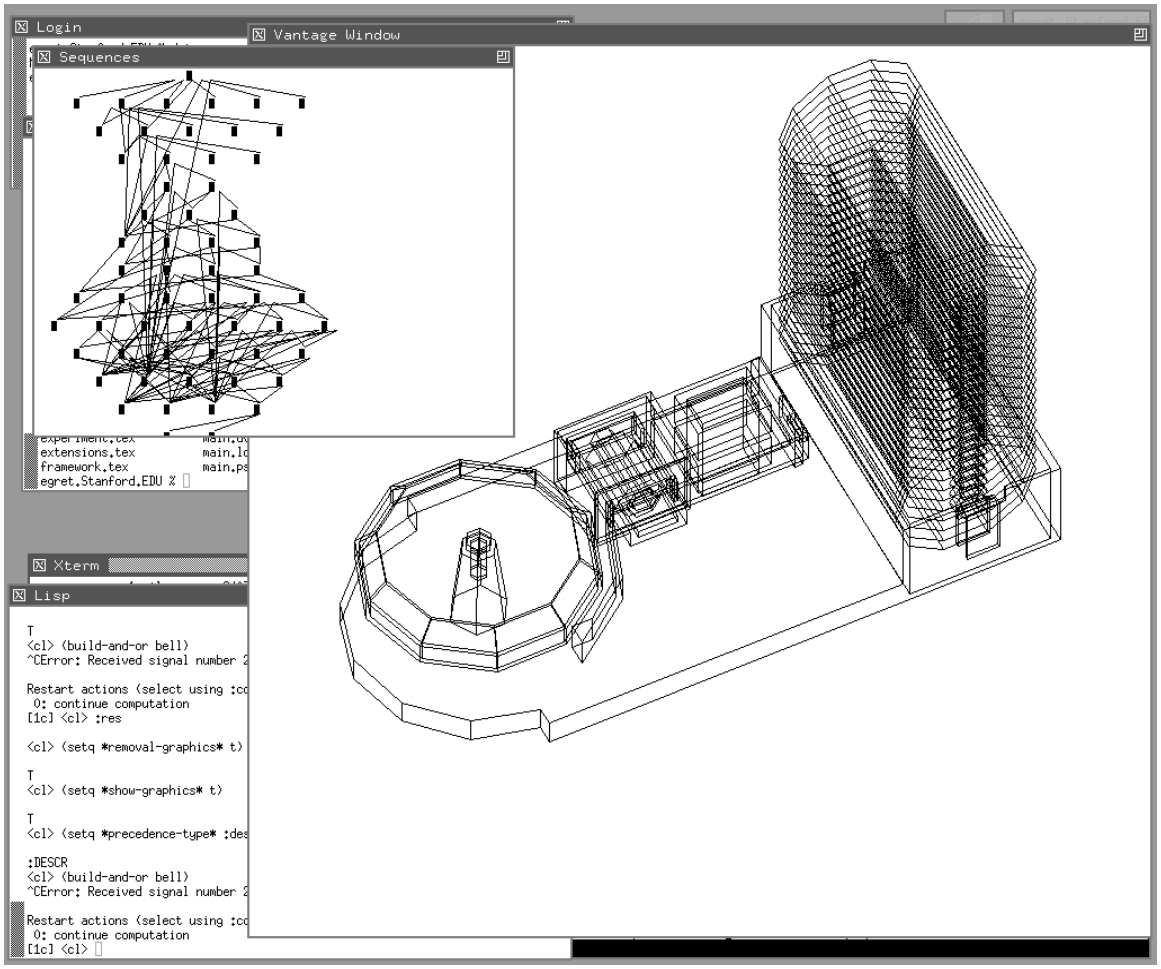

Figure 3.19: GRASP in operation

With motion planning and 6D freedom cones turned off, the AND/OR graph of assembly sequences for the bell has 2,320 nodes and 21,315 edges. Using the basic geometric assembly sequencing methods described in this chapter, GRASP builds it in 54 minutes, requiring 40,754 calls to procedure *SEPARABLE*, including 609 part-part sweeping calculations. A single valid assembly sequence is found in 76 seconds and requires 247 geometric calls.

Clearly, to be used in an interactive environment such as a concurrent design system, or to plan for assemblies with many more parts, these techniques must be improved upon. The following chapters present new, more efficient and more accurate algorithms that improve upon the basic methods given here.

Chapter 4

Partitioning for Local Motions

This chapter improves on one of the basic methods of geometric reasoning for assembly sequencing described in Chapter 3. The first test of an assembly partitioning given there is local freedom. Local freedom is a powerful constraint, but the method described in Chapter 3 follows a generate-and-test approach that might generate a large number of partitionings for only a few locally free ones.

This chapter presents an algorithm called *PARTITION* that efficiently finds locally free subassemblies of an assembly A in both translation and rotation. The algorithm is based on a new representation of the blocking relationships between part in A, called the *non-directional blocking graph* of A. Specifically, let A have n parts and let the connection graph $C(A)$ have m links and a total of k point–plane contact constraints. Then a subassembly S that is locally free in translation can be identified in time $O(mk^2)$, and all s locally free subassemblies can be found in time polynomial in s. Thus when there is an exponential number of locally free subassemblies, the algorithm requires exponential time. When general rigid motions are allowed, a single locally free subassembly can be found in time $O(mk^5)$, and all such subassemblies can be identified with a similar output-sensitive time bound. An extension of either algorithm yields connected locally free partitionings in the same time bounds.

Some assemblies have a large number of locally free subassemblies, few of which satisfy other geometric constraints, such as extended motion freedom. For such assemblies the total computing time to find a removable subassembly might be exponential using the method given here to generate candidate subassemblies. Chapter 5 describes a variation on the algorithm of this chapter to efficiently find subassemblies that are free for extended

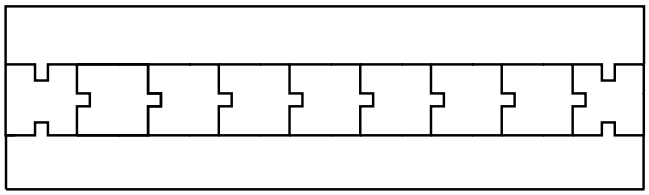

Figure 4.1: An assembly with 2 feasible decompositions

translations. However, in experiments local freedom has proven a powerful constraint, pruning most impossible operations. Experimental results are shown for an implemented hybrid algorithm that combines the low time bound of the translational algorithm with the ability to find most assembly operations involving rotations.

4.1 Generate-and-Test

The basic approach given in figure 3.8 to find a removable subassembly of A follows a generate-and-test scheme. All ways to partition A into two connected subassemblies are generated, and each partitioning is tested for local freedom and then extended motion.

This approach suffers from the fact that there may be an exponential number of candidate partitionings to test, even though few satisfy physical constraints. For instance, consider the planar assembly in figure 4.1, consisting of $n-2$ interlocking pieces sandwiched between two plates. A connected partitioning will be generated for each way to group the center pieces with the two plates, so there are 2^{n-2} ways to assign the interlocking pieces and more than 2^{n-2} connected partitioning. However, there are only two locally free subassemblies of the sandwich assembly—the top plate and the bottom plate.

In addition, there are some times when unconnected partitions may be desired, such as when an assembly cannot be built using connected subassemblies. Without the connectedness constraint, the generate-and-test approach is clearly impractical because an exponential number of candidate operations will always be generated.

The generate-and-test approach first identifies subassemblies, and then finds the motions that their contacts allow. The method described in this chapter reverses this order, by first identifying a critical set of motion directions for the assembly, and then finding the subassemblies that are free in each direction. Only a polynomial number of motions need be considered, so the algorithm is able to find a locally free subassembly in polynomial time. For clarity, section 4.2 first describes the method limiting motions to local translations.

Section 4.3 then extends this to the general case of local rotations and translations.

4.2 Partitioning for Translations

Let C be the set of all k point–plane contact constraints in the connection graph $C(A)$ of A. Remember that a constraint $c \in C$ is described by a 6-vector $n_c^T J_c$ and a pair of contacting parts that I will denote (P_{c1}, P_{c2}).

I first restrict attention to translational local motions d, where

$$d = (\dot{x}, \dot{y}, \dot{z}).$$

A translational local motion d violates contact c_i when $n_i^T d < 0$. A subassembly S is *locally free in translation* when there exists a translation d that violates no contacts between S and $A \setminus S$.

4.2.1 Directional Blocking Graph

The subset of the constraints C violated by a local translation d defines a *directional blocking graph* (or *DBG*) representing the blocking relationships between the parts in A in direction d. The directional blocking graph $G(d, A)$ is a directed graph with nodes representing the parts of A. An arc connects parts P_{c1} and P_{c2} in $G(d, A)$ if and only if there exists a constraint c such that d violates c. In other words, an arc connects part P_1 to P_2 when P_1 undergoing motion d is blocked by P_2 (unless P_2 also moves). If there are m links in $C(A)$ then $G(d, A)$ can have at most $2m$ arcs. Because it encodes constraints from contacts, $G(d, A)$ is called a *contact* DBG; Chapter 5 considers extended translations, which induce *extended* DBGs.

Figure 4.2 shows the DBGs induced by a vertical and a horizontal translation in the crate assembly of figure 1.1. For instance, in translations to the right , the `cargo` is constrained by its contacts with the `box`, so an arc connects the `cargo` node to the `box` node in the DBG of figure 4.2b. Even though in an extended translation to the right the `lid` would hit the `cargo`, no arc connects the two.

A subassembly $S \subset A$ is locally free in direction d if and only if $(S, A \setminus S)$ is a *directed partitioning* of $G(d, A)$. A directed partitioning of a directed graph G is a pair of subsets (S_1, S_2) such that $\{S_1, S_2\}$ is a partition of the nodes of G and no arcs in G connect nodes in S_2 to nodes in S_1. For instance, in the DBG of figure 4.2b, $(\{\texttt{lid}, \texttt{screw1}, \texttt{screw2}\}, \{\texttt{box}, \texttt{cargo}\})$

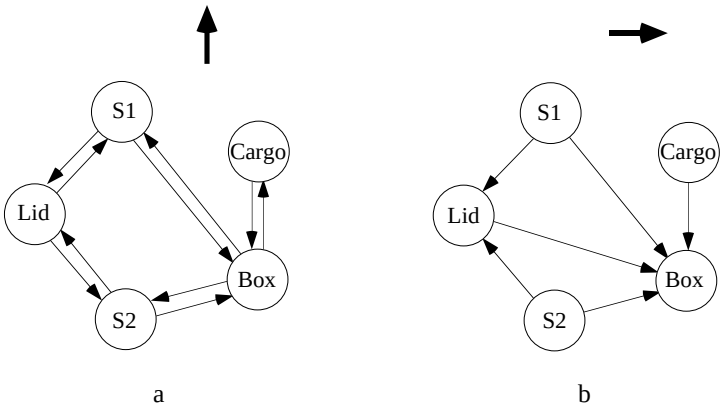

Figure 4.2: Two directional blocking graphs for the crate assembly

is a directed partitioning. If $G(d, A)$ is strongly connected[1], then clearly no directed partitioning of $G(d, A)$ exists. If $G(d, A)$ is not strongly connected, then at least one strong component of $G(d, A)$ must have no outgoing arcs, and this component is a locally free subassembly in direction d.

4.2.2 Non-directional Blocking Graph

Because two motions d_1 and d_2 differ only in velocity when $d_1 = sd_2$, $s > 0$, we restrict $|d| = 1$. Then the directions of translation d are represented as points on the unit sphere S^2. Each constraint c in C defines a plane $n_c^T d = 0$ that cuts the unit sphere along a great circle. Translations on the great circle are sliding motions for constraint c, while translations on either side represent breaking and violating motions. The set of great circles for all $c \in C$ determines an arrangement of *cells* on S^2 (figure 4.3). The cells are of three types:

Vertices lie at the intersection of two or more great circles.

Edges are maximal open connected arcs of great circles that do not include vertices.

Faces are maximal open connected components of the sphere not intersecting an edge or vertex.

There are $O(k^2)$ cells in the arrangement. The cells are regular in the following sense: for any two local motions d_1 and d_2 in the same cell, $G(d_1, A) = G(d_2, A)$. This is true because

[1] A *strongly connected component* (or *strong component*) of a directed graph is a maximal subset of nodes such that for any two nodes n_1 and n_2 in this subset, a path connects n_1 to n_2 and a path connects n_2 to n_1. A graph is strongly connected if it has only one strong component.

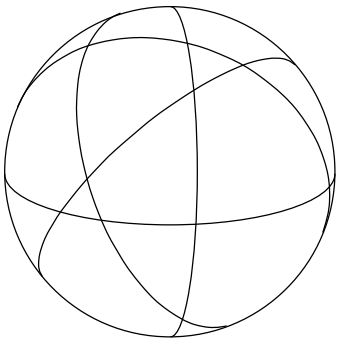

Figure 4.3: An arrangement of great circles on the sphere

d_1 and d_2 lie on the same side of all the great circles in the arrangement; thus they violate the same set of contacts. Define the graph $G(f)$ for a cell f to be $G(d, A)$ for motions $d \in f$.

The arrangement on the sphere, together with the DBGs for each cell, constitute a *non-directional blocking graph* (or *NDBG*) for A. In many cases the NDBG will not be constructed explicitly, or it will be constructed incrementally, without storing the whole NDBG at any one time. The non-directional blocking graph resembles Wolter's *assembly constraint graph* [68]; however, the ACG represents blocking relationships only in a small, heuristically chosen set of directions, and is the input to an assembly sequencer instead of being computed. In contrast, the NDBG represents the part blocking relationships in *all* directions, and can be computed directly from the connection graph of A.

For any two neighboring cells in the NDBG, one of the cells must be on the boundary of the other cell. For instance, a vertex bounds the edges and faces adjacent to it, and an edge bounds the faces on either side. The following property holds between any two neighboring cells:

Property 4.1 *For any two cells f_1 and f_2 such that f_1 is on the boundary of f_2, if there exists an arc from P_i to P_j in $G(f_1)$, this arc also exists in $G(f_2)$.*

Proof: Clearly f_1 and f_2 are not on opposite sides of any great circle. In addition, the set of great circles intersecting f_1 is a superset of the great circles intersecting f_2. Therefore any contact violated by motions in f_1 is violated by motions in f_2, so any arc in $G(f_1)$ is present in $G(f_2)$.□

In some cases $G(f_1) = G(f_2)$ for cells f_1 on the boundary of f_2, and sometimes the edges of $G(f_1)$ are a proper subset of the edges of $G(f_2)$. The former case happens when motions in cell f_2 break the corresponding contacts for the great circles intersecting at f_1. When f_2

is on the violating side of a constraint circle, then points f_1 on the circle will slide on that contact, so $G(f_1)$ will not include the corresponding arc of $G(f_2)$.

4.2.3 Finding a Locally Free Subassembly

Property 4.1 implies that if a graph $G(f_1)$ is strongly connected, then for all cells f_2 bounded by f_1, the graphs $G(f_2)$ are strongly connected as well. The vertices of the arrangement thus correspond to *critical motions*. If the DBG for a face f is not strongly connected, then the graphs $G(v)$ for vertices on the boundary of f are also not strongly connected. Hence it suffices to check the strong connectedness of $G(d, A)$ for all critical motions d at the intersection of two planes from the set C. If any such graph $G(d, A)$ is not strongly connected, then one strong component with no outgoing arcs is a locally free subassembly in direction d. The one exception occurs when all the constraint planes in C are parallel, so the resulting arrangement has no vertices. In this case Property 4.1 implies that a single point p on the great circle can be chosen and $G(p, A)$ checked for connectedness. In fact, a locally free subassembly will always be found in such an assembly. Note that the arrangement of circles on S^2 need not be explicitly computed.

For an assembly with k point–plane constraints C, there are $O(k^2)$ critical motions at the intersection v of two planes. Constructing each graph $G(v)$ requires comparing the motion v against every constraint in C, or $O(k)$ steps; since $G(v)$ has at most $2m$ arcs, its strong components can be found in $O(m)$ time [1]. Hence a translational locally free subassembly of A can be found in $O(k^3)$ time.

This computation can be reduced by noticing that there is little or no change between the DBGs of two adjacent regions. This leads to computing the DBG for one cell, then performing a systematic traversal over the arrangement and calculating the DBG for each new cell from the previous one. To this end, slightly modify the DBG by attaching a *weight* to each arc of the graph. In $G(d, A)$, the weight of the arc from P_i to P_j is the number of constraints c from P_i to P_j violated by d. The absence of an arc from P_i to P_j is treated as an arc of weight 0, and vice versa.

For two cells f_i and f_j, let the *crossing set* C_{ij} be the set of contacts violated by motions in f_j but not by motions in f_i. Property 4.1 implies that C_{ij} is empty when f_j is on the boundary of f_i. Then given the DBG of a cell f_i on the boundary of cell f_j, the DBG of f_j can be computed using the following *crossing rule*:

Initialize $G(f_j)$ to $G(f_i)$. For every $c \in C_{ij}$, add 1 to the weight of the arc from

P_{c1} to P_{c2} in $G(f_j)$.

For every contact $c \in C_{ij}$ the move from f_i to f_j corresponds to a change from sliding on c to violating c, so the crossing rule adds one to the weight of the corresponding arc in $G(f_j)$. By Property 4.1, $G(f_j)$ is strongly connected whenever $G(f_i)$ is.

Similarly, if f_j is a cell on the boundary of f_i, a similar crossing rule applies:

> Initialize $G(f_j)$ to $G(f_i)$. For every $c \in C_{ji}$, subtract 1 from the weight of the arc from P_{c1} to P_{c2} in $G(f_j)$.

In this case, some contacts are changing from violating to sliding contacts. The strong components of $G(f_j)$ will be the same as those of $G(f_i)$ unless an arc is actually deleted in applying the crossing rule.

To take advantage of the crossing rules, the arrangement of great circles on the sphere is computed explicitly. A central projection from the origin is used to map S^2 onto two parallel planes tangent to the sphere and not parallel to any great circle. The arrangement of great circles now becomes an arrangement of lines in each plane, where every cell in the planar arrangement corresponds to a cell on the sphere. Since a subassembly S is locally free in direction d exactly when $A \setminus S$ is locally free in direction $-d$, it suffices to consider one plane. The *PARTITION* algorithm can now be stated as follows:

1. Compute the vertices, edges, and faces of the planar arrangement and their adjacency relations, storing the set C_{ij} with the adjacency link between every pair of neighboring cells f_i and f_j.

2. Compute the DBG for an arbitrary cell f_0.

3. Perform a systematic traversal of the arrangement, computing the DBG for each new cell from the preceding cell using crossing rules. If any DBG is not strongly connected, one of its strong components is a locally free subassembly in the corresponding direction.

The cells in the plane and their adjacency relations can be computed in optimal $\Theta(k^2)$ time using a topological sweep [18, 27]. The cost of executing a crossing rule from cell f_i to cell f_j is proportional to the size of the crossing set C_{ij} (or C_{ji}). Although a single C_{ij} may include k contacts, each contact is only a member of crossing sets along its circle, and only those sets on the violating side of the circle. A circle is adjacent to at most $4k$ regions on

one side, so the sum of the sizes of all the crossing sets is bounded by $4k^2$. Hence the total amortized cost to incrementally construct all the DBGs is $O(k^2)$. Finally, checking each DBG for strong connectedness requires $O(m)$ time. We now have the following theorem.

Theorem 4.2 *Let A be an assembly of n parts with m contacts described as k point–plane constraints. It can be decided in $O(mk^2)$ steps whether there is a proper subassembly of A that is locally free in A for translations. Such a subassembly and a valid direction of translation can be computed in the same number of steps.*

In some assemblies with complicated contacts, $m \ll k$, and in those cases the more complicated algorithm saves significant time.

4.3 Partitioning for General Local Motions

Now consider general rigid motions ΔX, where

$$\Delta X = (\dot{x}, \dot{y}, \dot{z}, \dot{\alpha}, \dot{\beta}, \dot{\gamma}).$$

A motion ΔX violates contact c if and only if $n_c^T J_c \Delta X < 0$. As in translation, the subset of the constraints C violated by a local motion ΔX defines a weighted directional blocking graph $G(\Delta X, A)$, in which the weight of the arc from part P_i to part P_j is the number of constraints c between P_i and P_j violated by ΔX.

Again restrict $|\Delta X| = 1$, so the set of all local motions ΔX make up the unit sphere in six dimensions S^5. Each constraint c in C defines a hyperplane that divides S^5 in half. Motions ΔX on the hyperplane are sliding motions for c, and motions on either side are breaking and violating motions, respectively. The set of hyperplanes for all constraints C determines an arrangement of $O(k^5)$ cells on S^5, where the cells consist of open 5-dimensional sets of S^5 and various relatively open d-dimensional sets bounding them, for $0 \leq d < 5$ [27].

The cells of the arrangement on S^5 are regular in the same sense as in the translational case: for any two local motions ΔX_1 and ΔX_2 in the same cell, $G(\Delta X_1, A) = G(\Delta X_2, A)$. Again define the graph $G(f)$ for a cell f to be $G(\Delta X, A)$ for motions $\Delta X \in f$. The arrangement on S^5 and the corresponding DBGs constitute a non-directional blocking graph of A for local rigid motions. Property 4.1 holds equally for two neighboring cells in this NDBG. Thus the vertices of the arrangement on S^5 (the points at the intersection of 5 or more hyperplanes) constitute a set of critical local motions. To find a locally free subassembly of A it suffices to check the graphs $G(v)$ for all vertices v in the arrangement on S^5.

An exception occurs when no set of five hyperplanes from the constraints C define a single point, so there are no critical local motions. This happens when $rank(C) < 5$. In this case additional random hyperplanes can be added to C until $rank(C) = 5$, at which time a vertex will arise. Motions in the vertex slide on all the contacts of A, so a removable subassembly will be found in the DBG of the vertex.

Since a critical motion arises from every set of five hyperplanes, there are $\binom{k}{5} = O(k^5)$ critical local motions. Constructing and checking each graph requires $O(k)$ steps, so a locally free subassembly of A for general rigid motions can be identified in time $O(k^6)$.

In addition, the faster algorithm of the section 4.2 can be extended to the case of rigid motions. The crossing rules as stated there apply directly to the local rigid motion case. The arrangement on S^5 can be constructed in $O(k^5)$ time and has $O(k^5)$ cells [27]. The number of neighboring cells of a hyperplane is in $O(k^4)$, so each contact is in $O(k^4)$ crossing sets. Hence the sum of the sizes of all crossing sets is $O(k^5)$, and the amortized cost of applying the crossing rules over the whole NDBG is $O(k^5)$. Checking the strong connectedness of all the DBGs requires $O(mk^5)$ time, which dominates the other costs.

Theorem 4.3 *Let A be an assembly of n parts with m contacts described as k point–plane constraints. It can be decided in $O(mk^5)$ steps whether there is a proper subassembly of A that is locally free in A. Such a subassembly and a valid direction of rigid motion can be computed in the same number of steps.*

4.4 Incremental Construction

For some applications it may be advantageous to construct the non-directional blocking graph for an assembly A explicitly, and then make incremental changes when the geometry of the assembly changes. For instance, in a concurrent engineering system, the assembly design will evolve over time, as parts are added or removed or their geometry modified. After a small change, the NDBG for the new assembly will bear a strong resemblance to the previous NDBG. As a result, the time to generate an assembly sequence after a design change can be reduced by updating the NDBG incrementally. The procedures to update the NDBG according to several types of changes to the assembly are sketched below.

Changing contacts Consider first a change in the geometry of a part P_i in an n-part assembly A, resulting in a new assembly A'. If the new geometry of P_i leaves all previous

contacts of P_i with other parts unchanged, and introduces no new contacts, then the NDBG of A' will be the same as that of A.

On the other hand, suppose the new geometry of P_i introduces a contact constraint c with part P_j. In fact, any new contact c between P_i and P_j will produce a reciprocal constraint from P_j to P_i that must be added as well. The line corresponding to c can be incrementally added to the planar arrangement in $O(k)$ time ($O(k^4)$ time in the full rigid motion case), producing $O(k)$ (respectively $O(k^4)$) new cells [18, 27]. In addition, the DBGs for all the motions violating constraint c need to be updated; these motions are given by the cells on the violating side of the constraint line. For each such cell f, the weight of the arc from P_i to P_j in $G(f)$ must be increased by 1, a total of $O(k^2)$ steps ($O(k^5)$ for rotations). Finally, adding arcs can only increase the strong connectedness of a DBG, so the new contact will never allow assembly operations for A' that were not possible for A.

Contacts can be deleted using similar methods, and changing a contact constraint can be performed by deleting the old constraint and adding the new one.

Adding or deleting parts When a new part P_{n+1} is added to A, the NDBG can be modified in two steps. First P_{n+1} is added to the DBG for each cell. Then the contacts between P_{n+1} and the initial parts of A are added in the manner above.

Merging or splitting parts Merging two parts or splitting one part into two smaller parts are common operations in assembly design. For instance, if one area of a part is subjected to higher stresses than another area, the part can be divided to allow the low-stress section to be constructed of lighter or cheaper material. On the other hand, the cost of manufacturing a product can often be reduced by merging parts that do not move relative to each other in the finished product. The composite part is made as one piece, such as with a molding or stamping operation, thereby saving assembly costs.

Let P_i and P_j be two contacting parts in A that have been merged in A'. The contact constraints between P_i and P_j are no longer relevant, so they can be deleted as above. Then for each DBG in the NDBG, P_j is removed and the weights of any arcs incident to it (either into or out of P_j) are added to the corresponding arcs of P_i. For example, if the arc from P_j to P_h has weight 3, then 3 is added to the weight of the arc from P_i to P_h in the new DBG.

When a part is split into two sections, the reverse of the above process is performed.

The contacts of the original part are divided into those touching each section. A new node is created in each DBG having the corresponding incoming and outgoing arcs, and finally the contacts between the two sections are added to the NDBG.

Whenever the number of contact or part changes is much smaller than the number of contacts and parts in the assembly, incrementally updating the NDBG will save considerable time. In addition, the updates can be performed in the background while the designer works, so that the NDBG is ready when an assembly sequence is requested. These procedures have not been implemented in GRASP.

4.5 Connected Subassemblies

As noted in section 2.4.4, in many applications it is desirable to generate only connected partitionings of the assembly A. The above method can be extended to generate only connected, locally free partitionings, using the connection graph of A. Clearly the connection graph $C(A)$ is related to the graph $G(\Delta X, A)$ for a local motion ΔX. Specifically, if an arc connects P_i to P_j in $G(\Delta X, A)$, then a link connects P_i and P_j in $C(A)$.

If $C(A)$ is not connected and has exactly two connected components, then they must be locally free from each other. Obviously if $C(A)$ has more than two connected components, then A cannot be partitioned into two connected subassemblies. For the following, assume $C(A)$ is connected.

If S is a locally free subassembly of A in direction ΔX found with the algorithm above, then S is a strong component of $G(\Delta X, A)$ and therefore connected in $C(A)$. If $A \setminus S$ is not connected in $C(A)$, then the connected components C_i of $A \setminus S$ have no arcs between them in $G(\Delta X, A)$. Hence a new removable subassembly can be constructed by adding all but one C_i to S. Furthermore, since A is connected, the removal of S disconnected every C_i from the rest, so S must be connected to each C_i in A. Therefore choose an arbitrary connected component C_1. The subassembly $A \setminus C_1$ is connected and locally free from C_1 in direction ΔX.

Theorem 4.4 *Let A be an assembly of n parts with m contacts described as k point–plane constraints. It can be decided in $O(mk^2)$ steps whether there is a proper subassembly S of A such that S and $A \setminus S$ are connected in $C(A)$ and S is locally free in A for translations. Such a subassembly and a valid direction of translation can be computed in the same number of steps.*

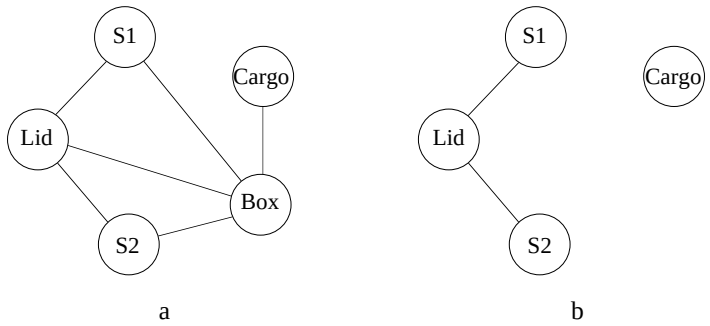

Figure 4.4: (a) the connection graph for the crate assembly in figure 1.1 and (b) the connected components after removal of the box

Theorem 4.5 *Let A be an assembly of n parts with m contacts described as k point–plane constraints. It can be decided in $O(mk^5)$ steps whether there is a proper subassembly S of A such that S and $A \setminus S$ are connected in $C(A)$ and S is locally free in A. Such a subassembly and a valid direction of rigid motion can be computed in the same number of steps.*

For example, consider the connection graph for the crate assembly shown in figure 4.4a. The subassembly {`box`} is locally free for motions to the right in the crate. After removing {`box`}, the remaining parts of the crate comprise two connected components of the connection graph of the crate (figure 4.4b). Either of these can be added to {`box`} to make a connected, locally free subassembly. Thus both {`box`, `cargo`} and {`box`, `lid`, `screw1`, `screw2`} are connected, locally free subassemblies of the crate.

4.6 Generating All Locally-Free Subassemblies

In most uses of the above algorithms, more than one locally-free subassembly will need to be generated. Local freedom is not a sufficient condition for a subassembly to be removable; as a result a number of locally free subassemblies might need to be tested against global motion constraints to find a valid assembly operation. In addition, more than one removable subassembly will need to be found when alternative assembly sequences are desired. This section presents procedures to find the set of all locally free, and connected locally free, partitionings of an assembly.

4.6.1 Unconnected Subassemblies

First consider the case when the subassemblies are not required to be connected. The set of all locally free subassemblies of an assembly A is equal to the union of the sets of locally free subassemblies for all directional blocking graphs of A. Thus it suffices to find the set of locally free subassemblies for each DBG.

Figure 4.5 shows procedure *ALL-SUBASSEMBLIES*, which generates the set of all locally free subassemblies in a direction ΔX, given $G = G(\Delta X, A)$. As in the previous sections, parts of A that form a strong component of G must necessarily move together, since they are interlocked in direction ΔX. Accordingly, *ALL-SUBASSEMBLIES* first calls function *REDUCE*, which computes the *reduced graph* of the strong components of G. The reduced graph R is a directed acyclic graph with a node corresponding to each strong component of G. An arc connects node n_1 to node n_2 in R if and only if an arc of G connects one of the parts of n_1 to one of the parts of n_2. Hence the directed partitionings of R correspond one-to-one with the directed partitionings of G. In the discussion below and in the algorithms I will not distinguish between the nodes of R and the strong components of G. Let $PRED(n, R)$ be the set of all predecessors of node n in R (including n), and $SUCC(n, R)$ be the set of all successors of n in R (including n). Let $D(R)$ be the set of directed partitionings of R.

The recursive procedure *ALL-REDUCED* enumerates $D(R)$. It maintains the following invariant:

> At each invocation of *ALL-REDUCED*, S_1 and S_2 are disjoint subsets of the nodes of R. In addition, S_1 is closed under the *PRED* function, and S_2 is closed under the *SUCC* function.

In other words, S_1 is a locally free subassembly of A in direction $-\Delta X$, and S_2 is locally free in direction ΔX. The nodes not in S_1 or S_2 are *undecided* nodes.

$ALL\text{-}REDUCED(S_1, S_2)$ generates all directed partitionings (S_1', S_2') of R such that $S_1 \subseteq S_1'$ and $S_2 \subseteq S_2'$. If $S_1 \cup S_2 = A$, then (S_1, S_2) is the only such partitioning. Otherwise, any undecided node n can act as a pivot: the directed partitionings (S_1', S_2') of R are divided into two sets: those with n in S_1', and those with n in S_2'. These two sets are computed by recursive calls to *ALL-REDUCED*. Since S_1 must be closed under the *PRED* function, the predecessors of n are added to S_1 in the first recursive call; similarly the successors of n are added to S_2 in the second recursive call. The call $ALL\text{-}REDUCED(\emptyset, \emptyset)$ thus enumerates $D(R)$.

```
Procedure ALL-SUBASSEMBLIES(A, G)
    D ← ∅;
    R ← REDUCE(G);
    ALL-REDUCED(∅, ∅);
    return(D);
end; {procedure}

Procedure ALL-REDUCED(S1, S2)
    if S1 ∪ S2 = A
        if S1 ≠ ∅ and S2 ≠ ∅
            push((S1, S2), D);
        else return;
    else
        choose a node n ∈ [A \ (S1 ∪ S2)];
        ALL-REDUCED(S1 ∪ PRED(n, R), S2);
        ALL-REDUCED(S1, S2 ∪ SUCC(n, R));
    end; {else}
end; {procedure}
```

Figure 4.5: The procedure to find all locally-free subassemblies for a DBG

Since in general there might be a large number of locally free subassemblies, the time complexity of *ALL-SUBASSEMBLIES* is dependent on the number s of directed partitionings of G. Consider the binary tree of recursive calls of *ALL-REDUCED*. Each leaf in the call tree corresponds to either a directed partitioning of R or an empty set S_1 or S_2. The latter case can happen at most twice—once when all pivot nodes are placed in S_1, and once when all pivot nodes are placed in S_2. Hence the call tree has at most $s + 2$ leaves and at most $s + 2 - 1$ internal nodes. The set operations in a single call to *ALL-REDUCED* can all be done in time linear in m, so *ALL-REDUCED*$(\emptyset, \emptyset)$ requires $O(ms)$ time. The reduced graph of a DBG can be found in time $O(m)$ [1], so *ALL-SUBASSEMBLIES* is also $O(ms)$.

To find all the locally free subassemblies of A, $D(G)$ must be calculated and combined for all DBGs G of A. A locally free subassembly might be found in many DBGs of A, so s can be as large as the number of locally free subassemblies of A. Using a trie structure [1] to represent the global list of partitionings, $D(G)$ can be merged with the global list in $O(ns)$ time. For local translations there are $O(k^2)$ DBGs to analyze, and for local rigid motions $O(k^5)$ of them. We now have the following theorems.

Theorem 4.6 *Let A be an assembly with m contacts described as k point–plane constraints. The set of all s locally free subassemblies of A in translation can be found in $O(msk^2)$ time.*

Theorem 4.7 *Let A be an assembly with m contacts described as k point–plane constraints. The set of all s locally free subassemblies of A can be found in $O(msk^5)$ time.*

4.6.2 Connected Subassemblies

The locally free connected partitionings of A could be found by generating all locally free subassemblies of A (using procedure *ALL-SUBASSEMBLIES*) and checking each partitioning for connectedness. However, this may require testing a large number of partitionings to find a few connected ones.

Assume that $C(A)$ is connected; the other cases are considered in section 4.5. Figure 4.6 shows procedure *CONN-SUBASSEMBLIES*, which combines aspects of *ALL-SUBASSEMBLIES* and *CONN-PARTITIONINGS* (from figure 3.9) to directly find the locally free connected partitionings of a DBG. The strong components of G (i.e. the nodes of R), as well as the predecessor and successor sets of a node of R, must be connected in $C(A)$, because the arcs of G derive from the links of $C(A)$. Since the parts in a node of R are connected and must move together, each node can be considered a single part for that direction of motion. *CONN-SUBASSEMBLIES* works much the same way as *CONN-PARTITIONINGS*; it continually chooses pivot nodes and generates the partitionings including the pivot in either S_1 or S_2.

After computing the reduced DBG R, *CONN-SUBASSEMBLIES* chooses pivot nodes n_i as before to split the directed connected partitionings of R into two sets (line 1). For each such set of partitionings, it chooses another pivot node n_j to split it again (line 2). This is necessary because *CONN-REDUCED* must be called with two non-empty sets S_1, S_2. n_j is chosen so as to keep S_1 connected in all calls to *CONN-REDUCED*. For each pivot node included in S_1, the predecessors of the pivot must also be in S_1; for each pivot placed in S_2, its successors must be in S_2 also. Whenever the added predecessors and successors cause S_1 and S_2 to intersect, clearly no directed partitionings exist with those pivot choices.

The following invariant is maintained on each call to *CONN-REDUCED*:

> S_1 and S_2 are non-empty disjoint subsets of the nodes of R, and closed under the *PRED* and *SUCC* functions, respectively. In addition, S_1 is connected in $C(A)$.

```
Procedure CONN-SUBASSEMBLIES(A, G)
    D ← ∅;
    R ← REDUCE(G);
    S3 ← ∅;
(1) while there is a node ni of R, ni ∉ S3
        S1 ← PRED(ni, R);
(2)     while there is a neighbor nj of S1 in C(A) such that nj ∉ S3
            S2 ← S3 ∪ SUCC(nj, R);
            if S1 ∩ S2 = ∅
                CONN-REDUCED(S1, S2);
            S1 ← S1 ∪ PRED(nj, R);
        end; {while}
        S3 ← S3 ∪ SUCC(ni, R);
    end; {while}
    return(D);
end; {procedure}

Procedure CONN-REDUCED(S1, S2)
    CC ← CONNECTED-COMPONENTS(A \ S1, C(A));
    if |CC| > 1
        if for some Ci ∈ CC,  Ci ⊇ S2
            S1 ← S1 ∪ ⋃j≠i Cj;
        else return;
    end; {if}
    push((S1, A \ S1), D);
    for each neighbor ni of S1 in C(A), ni ∉ S2
(3)     CONN-REDUCED(S1 ∪ PRED(ni, R), S2);
        S2 ← S2 ∪ SUCC(ni, R);
    end; {for}
end; {procedure}
```

Figure 4.6: The procedure to find all connected, locally-free subassemblies for a DBG

The procedure *CONN-REDUCED* works much like *CONN-SUPERSETS*, with the added constraint that whenever a pivot is added to S_1 or S_2, its predecessors or successors must be added, respectively.

Procedure *CONN-REDUCED* first computes the connected components CC of $A \setminus S_1$. If $A \setminus S_1$ is connected, then (S_1, S_2) is a directed connected partitioning. If $|CC| > 1$, then the only supersets S_1' of S_1 that will have a connected complement $A \setminus S_1'$ will include all the CC but one. As before there are two cases:

- If S_2 intersects more than one of the CC, then all supersets S_2' of S_2 are unconnected. In this case the procedure returns.
- If S_2 is contained in one connected component C_i, then that C_i cannot be added to S_1. Clearly then, all components $C_j, j \neq i$, must be added to S_1, and then (S_1, S_2) is a connected partitioning.

After one partitioning (S_1, S_2) is found, then each node n_i connected to S_1 and not in S_2 becomes a new pivot. Node n_i divides the superset partitionings (S_1', S_2') into those containing n_i in S_1' and those with n_i in S_2'. Recursive calls to *CONN-SUPERSETS* enumerate those partitionings, including the predecessors or successors with n_i, respectively. If all neighbors n_i are in S_2, then no superset of S_1 will be connected.

Not counting the time taken in *CONN-REDUCED*, *CONN-SUBASSEMBLIES* requires time $O(n^2m)$ in a straightforward implementation. However, using a marking scheme to do the set operations and *PRED* and *SUCC* calculations, this can be reduced to $O(nm)$ time. This optimization is straightforward but tedious, and will not be described here. Each call to *CONN-REDUCED* requires time linear in m, not including the recursive calls. There are $O(n^2)$ initial calls, and $O(ns)$ calls resulting from line (3). Hence *CONN-SUBASSEMBLIES* executes in time $O(nms)$. *CONN-SUBASSEMBLIES* must be called once for each DBG and the results combined as for the unconnected case.

Theorem 4.8 *Let A be an assembly of n parts with m contacts described as k point–plane constraints. The set of all s connected, locally free partitionings of A in translation can be found in $O(nmsk^2)$ time.*

Theorem 4.9 *Let A be an assembly of n parts with m contacts described as k point–plane constraints. The set of all s connected, locally free partitionings of A can be found in $O(nmsk^5)$ time.*

4.7 Implementation

Both the translational and rotational versions of the *PARTITION* algorithm given above have drawbacks for a practical assembly sequencing method. The first is limited to translations, which rules out its direct application to the many mechanical assemblies with threaded contacts. On the other hand, the rotational version can find any locally free subassembly in a rigid assembly, but at the cost of examining $O(k^5)$ DBGs.

The assemblies of figures 3.13 and 3.14 are anomalies in the real world. Almost all rotational motions used in real assemblies are either screwing motions required by threaded contacts or pure rotations around the axis of a cylindrical contact. The version of *PARTITION* implemented in GRASP is a hybrid algorithm that has the same time complexity of the translational version. It finds all locally free subassemblies that the translational version does, plus any subassemblies S for which the separating motion is suggested by a threaded or revolute contact between S and $A \setminus S$. In fact, the resulting algorithm finds the same set of assemblies as Homem de Mello's *GET-FEASIBLE-DECOMPOSITIONS* does for local motions, but with a much lower worst-case time complexity.

The hybrid algorithm begins by finding all subassemblies that are locally free in translation as described in section 4.2. Then it generates a list of *suggested motions* given by the nonplanar contacts of the assembly. For each threaded contact, a twisting trajectory with the same axis and pitch as the contact is added to the list of suggested motions. For each cylindrical contact, a pure rotation around the axis of the contact cylinder is generated. Then for each suggested motion ΔX, a DBG $G(\Delta X, A)$ is constructed and analyzed for connected, locally free partitionings.

There can be at most $O(k)$ suggested directions ΔX, and building the DBG for each ΔX requires examining all k contacts for compatibility with ΔX. Finding the connected locally free partitionings of the DBGs for all suggested directions is completed in time $O(nmsk)$, which is dominated by the $O(nmsk^2)$ time bound of the translational case.

Even if an assembly sequencer were required to disassemble products such as that in figure 3.14, the *PARTITION* algorithm as implemented in GRASP would be a practical first step. Most assemblies would be quickly partitioned, and in the few cases where the algorithm could not find a partitioning of the assembly, the full rotational version could be invoked.

Assembly	Number of Parts	*DECOMPOSE*			*PARTITION*		
		Node	Tree	Graph	Node	Tree	Graph
Engine	12	1.0	31	48	0.5	30	42
Bell	17	2.6	18	528	0.9	16	522
Bell	22	4.7	25	3245	1.4	21	2186
Engine	30	4.9	111	—	3.6	112	—
Skin	36	—	—	—	3.4	222	—
Engine	42	74	319	—	8.5	189	—

Table 4.1: Experimental timings comparing procedures *DECOMPOSE* and *PARTITION*, in seconds

4.8 Experiments

Table 4.1 shows experimental results obtained with the hybrid *PARTITION* on the assemblies described in Appendix B. For each assembly A, three times are given for both procedure *DECOMPOSE* (based on generate-and-test) given in figure 3.8 and *PARTITION*:

- the time required to partition the root node (i.e. find all connected locally free partitionings of A),
- the time to find one disassembly AND-tree for A, and
- the time to build the full AND/OR graph for A. For the larger assemblies, the full AND/OR graph is too large to generate in practice.

Several points should be noted about the results in table 4.1. For most assemblies, the two procedures are quite competitive in running times. However, the skin machine represents a bad case for procedure *DECOMPOSE*, since its connection graph has a very large number of cut-sets. As a result, *DECOMPOSE* was stopped after two days of trying to partition the skin machine. In that time it examined over 100,000 connected partitionings.

For the assemblies other than the skin machine, a relative advantage of *PARTITION* in decomposing the root node of the graph does not necessarily translate into a large decrease in sequencing time. For instance, *PARTITION* partitioned the 22-part bell in 1.4 seconds compared to 4.7 for *DECOMPOSE*, but the times to generate the whole AND/OR graph were 2186 and 3245 seconds, respectively. This is due in part to the other costs of sequencing, such as checking for global motions. In addition, the generate-and-test method is faster for small assemblies, which make up the bulk of the AND/OR graph.

The time required for *DECOMPOSE* to find the locally free subassemblies of the 22-part bell is much lower in table 4.1 than the 54 seconds reported in [64]. The previous figure was using Homem de Mello's cut-set generation algorithm [34], which is much slower in some cases than that given in figure 3.9.

Chapter 5

Partitioning for Extended Translations

The previous chapter described methods to find locally free subassemblies of an assembly in polynomial time. However, local freedom is only a necessary constraint; a globally valid path must be found for the removal of any locally free subassembly. In fact, a large number of locally free subassemblies might be tried before a globally free one is found. To avoid the generate-and-test cycle in such cases, an algorithm is desired that would find globally-free subassemblies in an efficient manner.

In addition, the basic module for extended collision checking in GRASP (section 3.4) checks for globally-valid motions by extending several local motions to infinity and checking for collisions using a sweeping computation. The translations are chosen heuristically based on the shape of the local freedom cone. This method is obviously not complete; in some cases a translation that would separate two subassemblies will not be tested.

This chapter presents an algorithm that corrects both of these inadequacies for assemblies of polyhedral parts and global motions consisting of single extended translations. Specifically, the following problems are addressed:

1. Given an assembly A of n polyhedral parts, decide whether there is a direction d and a subassembly $S \subset A$ such that a translation along d separates S from the remaining parts $A \setminus S$, and if so identify d and S.

2. For an assembly A of polyhedra, return a list of all subassemblies $S \subset A$ that can be separated from $A \setminus S$ by a single translation.

A method is given below to solve the first problem in polynomial time and the second problem in output-sensitive polynomial time.

Arkin, Connelly and Mitchell [3] address a planar version of problem 1 above. They use the concept of monotone paths among polygonal obstacles to identify a removable subassembly of simple polygons in the plane. The methods in [3] do not extend directly to the three-dimensional case. However, Mitchell has independently shown that extended translations for partitioning an assembly can be found in polynomial time [47].

For an assembly with n parts and v total vertices, I give an algorithm to identify a removable subassembly in $O(n^2v^4)$ time. The algorithm is closely related to the method given in the previous chapter. When applied to two polyhedral parts or subassemblies, the procedure becomes an algorithm to find the set of all translations separating the parts in $O(v^4)$ time, which is optimal in the worst case. Complete translational assembly sequences for polyhedral parts can easily be computed by recursive application of the method. I describe an implementation of the procedure and the results of various assembly planning experiments using the program.

5.1 Extended Blocking Graphs

The algorithm to find a subassembly of A that can be separated from the rest of A by a single translation is quite similar to the partitioning algorithm presented in the previous chapter. The set of translations is represented as the points on the unit sphere S^2, which is divided into regions based on which parts block the motions of others in each region. In this case extended translations are considered instead of local motions. A weighted blocking graph is associated with each region, crossing rules are defined, and the sphere is searched for a region whose blocking graph has a free subassembly.

Let $A = \{P_1, \ldots, P_n\}$ be an assembly of n polyhedral parts with a total of v vertices. Assume for now that the parts are not in contact; this restriction is removed below. A part P_i *collides with* another part P_j in direction d if there exists a point x in the interior of P_i such that $x + td$ is in the interior of P_j for some value t in $[0, \infty)$. A translation separating a subassembly S_1 from subassembly S_2 is a vector d such that no parts of S_1 collide with parts of S_2 in direction d. Let $G(d, A)$ denote the *extended directional blocking graph* of assembly A in direction d. $G(d, A)$ is a directed graph whose nodes are the parts of A, in which an arc connects part P_i to P_j if and only if P_i collides with P_j in direction d. Clearly

d separates subassemblies S and $A \setminus S$ exactly when $(S, A \setminus S)$ is a directed partitioning of $G(d, A)$.

The set of all translation directions d can be represented by the points on the unit sphere S^2 in three-dimensional space. For each pair of parts P_i, P_j, the configuration obstacle

$$C(P_i, P_j) = P_j \ominus P_i = \{a_j - b_i \mid a_j \in P_j,\ b_i \in P_i\}$$

is the set of placements of P_i such that P_i intersects P_j [45]. Let R_{ij} be the projection of $C(P_i, P_j)$ on the unit sphere using a central projection centered at the origin. The interior of the region R_{ij} is the set of translations along which P_i collides with P_j.

Since P_i and P_j are polyhedral parts, the C-obstacles $C(P_i, P_j)$ are also polyhedra [42, 45]. In a central projection, any line segment in space projects to an arc of a great circle on S^2, so the regions R_{ij} are bounded by arcs of great circles. Consider the set C of all bounding arcs of regions R_{ij}. C determines an arrangement of cells on S^2 of three types:

Vertices lie at the intersection of two or more arcs. Since the arcs are boundaries of regions, no arc ends without intersecting another.

Edges are maximal open connected arcs that do not include vertices.

Faces are maximal open connected components of the sphere not intersecting an edge or vertex.

The cells are regular in the following sense: for any two translations d_1 and d_2 in a cell, the extended DBGs $G(d_1, A)$ and $G(d_2, A)$ are equal. Define $G(f)$ for a cell f to be $G(d, A)$ for translations d in f. The arrangement on the sphere and the corresponding DBGs constitute an *extended non-directional blocking graph*. A parallel property to Property 4.1 holds for the cells in this arrangement.

If two parts are in contact in their initial positions, then the C-obstacle $C(P_i, P_j)$ includes the origin, and in this case the projection R_{ij} is undefined. Because the extended translations allowed by such a contact correspond to the local translations it allows, contact constraints can be added to the NDBG as with the contact DBGs of section 4.2. The non-contacting sections of P_i and P_j are then treated as described here. Including contact constraints does not add to the computing times of the methods in this chapter, but to clarify the presentation they will not be considered.

5.2 Finding a Removable Subassembly

The projected configuration obstacles R_{ij} need not be computed explicitly. The faces of P_i and P_j can be triangulated, and the configuration obstacle for each pair of triangles computed. Although the union of the triangle C-obstacles is not always equal to $C(P_i, P_j)$, R_{ij} is always equal to the union of their projections onto the sphere. To each arc in $G(d, A)$, attach a weight equal to the number of pairs of triangles from the two parts that collide for translation d. An arc of weight 0 is the same as the absence of an arc. The arrangement is then composed of regions T_{ij} corresponding to collisions between triangles T_i and T_j; the DBGs for two translations in a cell of this new arrangement are equal.

As in the local motion case, the DBGs for neighboring cells of the arrangement differ only slightly. Let the crossing set C_{ij} be the set of regions T_{hk} such that cell f_j is in the interior of T_{hk} but cell f_i is not. Crossing from f_i to f_j steps into the interior of the regions in C_{ij} or out of the regions in C_{ji} (either C_{ij} or C_{ji} will be empty depending on whether f_i is on the boundary of f_j or vice versa).

If $G(f_i)$ is known for a cell f_i on the boundary of a cell f_j, then $G(f_j)$ can be computed using the following crossing rule:

> Initialize $G(f_j)$ to $G(f_i)$. For every region $T_{hk} \in C_{ij}$, add one to the weight of the arc from P_a to P_b, where triangle T_h belongs to part P_a and T_k belongs to P_b.

Conversely, when $G(f_i)$ is known and f_j is on the boundary of f_i:

> Initialize $G(f_j)$ to $G(f_i)$. For every region $T_{hk} \in C_{ji}$, decrease the weight of the arc from P_a to P_b by one, where triangle T_h belongs to part P_a and T_k belongs to P_b.

To find a subassembly of A removable by an extended translation, the sphere is mapped to two parallel planes using a central projection from the origin. However, S can be separated from $A \setminus S$ in direction d exactly when $A \setminus S$ can be removed from S in direction $-d$. Thus it suffices to search only one planar arrangement. This gives rise to the following algorithm for finding a removable subassembly:

1. Triangulate the faces of the parts.
2. For each pair of triangles T_i, T_j from different parts, compute the projection T_{ij} of $C(T_i, T_j)$ on the plane.

3. Calculate the arrangement of boundary line segments of the regions T_{ij}.

4. Compute the crossing sets C_{hk} by traversing the boundary of every T_{ij}. For each cell f on the line, a pointer to T_{ij} is deposited in each C_{hk} toward the interior of T_{ij}.

5. For an arbitrarily selected cell f_0, compute $G(f_0)$ by comparing f_0 to every region T_{ij}.

6. Perform a depth-first traversal of all the cells in the arrangement by crossing from f_0 to neighboring cells. To step from a cell f_i to a neighboring cell f_j, calculate $G(f_j)$ from $G(f_i)$ using the crossing rules above. After visiting a cell, it is marked and not visited again. If $G(f_i)$ is not strongly connected for any cell f_i, output a direction d in f_i and the strong component of $G(f_i)$ with no outgoing arcs.

The faces of a part with v vertices can be divided into $O(v)$ triangles in optimal $\Theta(v)$ time [16] or in expected $O(v \log^* v)$ time using a simpler randomized algorithm [59]. There are $O(v^2)$ regions T_{ij}, each with a constant number of edges, so step 2 requires $O(v^2)$ operations. The arrangement induced by m line segments in the plane has $O(k)$ cells, where $k \leq \binom{m}{2}$ is the number of intersections between segments. The arrangement, including the adjacency relations between its cells, can be computed in optimal $\Theta(m \log m + k)$ time [17] and in expected $O(m \log m + k)$ time using a simple randomized algorithm [19]. Here $m = O(v^2)$, so the number of cells and the computing time for step 3 are $O(v^4)$ in the worst case.

The $O(v^2)$ edges in each segment have a total of $O(v^2)$ neighbors, so step 4 costs $O(v^4)$, and the sum of the sizes of all crossing sets is $O(v^4)$. Testing the initial face f_0 for inclusion in all regions requires $O(v^2)$ operations. The cost of computing the DBGs using the crossing rules is proportional to the number of regions in all crossing sets, so it is also $O(v^4)$. Finally, a DBG may have $n(n-1)$ arcs, so computing the strong components of all DBGs can be done in $O(n^2 v^4)$ time, which dominates the other times.

Theorem 5.1 *Let A be an assembly of n polyhedral parts with a total of v vertices. It can be decided in $O(n^2 v^4)$ steps whether there is a proper subassembly of A that can be translated to infinity without intersecting the remaining parts. An appropriate subassembly and direction can be computed in the same number of steps.*

As an example, consider the simple configuration of four cubes aligned along the x-axis in figure 5.1. The corresponding planar arrangement consists of 12 polygons; several of these

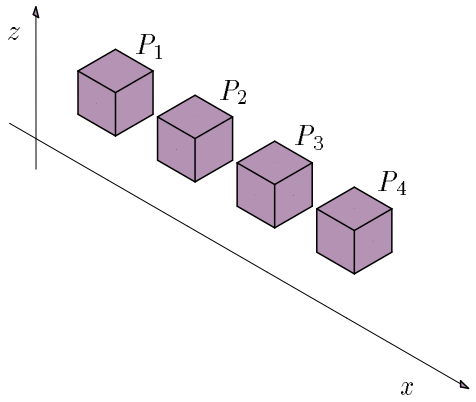

Figure 5.1: An assembly of cubes

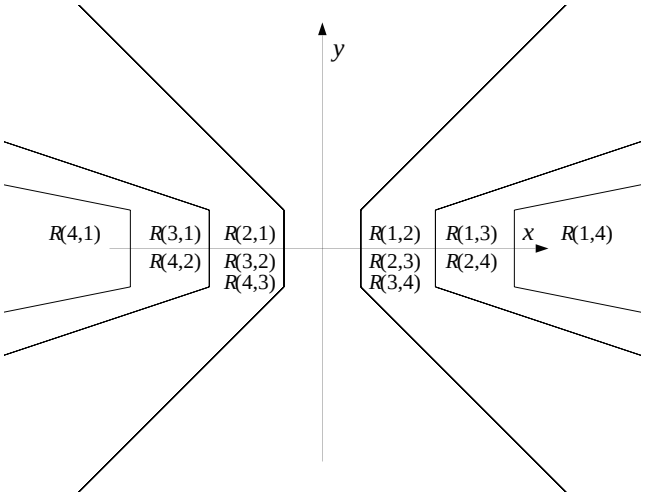

Figure 5.2: The arrangement for the assembly in figure 5.1

polygons coincide. Figure 5.2 shows the planar arrangement. The projected configuration obstacle corresponding to cubes P_1 and P_4 is the region marked $R(1,4)$ and is bounded by a line segment and two rays.

Figure 5.3a shows the directional blocking graph $G(f)$ for cell $f = R(1,4)$. $R(1,4)$ is contained in $R(1,2)$ and $R(1,3)$, so there are arcs in the graph from node 1 to nodes 2, 3, and 4, each of weight 1. $R(1,4)$ is contained in $R(2,4)$, $R(2,3)$, and $R(3,4)$. Since node 4 has no successors, it is a removable subassembly for translations in $R(1,4)$. If cubes P_2 and P_4 represent a single part P_{24}, the DBG in figure 5.3b results. Nodes 24 and 3 form a strongly connected component, so cubes 2, 3, and 4 must be removed simultaneously for translations in $R(1,4)$.

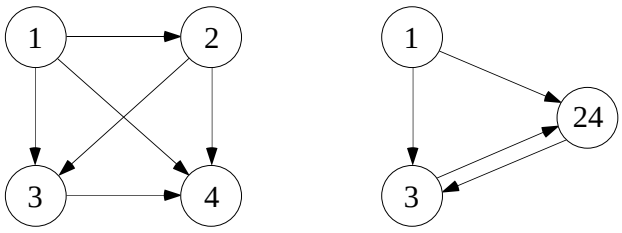

Figure 5.3: Extended DBGs for region $R(1,4)$ where (a) $P_1, \ldots, P_4$ can be moved independently (b) P_2 and P_4 must be moved simultaneously

5.3 Finding All Removable Subassemblies

In some cases, finding a single removable subassembly is not sufficient. For instance, to construct the AND/OR graph of all assembly sequences, at each node the set of all removable subassemblies must be found. The operations may also be subject to additional, non-geometric constraints in an assembly sequencing system, requiring alternatives to be found and tested. For these purposes, the set of all subassemblies removable by an extended translation can be computed by applying the algorithm *ALL-SUBASSEMBLIES* of figure 4.5 to the extended blocking graph $G(d, A)$. In this application, *ALL-SUBASSEMBLIES* finds all s subassemblies removable in a single extended DBG in time $O(n^2 s)$. This calculation must be done for each DBG, resulting in the following theorem.

Theorem 5.2 *Let A be an assembly of n polyhedral parts with a total of v vertices. The set of all s proper subassemblies of A removable by a single translation can be computed in $O(n^2 v^4 s)$ time.*

5.4 Connectedness

Some assemblies have no connected monotone assembly operation to create them, yet have unconnected subassemblies that can be mated by a single translation. An example is shown in figure 2.10. As a result, the straightforward application of the method of section 4.5 cannot succeed for an extended DBG. In addition, the procedure *CONN-SUBASSEMBLIES* of figure 4.6 cannot be directly applied to an extended DBG. Both rely on the fact that the arcs in a DBG are a subset of the arcs in the connection graph for the same assembly. This relationship holds between the connection graph of an assembly and the contact DBGs of that assembly, but does not hold between connection graphs and extended DBGs. I have

found no polynomial time algorithm to solve this problem.

Two generate-and-test methods can be used to find connected subassemblies removable in translation, but they both may require testing an exponential number of candidate operations in the worst case. The first generates removable subassemblies S and tests S and $A \setminus S$ for connectedness; the second generates locally free connected partitionings of A and then checks them for extended translations using the method of the next section. For many assemblies these methods will work, but guaranteed polynomial time algorithms are needed.

5.5 Separating Two Polyhedral Parts

The extended partitioning algorithm above can be applied in a straightforward manner to find the set of all extended translations separating two polyhedral parts or subassemblies. For instance, this algorithm could be used instead of the sweeping method described in section 3.4. There, a subassembly S is swept to find collisions with other parts, in directions chosen heuristically based on the shape of the local freedom cone of S. In contrast, the method here is guaranteed to find a separating translation if one exists.

Each subassembly is considered a single part. The arrangement on the sphere is built and searched, and all cells f_i in which one subassembly is removable from the other are collected. The translations in these cells are the set of all separating motions for the two subassemblies. Because $n = 2$, the above algorithm runs in time $O(v^4)$ for this case.

In fact, this algorithm is optimal in the worst case. The optimality directly follows from an example given by Pollack, Sharir and Sifrony [54]. The example in [54] concerns two polygons P and Q with r and s edges respectively; the number of connected components in the complement of the configuration obstacle corresponding to P and Q is proportional to r^2s^2 (figure 5.4). In our case the polygons P and Q are regarded as polyhedral parts of zero volume, and $r, s = v$; the following holds equally if P and Q are sufficiently thin polyhedra. Place P in a plane p and Q in a plane parallel to p, but distinct from p. Then the plane containing the configuration obstacle of P with respect to Q does not contain the origin, so the projection of the configuration obstacle of P with respect to Q on the sphere S^2 partitions S^2 into $\Omega(n^4)$ connected components. Therefore the set of translations separating P from Q consists of $\Omega(v^4)$ connected components.

Theorem 5.3 *The set of all translations separating two polyhedral parts with v vertices can be found in time $O(v^4)$, and this is optimal.*

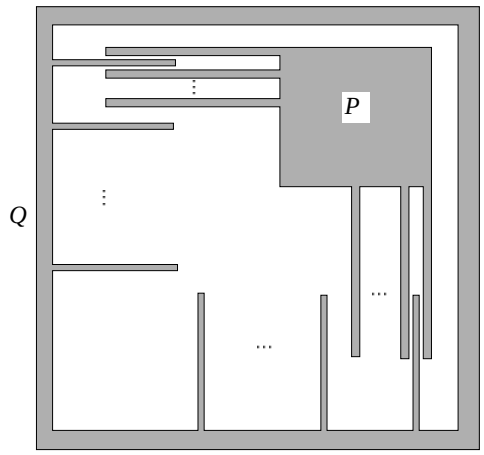

Figure 5.4: Polygons from Pollack et. al. [54]

5.6 Finding Assembly Sequences

The above method can be used to efficiently build an AND-tree representing a binary monotone assembly sequence for a polyhedral assembly in which the operations are restricted to single extended translations. The method of Theorem 5.1 is simply applied recursively to subassemblies. Since a monotone binary assembly sequence has $n-1$ operations, the method is applied $n-1$ times.

Theorem 5.4 *Given an assembly A of n polyhedral parts with v vertices, it can be decided in $O(n^3v^4)$ steps whether a binary monotone assembly sequence exists for A using only extended translations.*

5.7 Implementation

A drawback of the algorithm above is the storage requirement: the arrangement may take $O(v^4)$ space to store, which is impractical for complicated assemblies. The implementation reduces the space requirement by performing a simpler vertical line sweep [42, 56] over the arrangement of $O(v^2)$ line segments. This algorithm works as follows.

An imaginary vertical line passes over the arrangement. The cells cut by the sweep line in its current position are kept in a sorted list; the initial list is found by sorting the lines by slope. Start points and end points of segments and intersections between two segments are events, kept in a priority queue sorted by x-value. As the sweep-line moves from left

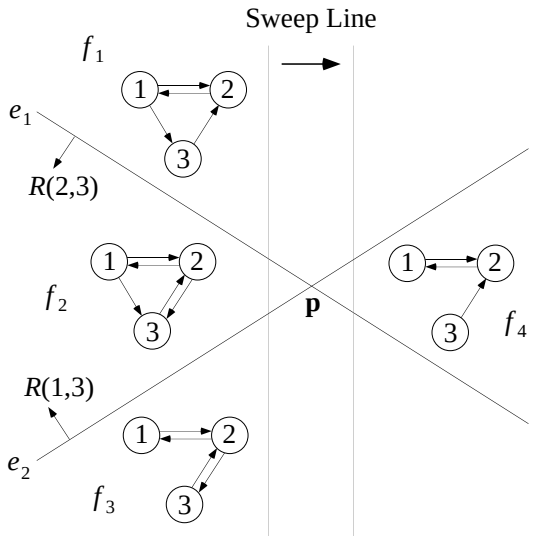

Figure 5.5: An intersection event in the sweep-line algorithm

to right, events are processed and the list of cut cells is changed accordingly. Each event can be processed in $O(\log m)$ time, so the total running time is $O((m+k)\log m)$, where k is the number of intersection events. In this case $m = v^2$, so the arrangement calculation requires $O((v^2+k)\log v)$ steps, where $k = O(v^4)$.

The sweep-line algorithm maintains the graph $G(f)$ for each cell cut by the vertical line. The graphs for cells intersecting the initial sweep-line are propagated down from an initial cell at the top of the sweep-line. To process an event, the graph for a new cell is calculated by stepping from the cell above it in the vertical line. Thus the graphs for all cells in the arrangement are calculated and checked without keeping the whole arrangement in memory. The total computing time for finding an appropriate subassembly using the modified algorithm is $O(n^2v^4 + v^4 \log v)$.

Figure 5.5 illustrates the processing of an intersection event. The interior of region $R(2,3)$ is below edge e_1, and edge e_2 is the lower boundary of region $R(1,3)$. The graphs for cells f_1, f_2, and f_3 have already been computed; all the graph links have weight one. When the sweep line processes the intersection of e_1 and e_2 at point p, the cell f_4 is entered. Edge e_2 is between f_1 and f_4 in the new sweep line, so $G(f_4)$ is computed by stepping over e_2 from $G(f_1)$. The interior of $R(1,3)$ is above e_2, so $G(f_4)$ is obtained from $G(f_1)$ by deleting the link from node 1 to node 3. Nodes 1 and 2 form a stongly connected component of $G(f_4)$, so the corresponding parts are a removable subassembly.

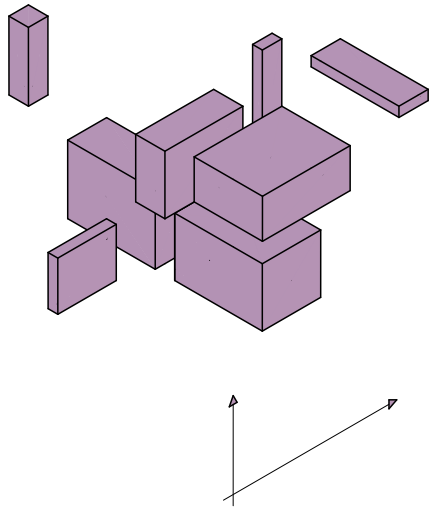

Figure 5.6: An assembly of eight random blocks

The implementation is written in C, and generates configuration obstacles for blocks instead of tetrahedra; however, the arrangement computation applies to the general case. The algorithm has not been implemented as a module in GRASP.

5.8 Experiments

To evaluate the practical computing bounds on the implementation, k random disjoint blocks were generated and linked together to form n complex objects for values of k ranging between 4 and 128 and n ranging between 2 and 16 (figure 5.6). Since each block has 8 vertices, $v = 8k$ ranges from 32 to 1024. Links were generated at random. A link between two blocks signifies that the blocks can only be moved simultaneously. Removable subassemblies were identified for these assemblies using the described implementation of the above method.

Table 5.1 shows the computing times and storage requirements observed. For each value of k and n, 32 samples were run and the average, minimum, and maximum running times recorded (t, t_{min}, and t_{max}, respectively), along with the maximum storage needed (s_{max}). In all cases the entire arrangement and all graphs were computed instead of stopping at the first removable subassembly found.

v	n	t	$10^3 t/n^4$	t_{min}	t_{max}	s_{max}
32	2	0.2	0.78	0.1	0.2	16210
64	4	0.6	0.15	0.5	0.6	19644
128	4	5.1	0.078	4.7	5.4	30908
256	8	48.7	0.046	39.4	48.7	44163
512	8	283.6	0.016	281.0	284.2	61816
1024	16	1150.7	0.0042	1120.8	1243.7	88264

Table 5.1: Computing times for partitioning composite objects consisting of blocks (units: seconds of CPU-time and 1024 Bytes)

Chapter 6

Maintaining Geometric Dependencies

The previous two chapters described algorithms to efficiently find all the ways to partition an assembly into two subassemblies that can be mated in a single operation. However, if we are willing to restrict the type of assembly operations allowed in our sequences further, it is possible to gain even greater planning efficiency. In this chapter, assembly sequences are restricted to be linear, so that every operation mates a single part with a subassembly. Under this restriction, the generate-and-test paradigm works well, since for an assembly of n parts, only n candidate operations (one removing each part) are considered for each subassembly to be partitioned.

Within the context of generating linear assembly sequences, I propose methods to extract extra information from each geometric calculation and save justifications for the results in a logical form called a *precedence expression*. Later geometric calculations can be avoided by evaluating whether the results are still valid in different assemblies from the ones where they were originally derived. Experimental results show large acceleration of assembly sequencing over the basic method for linear sequences.

6.1 Generating Linear Assembly Sequences

Generating only the linear assembly sequences for a product might be desirable in some situations. Adding a single part at a time could simplify factory layout or reduce the dexterity needed to place one subassembly into another. To quickly generate one assembly

```
Procedure DECOMPOSE(A)
    feasible-decompositions ← {};
    for each part P ∈ A
        A' ← A \ {P};
        if CONNECTED(A') and MOVABLE(P, A)
            push(({P}, A'), feasible-decompositions);
    end; {for}
    return(feasible-decompositions);
end; {procedure}
```

Figure 6.1: Procedure *DECOMPOSE*

tree, each subassembly could be tested for single-part removal first, only proceeding to the more time-consuming general case if no single-part operation is feasible. In addition, Baldwin [5] found that first computing the linear assembly sequences for a product was a useful heuristic to better order questions to a human designer; a parallel might be found in automated sequencing.

To find single parts that are removable from an assembly, the methods of the previous two sections could be applied, and only single-part subassemblies identified. A part P is removable in direction d if and only if in the directional blocking graph $G(d, A)$, P has no outgoing arcs. The approach used in this chapter is more efficient in practice for linear sequencing; the two approaches will be compared in section 6.5.

To generate an AND/OR graph representing all the feasible linear assembly sequences for an assembly A, procedure *DECOMPOSE* from figure 3.8 is changed as shown in figure 6.1. Each part in A is simply tested for removability from the rest of the parts using procedure *MOVABLE*, which is similar to *SEPARABLE*. To comprise a valid subassembly, the complement A' must be exactly one connected component. A basic version of *MOVABLE* is shown in figure 6.2. Since now the algorithm just tests one assembly task for each part in A, it can operate much more quickly. The AND/OR graph created is a subset of the non-linear AND/OR graph.

When the linear assembly sequences are generated for the 22-part electric bell (figure B.3) with 6D cones and path planning turned off, the resulting AND/OR graph has 1509 nodes and 6190 edges, and takes 15 minutes to build. 12,169 calls to procedure *MOVABLE* are performed. This chapter describes ways to achieve the same set of sequences with

Procedure $MOVABLE(P, A)$
    return($SEPARABLE(\{P\}, A \setminus \{P\})$);
end; {procedure}

Figure 6.2: Procedure *MOVABLE*

a radically smaller amount of reasoning, and correspondingly lower sequencing times.

6.2 Maintaining Movability Dependencies

The obvious algorithm given above for generating the linear AND/OR graph just checks the movability of each part at each node in the AND/OR graph. However, this repeats a great deal of computation about the geometry of the assemblies. Because there is little change in geometry between assemblies and their children in the AND/OR graph, most of the geometric reasoning about the movability of parts in the parent assembly should still be valid for the same parts in each child subassembly. For instance, in the crate assembly of figure 1.1, `screw2` is movable whether `screw1` is present or not, while the `cargo` is not movable as long as the `box` and the `lid` are there, independent of `screw1` and `screw2`. Essentially, we would like to exploit similarities in the geometry of assemblies and their subassemblies to reduce the geometric computation necessary to plan assembly sequences.

6.2.1 Precedence Expressions

In the algorithm above there is a very weak link between the geometric reasoning modules and the symbolic reasoner constructing the AND/OR graph. For each query about the movability of a part in an assembly, *MOVABLE* replies only that yes, the part is movable, or no, it is not. We can significantly reduce the number of geometric reasoning steps by having the geometric module return an expression stating the conditions under which the given part would be movable, called a *precedence expression* or PE. When this PE is still valid in other assemblies in the AND/OR graph, evaluating it should be much faster than performing a full geometric check.

In general, a PE cannot fully describe the conditions of movability for a part. It only gives some sufficient and some necessary conditions. In order to manage this easily, I use a direct extension of the classical propositional calculus where each expression can take

OR	T	?	F
T	T	T	T
?	T	?	?
F	T	?	F

AND	T	?	F
T	T	?	F
?	?	?	F
F	F	F	F

NOT	
T	F
?	?
F	T

Table 6.1: Truth tables for GRASP's three-valued propositional calculus

one of three values: T, F, and **?** (*true*, *false*, and *maybe*). PEs are constructed from atomic propositions P_i, each of which represents the assertion that part p_i is present in the assembly under consideration. These P_i, along with the reified value **?**, are connected with the standard logical connectors to make precedence expressions. The truth tables of this calculus are given in table 6.1. I will use $M(p, S)$ to denote the movability of a part p in an assembly S, and will write E_p to mean a PE for the movability of p.

Definition 6.1 *A* precedence expression *is defined recursively as follows:*

- *Any atomic proposition P_i is a PE. The constant* **?** *is a PE.*
- *If E is a PE, then $\neg E$ is a PE.*
- *If E_1 and E_2 are PEs, then the expressions $(E_1 \wedge E_2)$ and $(E_1 \vee E_2)$ are PEs.*

6.2.2 Expressing Part Movability

Consider the movability of the `cargo` in figure 1.1. After calculating that the `box` allows the cargo only to move left, where it collides with the `lid`, the geometric reasoner might construct the PE

$$E_{cargo} = (\mathbf{?} \wedge \neg P_{box}) \vee \neg P_{lid}.$$

When the `box` and `lid` are both present, E_{cargo} evaluates to F, so we conclude that the `cargo` is not movable. Whenever the `lid` is missing, E_{cargo} evaluates to T, signaling that the `cargo` can be removed to the left. Finally, if the `lid` is present but the `box` is not, E_{cargo} evaluates to **?**, and so E_{cargo} does not give a definitive answer on the removal of the cargo in that assembly.

In general, a PE can express the movability of a part in an assembly that was not considered in deriving the PE. In an AND/OR graph for an assembly A, each subassembly $S \subseteq A$ is equivalent to a truth assignment from the atomic propositions P_i to the values

$\mathcal{N}$	$\mathcal{S}$	$(? \wedge \mathcal{N}) \vee \mathcal{S}$
T	T	T
T	F	?
F	T	T
F	F	F

Table 6.2: Necessary and sufficient conditions represented in a single formula.

$\{\mathrm{T}, \mathrm{F}\}$, where

$$P_i = \begin{cases} \mathrm{T} & \text{when } i \in S \\ \mathrm{F} & \text{when } i \notin S \end{cases}$$

To *evaluate* a PE *in an assembly* S means to compute the truth value of the PE under the truth assignment S, using the truth tables in figure 6.1. I will write $E_p(S)$ to indicate evaluation of E_p in assembly S.

The main property that I use to construct PEs is the following. Let $(N_i)_{i \in I}$ be necessary conditions for the movability of part p, and let $(S_j)_{j \in J}$ be sufficient conditions for its movability. Then the expression

$$E_p = (? \wedge \bigwedge_{i \in I} N_i) \vee \bigvee_{j \in J} S_j \tag{6.1}$$

will be T whenever one of the S_j is true, F whenever one of the N_i is false, and **?** in all other cases (figure 6.2). When no necessary or sufficient conditions are present, the necessary ($\mathcal{N}$) and sufficient ($\mathcal{S}$) conditions are set respectively to T and F. Furthermore, only proven necessary and sufficient conditions are used; thus the third line of table 6.2, which should return a contradiction, will never arise. Expression 6.1 is the standard form of all the types of PEs that I use below.

6.2.3 Using PEs in Sequencing

The basic idea of geometric dependency maintenance is to keep a set of PEs characterizing the movability of each part in different assemblies. When a PE evaluates to T or F in the current assembly, no geometric reasoning is necessary. When all the PEs for a part evaluate to **?**, geometric reasoning must be performed to generate a PE to cover the current assembly. At this point, there are two issues to address:

1. A large number of PEs might be generated during planning. How will they be organized to quickly find one that applies in a given assembly?

2. How much extra work should be done to make each PE applicable in other assemblies than the immediate one?

I have developed and tested two approaches to answering these questions, called *local* and *global* precedence expressions.

With local, or *inherited* precedence expressions, only one PE is kept for each part, and the PEs are inherited by subassemblies from their parents. No extra reasoning is required to construct them. Using global precedence expressions, all PEs applying to a part are kept in a list which might grow quite long, but they are valid throughout the AND/OR graph. A small amount of extra geometric reasoning is required to ensure their validity in non-subassemblies of the current assembly. Local PEs are described in the next section and global PEs are discussed in section 6.4.

Precedence expressions are closely related to the common sense rules used by Hoffman to reduce computation in the assembly sequencer BRAEN [33]. BRAEN generates non-monotone assembly sequences for curved shapes, a task which requires large amounts of geometric calculation. Rules such as the following are used:

- If part P can translate a distance x in direction d, and P is moved y units along d, then P can now translate $x - y$ in direction d.
- The freedom of part P is unchanged when parts that do not interfere with the motion of P are moved such that they still do not interfere with its motion.

These simple rules save an order of magnitude in computation. Since GRASP is limited to monotone sequences, PEs need only express whether a part is removable or not, instead of how far it can be moved. However, because the assemblies considered are much more complicated than those in [33], more attention must be paid to how PEs are constructed and organized, to ensure maximum use of geometric results.

6.3 Local Precedence Expressions

Passing down movability properties from an assembly to its subassemblies can be implemented by passing down precedence expressions. To prove that the PE E_p, inherited by a subassembly $S \subset A$, still denotes movability in S, we only need implications of the form:

$$\begin{aligned} \forall S \subseteq A, \qquad & \mathcal{S} \Rightarrow E_p(S) \\ & \neg\mathcal{N} \Rightarrow \neg E_p(S) \end{aligned} \tag{6.2}$$

```
Procedure MOVABLE(p, S)
    A ← A-PARENT-OF(S);
    E_p^S ← E_p^A;
    if E_p^S(S) evaluates to
        T: return(T);
        F: return(F);
        ?: E_p^S ← LOCAL-PE(p, S);
           if E_p^S(S) evaluates to T
                return(T);
           else return(F);
    end; {if}
end; {procedure}
```

Figure 6.3: Procedure *MOVABLE*, using local precedence expressions

where $\mathcal{N}$ and $\mathcal{S}$ are necessary and sufficient conditions on the movability of p in A. Several types of expressions of this form are given in the next section.

The *MOVABLE* procedure of figure 6.2 is replaced by a more sophisticated version given in figure 6.3. *MOVABLE* still must return T if a part p is removable from an assembly S and F if it is not. The new version also has the side effect of setting the PE of a part. The PEs of all parts are **?** in the initial assembly node. There is one PE associated with every part in each node of the AND/OR graph; E_p^S denotes the PE for movability of part p stored in assembly S. Real geometric computation only occurs in the *LOCAL-PE* procedure, which is called when the movability of the part cannot be deduced symbolically from the inherited PE. *LOCAL-PE* mirrors the old *MOVABLE* procedure, but instead of just returning T or F, it constructs a PE describing the movability of p in the assembly and its subassemblies. Local PEs are only evaluated in subassemblies of the assembly for which they were created, and their constructions below take advantage of this fact.

Notice that in general, each node in the AND/OR graph has many parents, and so the choice of parent assembly from which to inherit is arbitrary. It would be possible to combine the PEs from the different parents for the child node's expression, sometimes saving more geometric computation than with the single-inheritance method. However, it is not clear that the savings would outweigh the extra overhead and complexity of combining PEs.

I have developed and tested three kinds of local PEs, called *simple*, *contact*, and

descriptive[1] precedence expressions. They are increasingly complex and accurate in describing symbolically when a part is movable.

6.3.1 A Simple Sufficient Condition

When parts are removed from an assembly of rigid parts A, the free space for the remaining parts is increased. Thus if a part is movable in A it is guaranteed to be movable in any subassembly S of A:

$$\forall S \subseteq A,\ \forall p \in S, \quad M(p, A) \Rightarrow M(p, S) \tag{6.3}$$

The simple type of PE takes advantage of this fact. Simple PEs are similar to the *subset rule* of Bourjault and others [14, 5]. Homem de Mello [34, page 168] also mentions the possibility of performing a check similar to the one that simple PEs achieve, but does not elaborate.

From relation 6.3, whenever a part p is movable in an assembly A, T is a sufficient condition for $M(p, S)$. Therefore, from expression 6.1, the local PE is set trivially to T. When subassemblies inherit their PEs from A, geometric computation will not have to be done for parts that were movable in A. On the other hand, the movability calculation will need to be redone in subassemblies of A for each unmovable part. For instance, after expanding the root node in the crate assembly in figure 1.1, the simple precedence expression E_{screw2} will be T, and will not need to be recomputed in the subassembly with `screw1` removed.

6.3.2 A Necessary Condition on the Constraining Parts

In the next version, contact precedence expressions, the geometric module supplies the planner with a list of parts that constrain a part p in the assembly A. The set of constraining parts $\mathcal{C}(p, A)$ is the set of all parts p' in A such that

- p' is in contact with p, or
- in an extended motion allowed by the contacts, p collides with p', or
- p' is one of the parts returned by the path planner as constraining p.

[1] In previous work [65, 66] descriptive local PEs were just called local PEs. With the advent of global PEs, a new term was required.

See section 3.5 for a description of the interface to a path planner. In subassemblies of A, movability does not need to be recomputed when all of these parts are still in the subassembly:

$$\forall S \subseteq A,\ \forall p \in S, \quad \neg M(p, A) \wedge (\mathcal{C}(p, A) \subseteq S) \ \Rightarrow\ \neg M(p, S) \tag{6.4}$$

From relations 6.2 and 6.4, we can infer that when p is not movable in A, $LOCAL\text{-}PE(p, A)$ must return an expression

$$E_p(A) = \mathbf{?} \wedge \neg \bigwedge_{q \in \mathcal{C}(p,A)} P_q.$$

When inherited by a subassembly $S \subset A$, E_p will evaluate to F as long as all of the original constraining parts $\mathcal{C}(p, A)$ are present in S. When one of them is removed, the truth value of a P_q will become F, causing $E_p(S)$ to evaluate to **?**. Geometric computation will then have to be done for p in S.

In addition, $LOCAL\text{-}PE(p, A)$ returns $E_p = \mathrm{T}$ when p is movable, so the simple sufficient condition of the previous section is maintained.

For instance, after expanding the root node A in the crate assembly, the contact precedence expression for the `cargo` in A will be

$$E_{cargo} = \mathbf{?} \wedge \neg(P_{box} \wedge P_{lid})$$

because the `cargo` is constrained to move left by the `box`, and sweeps into the `lid` in that direction. In subassemblies resulting from removing either or both screws, the `cargo` will still be unmovable but no geometric calculation will be done.

6.3.3 Necessary and Sufficient Conditions

In the final and most complicated version, descriptive precedence expressions, the geometric reasoner returns a local PE even more closely stating the conditions under which a part might be movable. The parts in contact with p in A are grouped such that all the parts in a group constrain the local freedom of p in the same way, either along the same plane or in parallel cylindrical contacts. Moreover, the parts swept into along one direction are also grouped together. In subassemblies of A, p will not be movable unless all of the parts in one such group are missing. Furthermore, if all swept-into parts in one direction are missing, a sweep in that direction must be collision-free, and so p is guaranteed to be movable. Finally,

the constraining parts returned by the path planner are grouped together, and if all of these are present, p cannot be movable:

$$\begin{aligned} & \forall S \subseteq A, \forall p \in S, \\ \neg M(p,A) \quad & \wedge \left[\textstyle\bigwedge_{f \in \mathcal{F}(p,A)} \bigvee_{q \in f} q \in S\right] \\ & \wedge \left[\textstyle\bigwedge_{d \in \mathcal{D}(p,A)} \bigvee_{r \in d} r \in S\right] \\ & \wedge \mathcal{P}(p,A) \subseteq S \qquad \Rightarrow \neg M(p,S) \qquad (6.5) \\ & \textstyle\bigvee_{d \in \mathcal{D}(p,A)} \neg \bigvee_{s \in d} s \in S \quad \Rightarrow M(p,S) \qquad (6.6) \end{aligned}$$

$\mathcal{F}(p,A)$ is a set indexed by facets of the local freedom cone of p in A, where f is the set of parts constraining a given facet; $\mathcal{D}(p,A)$ is a set indexed by directions of sweep, d being the set of parts swept into when p moves along one given direction; and $\mathcal{P}(p,A)$ is the set of parts returned by *PATH-PLAN* as constraining p in A.

Thus when a part is not movable, we have one necessary (6.5) and one sufficient (6.6) condition, so the full expression 6.1 applies and $LOCAL\text{-}PE(p,A)$ should return an expression

$$E_p = \left[? \wedge \neg \left(\bigwedge_{f \in \mathcal{F}(p,A)} \bigvee_{q \in f} P_q \wedge \bigwedge_{d \in \mathcal{D}(p,A)} \bigvee_{r \in d} P_r \wedge \bigwedge_{t \in \mathcal{P}(p,A)} P_t \right) \right] \vee \neg \bigwedge_{d \in \mathcal{D}(p,A)} \bigvee_{s \in d} P_s$$

In fact, $LOCAL\text{-}PE(p,A)$ returns an equivalent simplified expression

$$E_p = \left[? \wedge \neg \left(\bigwedge_{f \in \mathcal{F}(p,A)} \bigvee_{q \in f} P_q \wedge \bigwedge_{t \in \mathcal{P}(p,A)} P_t \right) \right] \vee \neg \bigwedge_{d \in \mathcal{D}(p,A)} \bigvee_{s \in d} P_s$$

In addition, when p is movable in A, *LOCAL-PE* returns the simple local PE T.

For example, without calling the path planner, the descriptive local PE for the `cargo` after the expansion of the root node A will be[2]

$$E_{cargo} = [? \wedge \neg P_{box}] \vee \neg P_{lid}$$

and the local precedence expression for the `lid` after expanding A will be

$$E_{lid} = ? \wedge \neg[(P_{screw1} \vee P_{screw2}) \wedge P_{box}]$$

Since the `lid` is completely constrained by contacts, no sweep term is included in the descriptive PE. Notice that using contact PEs, GRASP would recompute the movability of

[2] Actually, GRASP does not simplify its precedence expressions, and so P_{box} is listed three times because the `box` contributes three facets to the local freedom cone of the `lid`.

the `lid` after removing one screw; this is avoided with descriptive PEs. Section 6.6 presents theoretical and experimental evaluation of the different types of local PEs.

All three varieties of local PEs given above can be constructed with little additional geometric calculation. The parts in contact with the moving part must be calculated, and all parts must be checked for collision along each locally free direction tried. Grouping the parts according to their facets on the freedom cone can be accomplished while building the cone. Finally, a standard path planner might be modified to provide a set of constraining parts as discussed in section 3.5.

6.4 Global Precedence Expressions

The second type of precedence expressions I have tested are called *global* PEs. Rather than being inherited down the AND/OR graph from assembly to subassembly, global PEs apply throughout the AND/OR graph. A list of global PEs is kept, indexed by the part they apply to, called $PE(p)$. When procedure *MOVABLE* is called for a part p in an assembly S, all the PEs in $PE(p)$ must be evaluated. If a PE evaluates to T, it asserts that p is movable in S; if one evaluates to F, the sequencer concludes that p is not movable. Only when all the PEs for p evaluate to **?** in S is the geometric reasoner called to construct a new PE. The new PE for p does not replace any others, but is added to the list $PE(p)$. Obviously, if one PE evaluates to T and another to F for the same part in the same assembly, then one of the PEs was incorrect; the types of global PEs below are carefully designed to keep this from happening.

The *MOVABLE* procedure for global PEs is shown in figure 6.4. *MOVABLE* still must return T if a part p is removable from an assembly S and F if it is not. The array $PE(p)$ starts out empty for all p. Real geometric computation only occurs in the *GLOBAL-PE* procedure, which is called when the movability of the part cannot be deduced symbolically from any of the PEs in $PE(p)$. *GLOBAL-PE* is much like the procedure *LOCAL-PE* but it must do some extra reasoning to ensure global validity of the resulting PE.

To prove that a global precedence expression E_p, evaluated in an assembly S in the AND/OR graph with root assembly $\mathcal{A}$, correctly denotes movability of p in S, we need implications of the form:

$$\begin{aligned} \forall S \subseteq \mathcal{A}, \qquad & \mathcal{S} \Rightarrow E_p(S) \\ & \neg\mathcal{N} \Rightarrow \neg E_p(S) \end{aligned} \tag{6.7}$$

```
Procedure MOVABLE(p, S)
    for each E_p ∈ PE(p)
        if E_p(S) evaluates to
            T: return(T);
            F: return(F);
    end; {for}
    E_p ← GLOBAL-PE(p, S);
    push(E_p, PE(p));
    if E_p(S) evaluates to T
        return(T);
    else return(F);
end; {procedure}
```

Figure 6.4: Procedure *MOVABLE*, using global precedence expressions

where $\mathcal{N}$ and $\mathcal{S}$ are necessary and sufficient conditions on the movability of p in the root assembly $\mathcal{A}$. Note that whereas local PEs need only be valid in subassemblies of the node in which they were derived, global PEs must be valid in any assembly. Several types of expressions of this form are given below.

Since all the PEs generated for a part p are kept in a list $PE(p)$, the list might contain a very large number of PEs as sequencing progresses. In addition, at each node in the AND/OR graph all the expressions in $PE(p)$ must be evaluated to determine whether they are informative in the current assembly. If the length of $PE(p)$ becomes very large, evaluating all the PEs for p might take as long as the geometric computation itself, and possibly far longer. Several schemes are possible to attempt to prune the set of PEs for a part. A simple one is to record how often each PE proves "useful"—how often it evaluates to T or F—compared to the number of times it is evaluated. If a PE drops below a threshold level of usefulness, it could be removed from $PE(p)$. Another method might examine PEs to find whether one PE subsumes another. In experiments with real assemblies the number of PEs required to generate the entire AND/OR graph has been quite low, and so this possible problem has not materialized.

I have developed and tested three kinds of global PEs analogous to the three kinds of local PEs, called *simple*, *contact*, and *descriptive* global precedence expressions. Each bears a strong resemblance to the corresponding kind of local PE, but has slight differences due to the fact that local PEs implicitly can only be evaluated in subsets of the original assembly,

while global PEs must be valid throughout the AND/OR graph. For this reason, each kind of global PE must perform some calculations with respect to the root assembly $\mathcal{A}$ of the AND/OR graph.

6.4.1 A Simple Sufficient Condition

Like simple local PEs, a simple global PE represents a sufficient condition on the movability of a part p. Given any direction d, there is a set of parts $\mathcal{D}(p, d, \mathcal{A})$ constituting all the parts of $\mathcal{A}$ that prevent p from translating to infinity along d. In any assembly S that shares no parts with $\mathcal{D}(p, d, \mathcal{A})$, p is guaranteed to be removable in direction d:

$$\forall d, \forall S \subseteq \mathcal{A}, \forall p \in S, S \cap \mathcal{D}(p, d, \mathcal{A}) = \emptyset \Rightarrow M(p, S) \tag{6.8}$$

Furthermore, if p is removable along d in an assembly A, then the set of swept-into parts must be disjoint from A. Using $M(p, d, A)$ to denote the movability of p along direction d in assembly A,

$$\forall d, \forall A \subseteq \mathcal{A}, \forall p \in A, M(p, d, A) \Rightarrow A \cap \mathcal{D}(p, d, \mathcal{A}) = \emptyset \tag{6.9}$$

From equations 6.8 and 6.7, we can infer that when p is movable along direction d in S, $GLOBAL\text{-}PE(p, A)$ must return an expression

$$E_p = \mathbf{?} \vee \neg \bigvee_{q \in \mathcal{D}(p,d,\mathcal{A})} P_q$$

Equation 6.9 ensures that this PE will evaluate to T in A and that once p is found to be movable in A, the geometric reasoner need only sweep p against the parts in $\mathcal{A} \setminus A$ to construct E_p. Thus geometric computation will be avoided for p in all assemblies encountered later in which d is a valid removal direction. On the other hand, when p is fully constrained in A, E_p is set to **?**, and contributes nothing to future calculations. For example, the simple global PE for the `cargo` in the assembly {`cargo, box`} will be

$$\mathbf{?} \vee \neg P_{lid}.$$

If path planning is turned on and the path planner returns a valid path for p, then this path can be checked for collisions with the parts $\mathcal{A} \setminus A$ and a simple global PE constructed in the same way as for a sweep direction d. If a valid path is not returned by the path planner, as with the current interface to a human engineer, then a set of constraining parts cannot be computed, and a simple global PE cannot be built.

Bourjault [14] and Baldwin [5] use a *subset* rule and its contrapositive the *superset* rule to reduce the number of questions to the human. The subset rule states that if a subassembly S_1 can be removed from subassembly $S_2 = A \setminus S$, then S_1 can be removed from any subset of S_2. This resembles the effect of simple global precedence expressions. Contact global PEs (below) resemble the superset rule but are more discerning, because they identify a small subset of parts that constrain the part p. In [5, 14] the subset and superset rules are used in nonlinear sequencing. However, for large assemblies the cost of the generate-and-test cycle would make such use prohibitive (see Chapter 4).

6.4.2 A Necessary Condition on the Constraining Parts

Contact global PEs are a straightforward extension of contact local PEs. When a part p is not movable in assembly A, the geometric module supplies the sequencer with a list of constraining parts $\mathcal{C}(p, A)$. This list is exactly the same set of parts given in section 6.3.2 for contact local PEs. Since $\mathcal{C}(p, A)$ is sufficient to fully constrain p alone, any superset of $\mathcal{C}(p, A)$ will also constrain p:

$$\forall A, S \subseteq \mathcal{A}, \forall p \in A, \neg M(p, A) \wedge \mathcal{C}(p, A) \subseteq S \Rightarrow \neg M(p, S) \tag{6.10}$$

From equations 6.7 and 6.10, we can infer that when p is not movable in A, *GLOBAL-PE*(p, A) must return an expression

$$E_p = \mathbf{?} \wedge \neg \bigwedge_{q \in \mathcal{C}(p,A)} P_q.$$

In any assembly $S \subset \mathcal{A}$, E_p will evaluate to F as long as all of the original constraining parts $\mathcal{C}(p, A)$ are present in S. When one or more are missing, the truth value of a P_q will be F, causing $E_p(S)$ to evaluate to **?**. E_p will thus contribute no knowledge about the movability of p in S.

Note that when p is not movable, the local and global PEs are the same. However, when p is movable in A, the *GLOBAL-PE*(p, A) returns the simple global PE given above. Thus the simple sufficient condition of the previous section is maintained.

Contact global PEs are somewhat similar to the *superset* rule of Bourjault and others [14, 5], but limited to single parts and more accurate. The superset rule states that when a subassembly S_1 is not removable from another subassembly S_2, then S_1 is not removable from a superset $S_3 \supseteq S_2$. Contact global PEs are constructed from the geometrically

constraining subset of S_2, and thus will capture more cases S_3. By deriving the symbolic constraints directly from the geometry of the assemblies, a great deal of reasoning is saved in some situations.

6.4.3 Necessary and Sufficient Conditions

Descriptive global PEs are the most powerful and accurate type of PE I have tested. The groups of parts used in their creation are either the same or straightforward extensions of the groups for descriptive local PEs:

- As before, the parts in contact with a part p in an assembly A are grouped according to the facets $f \in \mathcal{F}(p, A)$ of the local freedom cone to which they contribute.
- The parts swept into along each attempted sweep direction d are grouped together. But instead of being swept against the parts of A, p is swept against all the parts in the root assembly $\mathcal{A}$ to form a list of directional blocking sets $\mathcal{D}(p, \mathcal{A})$. $\mathcal{D}$ is used to form a condition of global validity for each d.
- The constraining parts $\mathcal{P}(p, A)$ returned by the path planner are grouped together as with local PEs.

In other assemblies S in the AND/OR graph, the presence of subsets of $\mathcal{F}, \mathcal{D}, \mathcal{P}$ become sufficient and necessary conditions for the movability of p:

$$\begin{aligned} & \forall A, S \subseteq \mathcal{A}, \forall p \in A, \\ \neg M(p, A) \quad & \wedge \left[\textstyle\bigwedge_{f \in \mathcal{F}(p,A)} \bigvee_{q \in f} q \in S \right] \\ & \wedge \left[\textstyle\bigwedge_{d \in \mathcal{D}(p,\mathcal{A})} \bigvee_{r \in d} r \in S \right] \\ & \wedge \mathcal{P}(p, A) \subseteq S \qquad \Rightarrow \neg M(p, S) \qquad (6.11) \\ & \textstyle\bigvee_{d \in \mathcal{D}(p,\mathcal{A})} \neg \bigvee_{s \in d} s \in S \quad \Rightarrow M(p, S) \qquad (6.12) \end{aligned}$$

When a part p is not movable, the necessary (6.11) and sufficient (6.12) conditions can be combined in the form of expression 6.1:

$$E_p = \left[? \wedge \neg \left(\bigwedge_{f \in \mathcal{F}(p,A)} \bigvee_{q \in f} P_q \wedge \bigwedge_{t \in \mathcal{P}(p,A)} P_t \right) \right] \vee \neg \bigwedge_{d \in \mathcal{D}(p,\mathcal{A})} \bigvee_{s \in d} P_s$$

When p is movable in A, a simple global PE is returned.

In almost all cases, global PEs require extra geometric computation—sweeping p against parts not in the assembly being considered—to construct PEs that are valid anywhere in the AND/OR graph. Some of these part-part sweeps would never be checked in building the AND/OR graph, were global PEs not in use. In experiments the other savings in computation have overshadowed this effect (see below).

6.5 Nonlinear Sequencing

So far in this chapter, only linear assembly sequences have been considered. However, precedence expressions could in theory be applied to nonlinear sequencing. For instance, a subassembly can be considered as a part for the purpose of movability, as long as it remains stable throughout the removal motion. From this observation, a straightforward extension to the non-linear case was outlined in [66]: instead of keeping a PE E_p for the movability of each part p in the assembly A, the planner would maintain expressions E_S for each subassembly S of A. When E_S evaluates to T or F in a node A', then S is removable or not removable, respectively. The E_S could still be constructed from elementary propositions P_i asserting the presence of parts i. Thus, after expanding the root node in the crate assembly of figure 1.1, the precedence expression for the subassembly {`box,cargo,screw1`} might be:

$$E_{\{box,cargo,screw1\}} = \mathbf{?} \wedge \neg[(P_{lid} \vee P_{screw2}) \wedge P_{lid}] \tag{6.13}$$

However, in spite of the theoretical validity of maintaining dependencies for subassemblies, this method is impractical. As noted in section 4.1, an exponential number of subassemblies might be candidates for removal at any one node in the AND/OR graph; keeping PEs for all of these is unmanageable. The use of precedence expressions as above depends strongly on linear assembly sequences being generated. Thus the partitioning methods of Chapters 4 and 5 are far better suited when nonlinear sequences are desired.

On the other hand, the NDBG encodes more information than is necessary when linear sequences are desired. This is because a new cell is created in the NDBG for every possible change in the blocking relationships between *subassemblies* of the product, instead of single parts. For instance, consider the translational contact NDBG for an assembly A of n parts, each having c contact constraints with other parts, for $k = cn$ total constraints. If no two constraints generate the same great circle in the arrangement, $\Omega(n^2c^2)$ cells will be created. To check the local freedom of each part in each cell will then require $\Omega(n^3c^2)$ time. On

the other hand, testing all parts individually using the local freedom methods of Chapter 3 requires only $O(nc \log c)$ time.

Precedence expressions accelerate linear sequencing even more. For instance, *PARTITION* generates the full AND/OR graph of nonlinear assembly sequences for the 22-part electric bell in 2186 seconds; using global PEs, the set of linear sequences is found in 51 seconds (see below). The respective count of AND-arcs in the two graphs is 21,315 and 6,190, so the numbers are not directly comparable, but precedence expressions clearly work faster when linear sequences are desired.

6.6 Theoretical Complexity

Precedence expressions were introduced to enrich the communication between the assembly sequencer and the geometric reasoning module, in order to speed up the process of assembly sequencing. This section presents a theoretical analysis of local PEs for certain simple types of assemblies. I have not been able to characterize the types of assemblies in which global PEs do better than local PEs; the results in this section also apply to them. I assume that the number of calls GC to the geometric reasoner is the overriding factor for the total running time of the algorithm to generate the AND/OR graph. This assumption is not fully borne out by the experiments in the next section, but for the simple assemblies considered here it is a reasonable assumption.

Because of the complex ways in which the geometry of an assembly can affect the size of its AND/OR graph, it is difficult to find meaningful bounds on the computation required to build it. For instance, given an assembly with N parts, the number of nodes in the AND/OR graph can range from $2N - 1$ when there is only one legal sequence, to $2^N - 1$ when all sequences are legal. Below we analyze the complexity for three types of assemblies and for each type of local precedence expressions.

Consider the situation in which all parts are free to move in the initial assembly, but only one sequence satisfies stability considerations, as in Figure 6.5a. With N parts, the AND/OR graph has $N - 1$ non-terminal nodes. The time required to generate the graph using each type of precedence expression is:

None At each step in the generation of the AND/OR graph, all of the parts in the subassembly being considered must be checked for movability. Therefore, without precedence expressions $GC = \sum_{i=0}^{N-1} N - i = \frac{N(N+1)}{2} = O(N^2)$

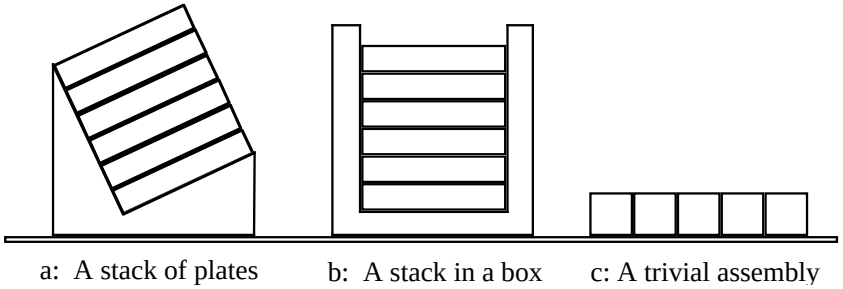

Figure 6.5: Three simple types of assemblies

Simple Using simple precedence expressions, the accessibility will be computed for the original N parts, finding each expression to be *true*. These expressions will be inherited downward, and no more geometric reasoning will be necessary. Thus $GC = O(N)$.

Contact and Local The complexity is the same as in the simple case.

In assemblies like the one in Figure 6.5b with $N-1$ plates inside a box, only one sequence is valid because just one part is removable in each subassembly. Again the complexity depends on the type of precedence used:

None The obvious algorithm will again require $GC = O(N^2)$ calls to the geometric reasoner.

Simple In this case simple precedence gains us nothing. In each node, only one part is movable, so no *true* precedence expressions will be inherited. As a result, $GC = O(N^2)$.

Contact Since each plate p_i is constrained by the box and parts p_{i-1} and p_{i+1}, when we remove part p_{i+1} only the contact precedence expressions of the box and part p_i will evaluate to *maybe*, forcing a geometric call. Thus the number of calls will be 2 at each step except the last where only the box remains, so $GC = N + 2N - 1 = O(N)$.

Local The local precedence expressions for the plates will result in the same behavior as in the contact case. However, since the last plate p_{N-1} contributes to each of the constraints on the box, the box's precedence expression will not evaluate to *maybe* until p_{N-1} is removed, so $GC = N + N - 1 = O(N)$.

Finally, consider an assembly in which all sequences of assembly are valid, such as in Figure 6.5c. Without precedence expressions, the accessibility of every part in each of the

Local Precedence	Geometric Calls	Time in Seconds
none	14	9.0
simple	12	8.8
contact	9	8.5
local	6	8.9

Table 6.3: Planning times for the crate assembly

$2^N - 1$ nodes would be computed. Using simple (or any other) precedence expressions, the accessibility would be found to be *true* for each part in the final assembly. This information would be inherited down the tree, making the total number of geometric calls N, even though there are $2^N - 1$ nodes in the AND/OR graph.

6.7 Experiments

Precedence expressions of different types have been used to plan the assembly sequences for a variety of assemblies, to evaluate their utility in accelerating the sequencing process in real assemblies. The assemblies in this section are described in Appendix B.

6.7.1 2D Assemblies

The following 2D assemblies were planned for using the 2D prototype of GRASP. Neither global PEs nor path planning were implemented in the prototype, so experimental data is not available for these techniques on 2D assemblies. In addition, sweep caching (section 3.4) was not included in the 2D prototype.

Table 6.3 shows the number of geometric calls required for the prototype of GRASP to generate the full AND/OR graph for the crate assembly in figure 1.1 using each kind of local PE. The dedicated reader can check it by hand to help understand the method. Note that the time to generate the graph is greater using descriptive local PEs than with contact PEs; the geometry is so simple that the time required to create complex precedence expressions outweighs the savings.

A more interesting example is the transmission with which De Fazio and Whitney [25] illustrate their method for generating assembly sequences. Figure 6.6 shows the linear assembly sequences for the transmission without bolts, represented as a state graph. The unassembled state is not shown. Assemblies are shown by filling the box corresponding to

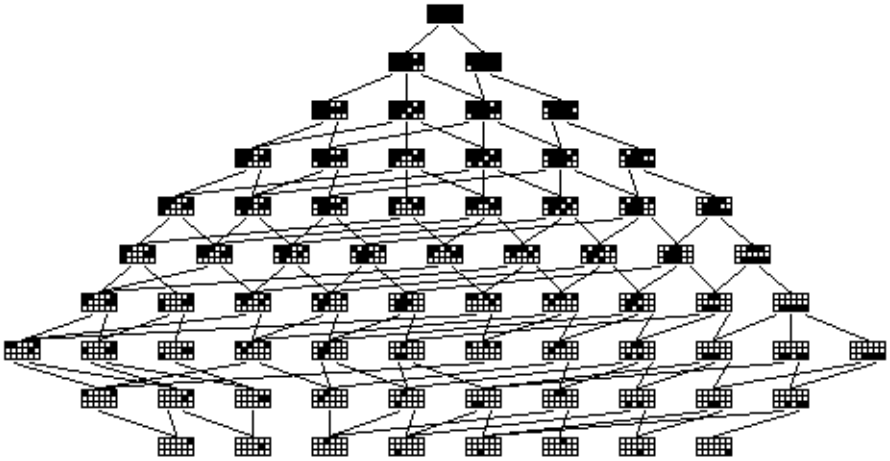

Figure 6.6: Assembly sequences for the transmission

Local Precedence	Geometric Calls	Time in Seconds
none	2508	1151
simple	2035	943
contact	669	445
local	121	99

Table 6.4: Planning times for the transmission, with bolts

each liaison that has been established in that assembly (liaisons 1-6 in the first row, etc.). For example, the leftmost assembly in the third row down has all connections established except for 4, 5, 16, 17, and 18; this corresponds to the assembly with all parts except K and L.

The set of sequences shown in figure 6.6 is not quite the same as the ones given in [25]. These differences are a result of GRASP generating its AND/OR graph from geometry alone, while De Fazio and Whitney compute their sequences from *precedence constraints* incorporating human geometric and mechanical insight. For example, they find six possible ways to start the assembly process; GRASP finds eight (the bottom row of figure 6.6). One of these, the assembly consisting of parts `C` and `D`, cannot result in a finished assembly because the bolts connecting `C` to `A` are not accessible when `C` and `D` are connected. Because the bolts are not represented explicitly, GRASP cannot take this into account. However, when GRASP is run on the full model including bolts, it does not find any sequences using the assembly of only `C` and `D`.

Precedence Type		One Sequence		AND/OR Graph	
		GC	Time	GC	Time
none		189	28	12169	878
local	simple	150	21	6757	458
	contact	48	15	950	98
	descr	32	15	718	83
global	simple	150	22	6002	490
	contact	48	22	49	63
	descr	32	22	34	51

Table 6.5: Planning times for the 22-part electric bell

The number of geometric calls and the time required for GRASP to generate the AND/OR graph for the transmission, with bolts as separate parts, is shown in table 6.4. The resulting AND/OR graph has 295 subassembly nodes and 668 AND-arcs.

6.7.2 3D Assemblies

Due to the more complicated reasoning required in three dimensions, PEs are more important than in 2D to avoid needless recalculations. In the following experiments, GRASP was run in a fully autonomous mode, with 6D local freedom checking and path planning turned off. Sweep caching was enabled.

The AND/OR graph representing the linear sequences of assembly for the electric bell of figure B.3 has 1509 nodes and 6190 AND-arcs. Table 6.5 shows the number of geometric calls, the sweeping calculations, and the time required to generate an AND/OR graph for the bell assembly, using both local and global PEs[3].

Several points should be noted about table 6.5. First, precedence expressions do not save much time when only one sequence is found, since the other costs outweigh the geometric calculation time. In addition, global PEs take more time to generate a single sequence than local PEs, due to the extra geometric calculation to construct them.

Note that using global descriptive PEs, about 1.5 calls to the geometric reasoner were necessary for each part in the bell, on average. This means that for each part, one or two PEs were sufficient to characterize its movability in all 1509 subassemblies that occur in linear assembly sequences for the bell. However, the added sweeping and the overhead time of building the graph made the overall time savings of global over local PEs somewhat

[3]These figures differ from those in [66] because threaded contacts are considered and an error was corrected in the model of the bell.

Precedence Type	Answers	Geometric Calls	User Queries	Time in Minutes
none	Yes/No	—	> 1200	> 60
local	Yes/No	—	> 1200	> 60
	Constrain	841	50	10
global	Yes/No	298	263	18
	Constrain	40	6	3

Table 6.6: Path planning experiments with the electric bell

smaller.

6.7.3 3D Assemblies with Path Planning

When the path planning module is enabled, precedence expressions become critical. An automated path planner might take anywhere from a minute to several hours to find a path for a part, depending on the path planning method employed. When an engineer is answering path planning questions, each might be answered in a few seconds. However, the user will never use the assembly sequencer if it requires answering hundreds of questions.

Table 6.6 shows the results of experiments on the 22-part electric bell when path planning is enabled. The resulting AND/OR graph has 1747 nodes and 6965 edges. The following five planning methods were tried:

- Using no PEs, each path planning query was answered with Movable or Not Movable—yes, the part can be removed, or no, it cannot. This is the most straightforward approach to generating the graph.
- Using descriptive local PEs, yes/no answers were given as before.
- Using descriptive local PEs, the queries were answered by identifying a constraining set of parts when the part cannot be removed (as described in section 3.5).
- Using descriptive global PEs, each query was answered with a yes/no answer.
- Using descriptive global PEs, constraining sets were identified.

In the first two cases, over 1200 queries were answered over the course of an hour, generating only a small fraction of the graph, after which the experiments were stopped. Table 6.6 shows the results for the other three cases.

As argued in section 3.5, an automatic path planner could be modified to produce the list of constraining parts with little extra computation. Since global PEs extract the most from each geometric calculation, the number of path planning calls will be minimized. For instance, to generate the full AND/OR graph for the 22-part bell requires only 6 path planning calls using global PEs.

The values in table 6.6 compare very favorably with the results of other papers in interactive assembly sequencing:

- Bourjault [14] and Baldwin [5] require yes/no answers to user queries. To reduce the number of queries, they invoke subset and superset rules that have the same effect as global contact PEs with yes/no answers. GRASP answers more questions automatically than either of those systems, yet the query count is still quite high for large assemblies.

- DeFazio and Whitney [25] ask only $2m$ questions for an assembly with m liaisons. However, the answer to each question is a logical expression defining *all* situations in which each liaison can be established. These expressions can be very complex, and it is difficult for a human to answer them accurately.

- Using global PEs with a list of constraining parts is a powerful compromise between the previous two approaches. A great deal of information is extracted from each answer. However, the answers are easy to give and accurate, since they only require the user to reason about the constraining parts in a single situation. As seen in table 6.6, this results in a very low query count, while maintaining simplicity and accuracy.

Table 6.7 shows the number of user queries required to find a single assembly sequence for several additional assemblies. For these assemblies, generating a full AND/OR graph is not feasible. The sequences were generated using no PEs, global PEs with yes/no answers, and global PEs with constraining sets identified.

Assembly	Number of Parts	No PEs	Yes/No	Constrain
Bell	22	37	37	4
Engine	30	85	85	5
Skin	36	32	32	4
Engine	42	173	144	7

Table 6.7: User query count to find a single sequence using PEs

Chapter 7

Conclusion

In a concurrent engineering system, an automated assembly planner would be invaluable to give immediate manufacturing feedback. Features that make a product impossible or difficult to assemble could be identified when they are introduced into the design. The designer could ask "what if" questions and quickly evaluate the impact of design changes on the assembly process. As a result, the designer would be relieved of the tedium of current methods to assess the assemblability of the product, while receiving manufacturing feedback early in the design process, when it can have the most impact.

7.1 Geometric Assembly Sequencing

Because it identifies constraints on assembly plans resulting strictly from the design of the product, assembly sequencing can give design-for-assembly feedback early in the design process, when a specific manufacturing scheme has not yet been chosen. However, designers will not tolerate a design tool that requires them to add tedious details to a product model, that gives spurious assembly plans, or that only yields results after days of processing. Therefore, to serve as an interactive design tool, an assembly sequencer will need to be autonomous, capable, and fast. This thesis presents significant progress toward realizing such an assembly sequencer.

A basic approach to assembly sequencing directly from geometric models of a product was described. The approach includes practical methods to assess the geometric feasibility of assembly operations; these methods are fast in practice yet find the great majority of assembly operations in real products. GRASP, the implementation of this basic approach,

successfully planned for the assembly of a number of real products, and served as a valuable testbed in which to evaluate the more advanced techniques presented.

A new data structure, the non-directional blocking graph, was introduced to represent the blocking relationships between parts in an assembly. The NDBG is the basis of several polynomial-time methods to efficiently find subassemblies that are removable from the product under varying constraints on motions. The NDBG is incremental in two ways:

- It can be constructed in an incremental fashion, using *crossing rules* to compute the blocking relationships in neighboring cells. In fact, the whole structure need not be stored at once; each cell can be discarded after traversing it, to minimize storage requirements.

- On the other hand, the NDBG for a product can be built incrementally as the product is designed, updating it to account for design changes. When a design change is small, the change to the NDBG can be computed much more quickly than it can be built from scratch. Using this incremental construction, a sequencer can respond to user queries about the assembly process much more quickly.

When applied to the case of general rigid motions, the algorithm efficiently computes the set of subassemblies that satisfy a powerful, necessary constraint on binary assembly operations with rigid parts.

Finally, a method was described wherein the results of geometric calculations are saved in *precedence expressions*, which are evaluated to answer similar geometric queries later in the sequencing process. The method results in greatly reduced geometric computation and a corresponding acceleration in sequencing. When used to represent the constraints on part motion identified by a path planner, precedence expressions reduce by several orders of magnitude the number of path planning problems that are necessary to complete assembly sequencing.

7.2 Representing Geometric Assembly Constraints

The choice of sequence representation is a crucial decision in designing an assembly sequencing system. This thesis provides several observations about practical methods to represent sets of geometrically valid assembly sequences.

7.2.1 AND/OR Graphs

The AND/OR graph is a powerful explicit representation for assembly sequences of small assemblies, and it is a useful formalization of the space of possible sequences. However, it is not a practical representation for assembly sequences of complex products with many parts. For instance, in Chapter 6 the AND/OR graph of linear sequences could not be generated for experimental assemblies of 30 or more parts.

Several techniques exist to extend the usefulness of explicit AND/OR graphs to somewhat larger assemblies; however, none can completely overcome the combinatorics of representing so many subassemblies. For instance, the implicit AND/OR graph of sequences might be searched for an optimal AND-tree according to some evaluation function, using a search algorithm such as AO* and an appropriate heuristic [49]. Additional AND-trees could be produced as needed. The heuristic would have to be quite powerful to avoid generating a large subset of the AND/OR graph, and such heuristics are usually difficult to create. Fasteners may be not represented as individual parts in the assembly to reduce the part count, as in [34]; however, the AND/OR graph will still grow quickly as the "real" part count increases. Finally, parts and fasteners can be clustered according to heuristics or user directives into preferred subassemblies, orders of part insertion, and so on, to reduce the size of the search space [10]. With such clusters, one can no longer guarantee that a sequence will be found if one exists. Combining the above techniques might yield an AND/OR-based sequencer that could handle some assemblies of 50 parts, but it is clear that more concise representations of assembly sequences must be found.

7.2.2 Implicit Representations

Homem de Mello [34] describes several implicit representations of assembly sequences. An implicit representation (briefly discussed in section 2.5) consists of a set of rules restricting the operations in a sequence. In principal, the set of rules may be quite compact. However, in [34] the rules are derived from a complete AND/OR graph. Instead of deriving the constraints from geometry, the geometry is used to validate operations, and the rules are abstracted from the operations. This has two drawbacks. First, the AND/OR graph must be constructed explicitly before the rules can be found. Second, the resulting set of rules is very complex and must be simplified to obtain a concise representation. In contrast, this thesis has presented two implicit representations that are derived directly from the

geometry of the product itself.

7.2.3 Non-Directional Blocking Graph

By representing the blocking relationships between the parts of an assembly in all directions, an NDBG implicitly defines a set of assembly sequences for an assembly A. Namely, an assembly sequence τ *satisfies* the NDBG for A if for every assembly operation in τ partitioning S into S_1 and $S_2 = S \setminus S_1$, there exists a cell f in the NDBG such that no arcs connect S_1 to S_2 in $G(f)$. There are often an exponential number of geometrically feasible sequences for an assembly, yet the NDBG for A is always of polynomial size, and a sequence satisfying it can be found in polynomial time.

The non-directional blocking graph can be extended in several ways. The most general forms described here apply to infinitesimal rigid motions and extended translations, but the NDBG is not inherently limited to any type of motion. However, its construction for more complex motions may be difficult. An obvious variation would allow extended screw motions, i.e. extended rigid motions defined by a translation vector and a rotation about a constant axis parallel to the translation. For extended screw motions, the cells in the NDBG will no longer be bounded by linear constraints, greatly complicating the partitioning algorithm.

The NDBG might also be extended to support assembly operations that require more than two hands. For instance, when a product must be designed in such a way that it has no binary assembly sequence, this method could be invoked to find 3-handed assembly operations to construct it. Each 3-handed blocking graph $G(d_1, d_2, A)$ will contain information on the blocking relationships between all pairs of parts in A when some parts are stationary, others are moving in direction d_1, and still others are moving along d_2. In this case the possible composite motions would reside in a $2d$-dimensional space, where d is the dimension of each motion relative to the world coordinate system. For rigid motions and extended translations the cells will be bounded by linear constraints; however, analyzing each DBG for separable subassembly triples will be more complicated than for the 2-handed case.

7.2.4 Precedence Expressions

For the special case of linear assembly sequences, global precedence expressions constitute another type of implicit representation. After an AND/OR graph has been generated, the

final set of PEs capture the results of all geometric calculations performed to construct the graph. As a result, the AND/OR graph can be discarded and the set of assembly sequences represented by the PEs themselves. This shares the disadvantage with Homem de Mello's implicit representations that the AND/OR graph must be generated to compute the set of PEs. On the other hand, each PE is derived directly from the geometry of a subassembly instead of from an operation; as a result the rule set does not need to be simplified. For instance, 34 global PEs describe the same set of sequences for the 22-part electric bell as an AND/OR graph with 6190 AND-arcs.

It would be preferable to generate a small set of global PEs for a product without explicitly building its AND/OR graph. However, it remains to be seen whether this can be accomplished. One practical approach would simply generate them on demand, as when building an AND/OR graph. Thus an assembly planner would begin with no PEs, and each time a geometric answer is needed, it would first be answered by PEs, and the geometric module called when no answers are found there. The first few queries would generate geometric calls, but after a short time most constraints would be represented in PEs.

7.3 Other Applications

Although the methods described in this thesis were designed for assembly sequencing, some have the potential for wider application in reasoning about assemblies and other tasks.

The non-directional blocking graph has several uses in analyzing the motions of parts in an assembly. For instance, the NDBG could be used to efficiently identify motions of subassemblies in a product to ensure it will function properly, supplementing the methods in [38, 40]. If motions exist that are not desired, the design must be modified. Such techniques would be valuable to ensure the safety of toys, for example, since children can be ingenious in finding different motions than those intended by the designer [52].

Another application of the NDBG is in checking stability of assemblies. Palmer [51] showed that guaranteeing stability of a polyhedral assembly is NP-hard; however, stability can be determined efficiently for certain practical cases. If a subassembly is locally free, then the assembly might be unstable. Specifically, each locally free subassembly could be checked against the gravitational force. In practice this would catch many but not all unstable assemblies [66].

Finally, precedence expressions are general enough to store the results of many types of

calculations. For instance, the calculations performed by a grasp planner or stability checker could be stored in symbolic form to avoid recomputation when possible. In addition, PEs can be seen as a model for communication in a large reasoning system with heterogeneous agents. In such a system, queries should be answered with justifications and situations in which the answer applies, instead of yes/no answers. For instance, "no, part 17 is not graspable because part 4 interferes" is far more useful than "no, part 17 is not graspable." To construct these answers, the agents will require techniques similar to those used in Chapter 6 to construct precedence expressions.

Computer tools to evaluate assembly designs are a crucial enabling technology for concurrent engineering. This thesis has presented several techniques for reasoning about assemblies, specifically with regard to assembly sequencing. These techniques form the beginnings of an algorithmic approach to investigate the complexity of assembly designs.

Appendix A

Input to GRASP

In many integrated assembly systems, the connection graph and related information will be included as part of the input to an assembly planner. However, some of this information may be incomplete, or human input may need to be supplemented by automatic completion techniques. Furthermore, constraints on the assembly process could be represented in the input model of an assembly. For instance, the virtual contacts in Homem de Mello's relational model of an assembly [34] give non-contact, part blocking relationships, a large part of the geometric reasoning required to do assembly sequencing. For research in planning to be clear, an explicit boundary must be drawn between the description of an assembly and the reasoning necessary for planning. This appendix details the assembly description files used as input to GRASP, and describes the geometric reasoning routines that create the connection graph used as the basis of the planning process in Chapter 3.

A.1 The Assembly Description File

The input to GRASP consists of geometric descriptions of all parts in the assembly with their relative positions, and a list of declarations. The program uses Vantage [4], a three-dimensional modeling system, to build solid models of the parts and access geometric information in the models. The user defines the geometry and position of each part in constructive solid geometry, from which the modeler creates a boundary representation. Figure A.1 gives a sample input file to GRASP, and the model created is shown in figure A.2.

Vantage uses primarily a polyhedral boundary representation. However, it retains a small amount of non-planar information that is important to the operation of the assembly

```
(csgnode block1 cube (100 100 50) :trans (0 0 75 0 0 0))
(3d-structure block1)
(setq block1 (make-part :name "Block 1"
                        :b-rep 'block1z))

(csgnode c2 cube (100 100 100))
(csgnode hole cyl (20 50 10) :trans (25 0 0 0 90 0))
(csgnode block2 difference (c2 hole))
(3d-structure block2)
(setq block2 (make-part :name "Block 2"
                        :b-rep 'block2z))

(csgnode peg cyl (20 100 10) :trans (50 0 0 0 90 0))
(3d-structure peg)
(setq peg (make-part :name "Peg"
                     :b-rep 'pegz))

(setq simple (make-assembly :name "Simple Assembly"
                            :parts (list block1 block2 peg)
                            :decl '(threaded (20 0 20) 0.2)))

(fit-screen block2)
(complete-assembly simple)
```

Figure A.1: Sample GRASP assembly description

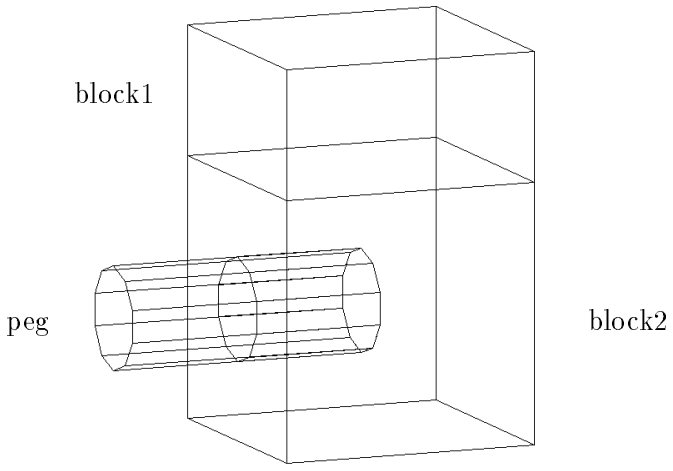

Figure A.2: The assembly created by the description file in figure A.1

planner. Specifically, a cylinder, sphere, or cone is approximated by a set of planar faces. However, Vantage records

- a parametric description of the original surface that generated a set of planar approximating faces, and
- either the simple curve that generated a sequence of line segment edges (such as the circle at each end of a cylinder primitive), or the intersection of two surfaces that resulted in the edges (such as a cylinder coming out of a plane at an angle, resulting in an elliptical intersection curve).

As a result, GRASP is able to recognize and reason about cylinders, cones, and spheres in a competent but limited way.

Some features of assemblies are not easily expressed in the form of geometric models. This information ranges from non-geometric features of the assembly, such as glued or press-fit contacts between parts, to geometric information that is more perspicuous and usable when represented symbolically, such as threaded surfaces and snap-fit connections. *Declarations* are designed to allow this information to be specified to the assembly sequencer. Since GRASP mainly reasons about geometry, only a declaration of threaded connections has been implemented; each threaded contact is identified by a point on the threaded surfaces and the pitch of the threads (in figure A.1 the ":decl" line declares a threaded contact between the `peg` and the hole in `block2`).

A.2 Building the Connection Graph

From the solid models built by Vantage, the procedure *COMPLETE-ASSEMBLY* deduces all the contacts between surfaces of parts and records the contact information in a connection graph model of the assembly. No tolerances are assumed on the parts.

The procedure *COMPLETE-ASSEMBLY* checks each pair of parts with intersecting bounding boxes for possible contacts. For each pair of possibly-touching parts, every pair of surfaces from different parts is checked for a possible contact. For every pair of surface types, a special-purpose routine determines whether there is contact between an instance of each. GRASP uses the following rules to recognize contacts between surfaces:

Planar A planar contact exists between two planar faces P_1 and P_2 if and only if:

- P_1 is coplanar with P_2,
- P_1's normal opposes P_2's normal, and
- the projections of P_1 and P_2 into their common plane intersect.

The polygon intersection required by the third step must ensure that neither polygon is contained in a hole of the other. For each planar contact, GRASP records the normal and the vertices of the convex hull of the two faces.

Cylindrical GRASP considers a cylindrical shaft surface C_1 and a cylindrical hole surface C_2 to be in contact if and only if:

- C_1 and C_2 have equal radius r,
- C_1 and C_2 have a shared axis A,
- the projections of C_1 and C_2 onto A overlap in an interval I, and
- no threaded declaration indicates a point whose distance to A is equal to r and whose projection onto A is in I.

In fact, the third condition is necessary but not sufficient for the cylinders to be in contact. Figure A.3a shows two cylindrical surface patches that *COMPLETE-ASSEMBLY* will incorrectly record as contacting. To detect such cases would require intersecting the cylinder surface patches, for instance by projecting the two surfaces onto a plane and intersecting the projections. GRASP records the common axis of the two cylinders for a cylindrical contact.

Cylinder-Plane A cylindrical shaft surface C and a planar face P are in contact if and only if:

- the axis of C and the normal of P are perpendicular,
- the radius of C is equal to the distance from C's axis to the plane of P, and
- the projection of C's axis intersects with P in its plane

The third criteria suffers from similar problems as the cylindrical contact test, and a counter-example is shown in figure A.3b. For a cylinder-plane contact, GRASP records the normal of P and the two endpoints of the line of contact.

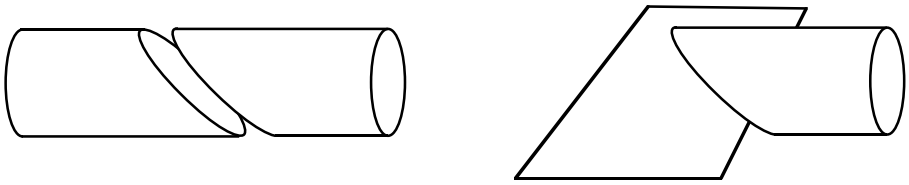

Figure A.3: Two contacts GRASP incorrectly detects

Threaded A cylindrical shaft surface C_1 and a cylindrical hole surface C_2 are in threaded contact with thread pitch p if and only if:

- C_1 and C_2 have equal radius r,
- C_1 and C_2 have a shared axis A,
- the projections of C_1 and C_2 onto A overlap in an interval I, and
- a threaded declaration with pitch p indicates a point whose distance to A is equal to r and whose projection onto A is in I.

For a threaded contact, the axis of the cylinder and the thread pitch are recorded in the connection graph.

Sphere-Plane A sphere S is in contact with a planar face P if and only if:

- the distance from the center of S to P is equal to the radius of S, and
- the projection of the center of S onto P is inside the polygon P.

The normal of P and the point of contact fully characterize a spherical contact.

The contact types given above comprise the large majority of contacts in mechanical assemblies. Similar routines could be devised to find other contacts such as point-plane, edge-edge, and point-edge contacts. However, the current implementation does not detect these contacts.

For greater efficiency in identifying contacting surfaces, standard methods of geometric modeling could be applied. In one scheme, a uniform grid is placed across three-dimensional space, dividing it into small cubes. Then only surfaces intersecting the same cube need be checked for contact. In another scheme, surfaces could be grouped in bins according to characteristic attributes; only surfaces in certain pairs of bins need be checked for contacts.

```
(block2 (name "Block 2")
        (assembly simple)
        (b-rep block2z)
        (part-number 2)
        (links ((peg (threaded-contact (1.0 0.0 0.0)
                                       (50.0 0.0 0.0)
                                       0.2)
                     (planar-contact (-1.0 0.0 0.0)))
                (block1 (planar-contact (0.0 0.0 -1.0))))))
```

Figure A.4: GRASP's contact representation

For instance, each planar face could be placed in a unique bin in a three-dimensional array, where the bin coordinates are the coordinates of the dual point [18] of the supporting plane of the face. Then only faces in the same bin need be tested for possible contact. Similar schemes could be used for cylindrical and other types of faces. Such a technique has not been implemented in GRASP.

COMPLETE-ASSEMBLY also finds and records symmetries of individual parts, which are used later in assembly sequencing to detect useless motions (see section 3.3). A line L is an axis of symmetry for a part p if and only if all cylindrical and conical surfaces of p have L as their axis, L passes through the center points of all spherical surfaces of p, and all planar faces of p have normals parallel to L. A point C is a point of symmetry for p if and only if all surfaces of p are spherical with center C.

Figure A.4 shows GRASP's representation of `block2` and its connections, from figure A.2. The points on the convex hull of the planar contacts are not shown. The connection graph, along with the boundary representations of the individual parts of the assembly, forms the basis of the planning process.

Vantage requires 155 seconds to generate the boundary representation of the electric bell (Appendix B) from the CSG description, and *COMPLETE-ASSEMBLY* finds all contacts between the parts of the bell in another 45 seconds.

Appendix B

Assemblies

This appendix describes several assemblies for which GRASP has generated assembly sequences using the methods described in this thesis. The experimental results are given in the chapters where the methods are described. All assemblies but the transmission are described using Vantage and are available by email from the author at `rwilson@cs.stanford.edu`.

B.1 The Transmission

The *Assembly from Industry* is a simplified model of a transmission with which De Fazio and Whitney [25] illustrate their method for generating assembly sequences. Figure B.1 shows the transmission, and figure B.2 shows its liaison diagram. The transmission has 11 parts, or 21 parts when the the bolts are explicitly represented. It is symmetric around an axis of revolution, and as such its geometry can be fully modeled in the two dimensions of the GRASP prototype [65]. The prototype of GRASP uses a special purpose representation of parts as possibly disconnected polygons in the plane. Experimental results on the transmission are given in chapter 6.

B.2 The Electric Bell

The *electric bell* is a 22-part assembly made from a kit. Figure B.3 shows the solid model Vantage generates from the bell description file. GRASP does not represent or reason about the flexible wires in the real bell. A simplified version of the bell with 17 parts was used for some experiments.

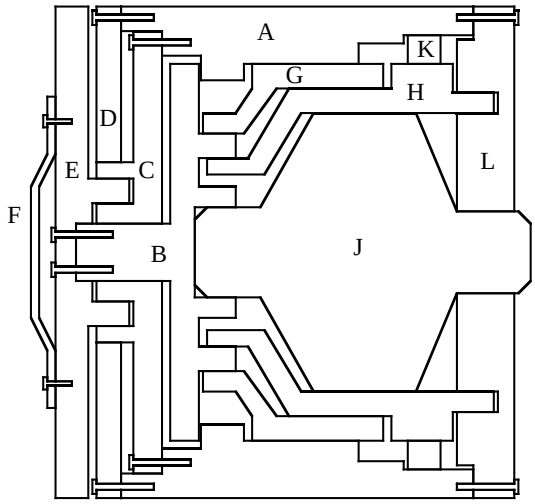

Figure B.1: The transmission

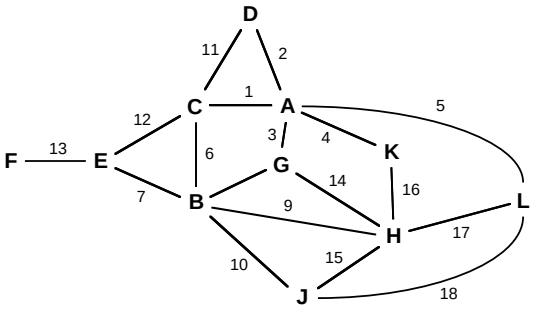

Figure B.2: Liaison diagram for the transmission

B.3 The Skin Machine

Figure B.4 shows the friction testing machine, or *skin machine*, a mechanism designed to allow precise translational force testing in a single direction. The real skin machine was used for experiments described in [21], and it was modeled and used by Konkar et al. [40] in a concurrent design system for assemblies. The skin machine consists of 12 main parts and 24 fastening screws. In some ways it is a bad case for any assembly planner that generates a large amount of the AND/OR graph, since the screws can be placed in any order. Even if only one sequence is generated, the skin machine is difficult for a decomposition procedure based on generating subassemblies and then testing their movability, since more than 2^{16} candidate subassemblies exist in the first decomposition.

In addition, the skin machine is designed in such a way that it is severely overconstrained

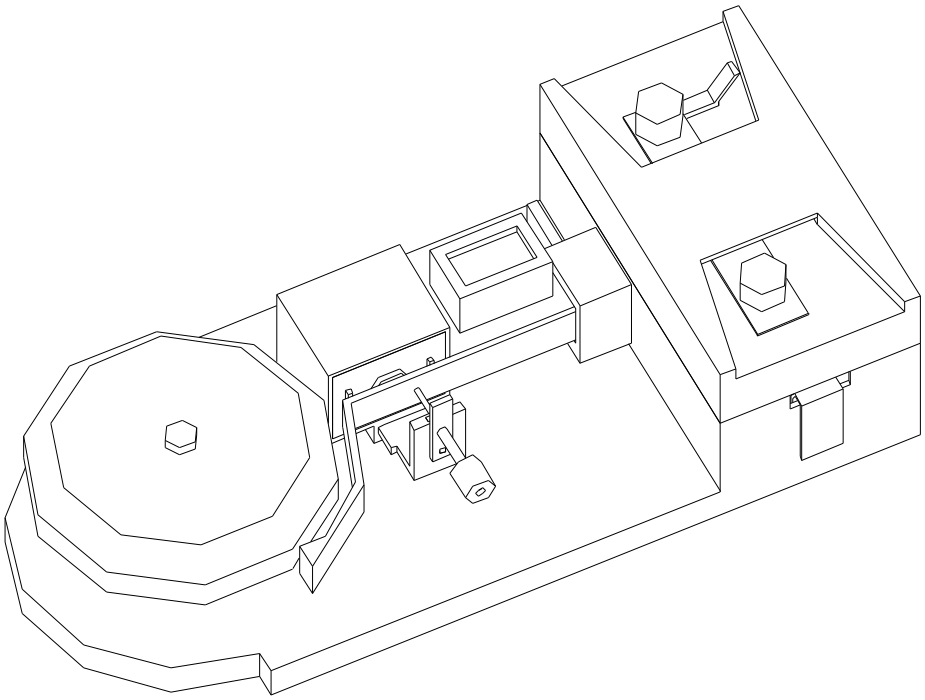

Figure B.3: The electric bell

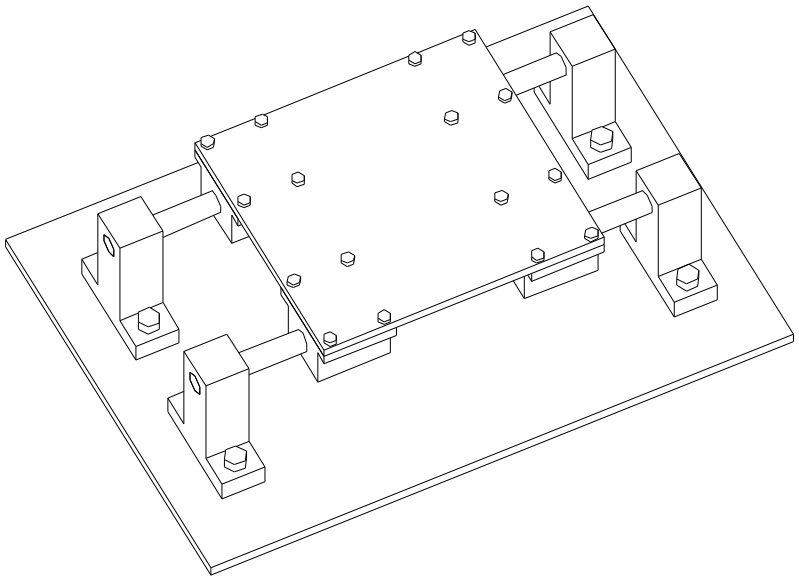

Figure B.4: The skin machine

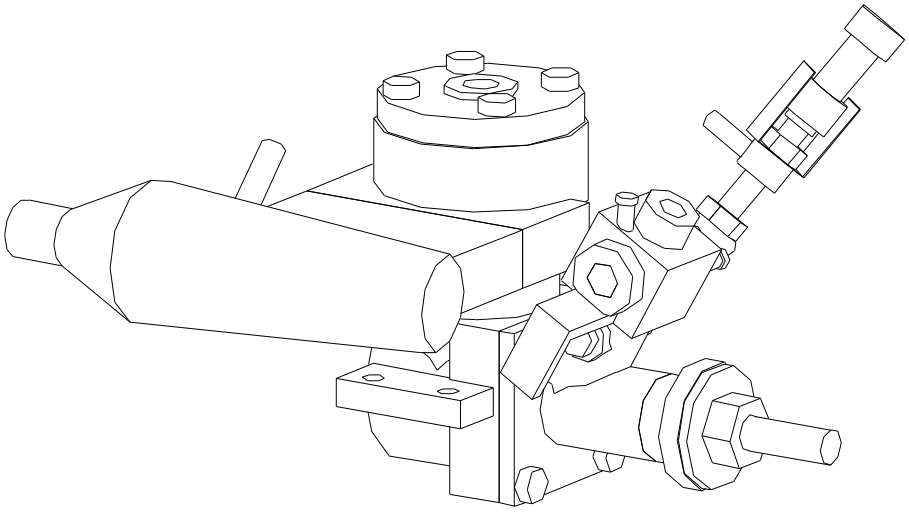

Figure B.5: The engine

kinematically. As a result, it only functions correctly if care is taken during assembly to keep certain parts perfectly aligned as the screws are tightened. These manufacturing constraints cannot be reasoned about in a purely geometric assembly sequencer such as GRASP.

B.4 The Engine

The Enya 09-IV T.V. model aircraft engine is a single-piston internal combustion engine with 42 parts. The assembled engine is shown in figure B.5. Several versions of the engine model were planned for, including

- a 12-part model including most of the major parts,
- a 30-part version with all non-fastener parts included, and
- the full 42-part model including all fastening bolts.

It is interesting to note that, contrary to the assumptions of many papers on assembly planning, the geometry of the fasteners affect the possible assembly sequences for the engine. Specifically, two bolts securing the carburetor to the intake manifold obstruct the motion of two of the four bolts fastening the crankcase to the engine body. Any assembly sequence analysis without explicit geometric models of the fasteners would certainly miss this constraint on the assembly orders. In addition, the interference might be an example of

a manufacturing feature that could have been detected and corrected using early sequence analysis in a concurrent engineering design environment. However, this is hard to determine without knowing the design history of the engine.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] R. Alami, T. Simeon, and J-P Laumond. A geometrical approach to planning manipulation tasks. The case of discrete placements and grasps. In *Preprints of the Fifth Intl. Symp. of Robotics Research*, pages 113–119, 1989.

[3] E. M. Arkin, R. Connelly, and J. S. B. Mitchell. On monotone paths among obstacles, with applications to planning assemblies. In *Proc. 5th ACM Symp. on Computational Geometry*, pages 334–343, 1989.

[4] P. Balakumar, J.-C. Robert, R. Hoffman, K. Ikeuchi, and T. Kanade. *VANTAGE: A Frame-Based Geometric Modeling System - Programmer/User's Manual V1.0*. The Robotics Institute, Carnegie-Mellon Univ., 1989.

[5] D. F. Baldwin. Algorithmic methods and software tools for the generation of mechanical assembly sequences. Master's thesis, Massachusetts Institute of Technology, 1990.

[6] R. S. Ball. *A Treatise on the Theory of Screws*. Cambridge Univ. Press, 1900.

[7] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. *Intl. J. of Robotics Research*, 10(6):628–649, 1991.

[8] T. Binford, L. Frants, M. Cutkosky, and J.-C. Tsai. Represention and propagation of tolerances for cad/cam systems. In *Proc. IFIP WG5.2 Workshop in Geometric Modeling*, 1990.

[9] M. Blum, A. Griffith, and B. Neumann. A stability test for configurations of blocks. Memo 188, MIT AI Lab, 1970.

[10] N. Boneschanscher and C. J. M. Heemskerk. Grouping parts to reduce the complexity of assembly sequence planning. In E. A. Puente and L. Nemes, editors, *Information Control Problems in Manufacturing Technology 1989: Selected Papers from the 6th IFAC/IFIP/IFORS/IMACS Symposium*, pages 233–238. Pergamon Press, 1989.

[11] N. Boneschanscher, H. van der Drift, S. J. Buckley, and R. H. Taylor. Subassembly stability. In *Proc. National Conf. on Artificial Intelligence*, pages 780–785, 1988.

[12] G. Boothroyd and P. Dewhurst. *Design for Assembly: A Designer's Handbook.* Boothroyd Dewhurst, 1983.

[13] O. Bottema and B. Roth. *Theoretical Kinematics.* North-Holland, 1979.

[14] A. Bourjault. *Contribution à une approche méthodologique de l'assemblage automatisé: élaboration automatique des séquences opératoires.* PhD thesis, Faculté des Sciences et des Techniques de l'Université de Franche-Comté, 1984.

[15] J. Canny. Collision detection for moving polyhedra. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(2):200–209, 1986.

[16] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(5):485–524, 1991.

[17] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

[18] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[19] K. L. Clarkson. Applications of random sampling in computational geometry, II. In *Proc. 4th ACM Symp. on Computational Geometry*, pages 1–11, 1988.

[20] M. R. Cutkosky. *Robotic Grasping and Fine Manipulation.* Kluwer, 1985.

[21] M. R. Cutkosky, J. M. Jourdain, and P. K. Wright. Skin materials for robotic fingers. In *Proc. IEEE Conf. on Robotics and Automation*, pages 1649–1654, 1987.

[22] M. R. Cutkosky and J. M. Tenenbaum. A methodology and computational framework for concurrent product and process design. *ASME Journal of Mechanism and Machine Theory*, 25(3):365–381, 1990.

[23] M. R. Cutkosky and J. M. Tenenbaum. Toward a framework for concurrent design. *Intl. J. of Systems, Automation: Research and Applications*, 1(3):239–261, 1991.

[24] R. J. Dawson. On removing a ball without disturbing the others. *Mathematics Magazine*, 57(1):27–30, 1984.

[25] 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.

[26] N. Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice-Hall, 1974.

[27] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[28] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[29] M. Genesereth. Designworld. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 2785–2788, 1991.

[30] A. J. Goldman and A. W. Tucker. Polyhedral convex cones. In H. W. Kuhn and A. W. Tucker, editors, *Linear Inequalities and Related Systems*, pages 19–40. Princeton Univ. Press, 1956.

[31] L. Guibas and F. F. Yao. On translating a set of rectangles. *Advances in Computing Research*, 1:235–260, 1983.

[32] H. Hirukawa, T. Matsui, and K. Takase. Automatic determination of possible velocity and applicable force of frictionless objects in contact from a geometric model. *IEEE Trans. on Robotics and Automation*, 10(3):309–322, 1994.

[33] R. L. Hoffman. A common sense approach to assembly sequence planning. In L. S. Homem de Mello and S. Lee, editors, *Computer-Aided Mechanical Assembly Planning*, pages 289–314. Kluwer, 1991.

[34] L. S. Homem de Mello. *Task Sequence Planning for Robotic Assembly.* PhD thesis, Carnegie-Mellon Univ., 1989.

[35] L. S. Homem de Mello and A. C. Sanderson. AND/OR graph representation of assembly plans. Technical Report CMU-RI-TR-86-8, Robotics Institute - Carnegie-Mellon Univ., 1986.

[36] L. S. Homem de Mello and A. C. Sanderson. Automatic generation of mechanical assembly sequences. Technical Report CMU-RI-TR-88-19, Robotics Institute - Carnegie-Mellon Univ., 1988.

[37] J. Jones and T. Lozano-Pérez. Planning two-fingered grasps for pick-and-place operations on polyhedra. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 683–688, 1990.

[38] L. Joskowicz and E. Sacks. Computational kinematics. *Artificial Intelligence*, 51(1–3):381–416, 1991.

[39] H. Ko and K. Lee. Automatic assembling procedure generation from mating conditions. *Computer Aided Design*, 19(1):3–10, 1987.

[40] R. R. Konkar, M. R. Cutkosky, and J. M. Tenenbaum. Towards an assembly editor for concurrent product and process design. In *Proc. IFIP WG 5.2 Workshop on Geometric Modeling*, 1990.

[41] S. S. Krishnan and A. C. Sanderson. Path planning algorithms for assembly sequence planning. In *Intl. Conf. on Intelligent Robotics*, pages 428–439, 1991.

[42] J.-C. Latombe. *Robot Motion Planning*. Kluwer, 1991.

[43] J.-C. Latombe, A. Lazanas, and S. Shekhar. Robot motion planning with uncertainty in control and sensing. *Artificial Intelligence*, 52:1–47, 1991.

[44] S. Lee and Y. G. Shin. Assembly planning based on geometric reasoning. *Computation and Graphics*, 14(2):237–250, 1990.

[45] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, 1983.

[46] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor. Automatic synthesis of fine-motion strategies for robots. *Intl. J. of Robotics Research*, 3(1):3–24, 1984.

[47] J. S. B. Mitchell. Personal communication, December 1990.

[48] B. K. Natarajan. On planning assemblies. In *Proc. 4th ACM Symp. on Computational Geometry*, pages 299–308, 1988.

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

[50] D. K. Pai and B. R. Donald. On the motion of compliantly-connected rigid bodies in contact, part I: The motion prediction problem. Technical Report 89-1047, Dept. of Computer Science – Cornell Univ., 1989.

[51] R. S. Palmer. *Computational Complexity of Motion and Stability of Polygons*. PhD thesis, Cornell Univ., 1989.

[52] E. Paté-Cornell. Personal communication, March 1992.

[53] J. Pertin-Troccaz. Grasping: A state of the art. In Khatib, Craig, and Lozano-Pérez, editors, *The Robotics Review 1*. MIT Press, 1989.

[54] R. Pollack, M. Sharir, and S. Sifrony. Separating two simple polygons by a sequence of translations. *Discrete and Computational Geometry*, 3:123–136, 1988.

[55] F. P. Preparata and D. E. Muller. Finding the intersection of n halfspaces in time $O(n \log n)$. *Theoretical Computer Science*, 8:45–55, 1979.

[56] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[57] E. Sacerdoti. *A Structure for Plans and Behavior*. American Elsevier, 1977.

[58] J-R Sack and G. T. Toussaint. Separability of pairs of polygons through single translations. *Robotica*, 5:55–63, 1987.

[59] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991.

[60] G. T. Toussaint. Movable separability of sets. In G. T. Toussaint, editor, *Computational Geometry*. Elsevier, 1985.

[61] J-M Valade. Geometric reasoning and automatic synthesis of assembly trajectory. In *Proc. Intl. Conf. on Advanced Robotics*, pages 43–50, 1985.

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

[63] R. H. Wilson. Efficiently partitioning an assembly. In *Proc. IASTED Intl. Symp. on Robotics and Manufacturing*, pages 34–37, 1990.

[64] R. H. Wilson. Efficiently partitioning an assembly. In L. S. Homem de Mello and S. Lee, editors, *Computer-Aided Mechanical Assembly Planning*, pages 243–262. Kluwer, 1991.

[65] R. H. Wilson and J-F Rit. Maintaining geometric dependencies in an assembly planner. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 890–895, 1990.

[66] R. H. Wilson and J-F Rit. Maintaining geometric dependencies in assembly planning. In L. S. Homem de Mello and S. Lee, editors, *Computer-Aided Mechanical Assembly Planning*, pages 217–242. Kluwer, 1991.

[67] R. H. Wilson and A. Schweikard. Assembling polyhedra with single translations. Technical Report STAN-CS-91-1387, Dept. of Computer Science, Stanford Univ., 1991.

[68] J. D. Wolter. *On the Automatic Generation of Plans for Mechanical Assembly.* PhD thesis, Univ. of Michigan, 1988.

[69] J. D. Wolter. On the automatic generation of assembly plans. In L. S. Homem de Mello and S. Lee, editors, *Computer-Aided Mechanical Assembly Planning*, pages 263–288. Kluwer, 1991.