Automatic precedence relationship extraction for assembly sequence generation

by

Hung-Yi Tu

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Industrial and Manufacturing Systems Engineering
Major: Industrial Engineering

Iowa State University
Ames, Iowa
1992

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Overview

Studies have shown that 75% of a product's life cycle cost is determined in the early stages of design, and later decisions make only minor changes to the total cost. A concurrent design approach in which the design of product, process, and system may be examined simultaneously in the early design phase will significantly reduce life cycle cost and product design lead time (Nevins and Whitney, 1989). Therefore, procedures which will allow the designer to quickly evaluate the process aspect of a product design are needed for the implementation of concurrent design. The current research efforts in manufacturing process planning encompass machining process planning, often referred to as process planning, and assembly planning. In the area of machining, most of the machines and processes are well established, and thus machining processes can be selected by applying manufacturing knowledge to relate the part features to the existing processes. On the other hand, assembly planning techniques are not well established. Thus, it is difficult to develop a planning system which can automatically generate assembly plans for various types of products. This is also the reason that the assembly planning function has been very much experience-based and human-dependent.

The major activities involved in assembly planning include (Nevins and Whitney, 1989):

1. Establishment of an assembly sequence.

2. Division of the product into subassemblies.

3. Selection of an assembly method for each step.

4. Integration of a quality control strategy.

5. Economic analysis and choice of assembly method.

Point 1 above shows that the derivation of valid assembly sequences from a design model is a major activity of assembly planning. De Fazio and Whitney (1987) characterized a product as a network graph, or liaison diagram, in which the parts are represented by nodes and the liaisons between parts are represented by arcs. A complete assembly is produced if all the liaisons are properly established. All the geometric constraints have to be satisfied when a sequence is generated; otherwise, the product cannot be assembled successfully. A geometrically feasible assembly sequence may not be feasible when process constraints such as assembly machines, assembly fixtures, assembly tools, and assembly system layouts are considered. Therefore, because of the geometric and/or process constraints, liaison precedences exist, forcing some liaisons to be established before others. Unless these precedence requirements are followed, the assembly sequence will not lead to a complete product assembly. If all of the liaisons and the precedences among them can be properly identified, a complete set of valid liaison assembly sequences can be generated.

Research Objective

One objective of this research is to develop a methodology for automatic precedence relationship extraction and to implement this methodology in a solid modeling environment. The second objective is to generate all valid liaison assembly sequences based on the precedence relationships extracted.

Assumptions

To carry out the automatic precedence relationship extraction, this research assumes that a detailed product design in a solid modeling environment is available. The assumptions in this research are stated below:

1. Product parts are rigid bodies which generally do not change shape during assembly. Deformable objects are not considered in this research.

2. The effects of dimensional and geometric tolerances for the product design are not taken into account. For this reason, the part solid models represent only nominal sizes of part solids.

3. Only cylindrical and planar mating faces are considered.

4. The contacts between parts are limited to surface contacts only; line-contacts and point-contacts are not considered.

5. All parts are 1-disassemblable (Woo and Dutta, 1991). A part is 1-disassemblable if only one single translation is needed to remove it. Rotation and sequential disassembly movements are not considered in this research.

6. No process constraints are imposed on the methodology. The liaison assembly sequences generated from the extracted precedence relationships will represent all of the geometrically feasible sequences.

7. The liaison diagram of a product design has to be in the simple-cycle structure. The definition of simple-cycle liaison diagrams will be discussed in Chapter 3.

Basic Approach

In a manual design and planning environment, the assembly planner uses the design drawings to figure out how parts can be put together. The reasoning process is an interlinked activity and based mostly on the expertise of the planner. This is an iterative process. The planner sometimes needs to consult with the designer with regard to the design itself, the functionality, the assembly feasibility, and so forth. This can be a costly and time consuming approach, and often does not lead to a good candidate set of assembly sequences.

In this research, a disassembly approach is used to derive the precedence relationships in a solid modeling environment. A framework for automatic precedence relationship extraction and assembly sequence generation is shown in Figure 1.1. Implementation of this approach will significantly alleviate the problems mentioned above. This approach requires the following inputs:

1. The liaison diagram of the product.

2. The disassembly directions for each part in the product.

3. The product design represented in a solid modeling system.

The output is a set of precedence relationships. Based on these precedence relationships, all of the liaison assembly sequences can be generated. The main focus of the current study lies on the precedence extraction methodology.

Organization of This Thesis

This thesis is organized into 5 chapters. Chapter 1 is the introduction. Chapter 2 gives a review of related research in the area of assembly sequence generation. Chap-

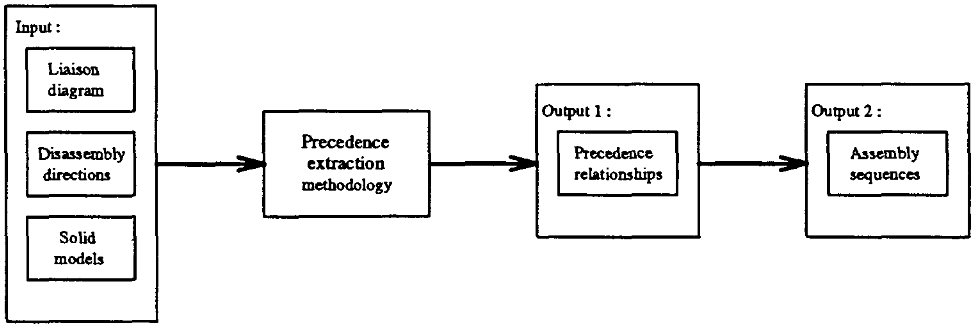


Figure 1.1: Framework of automatic precedence relationship extraction and assembly sequence generation

ter 3 presents the background information related to the subject and the proposed methodology for precedence relationship extraction. Implementation of the proposed methodology is discussed in Chapter 4. Finally, a summary of the research work and future research in assembly planning are given in Chapter 5.

CHAPTER 2. LITERATURE REVIEW

In this chapter, a review on related research works conducted in the area of assembly sequence generation is given. Since the proposed methodology employs interference detection, a brief review on that subject is also presented.

Assembly Sequence Generation

As mentioned in the previous chapter, the techniques of assembly planning are not well established, thus it is difficult to develop a planning system for automatic assembly sequence generation. A study (Nevins and Whitney, 1989) shows that generating assembly sequences manually can be a tedious process with no guarantee that all of the good ones will be discovered. Various methods for assembly sequence generation have been attempted by researchers.

Huang and Lee (1989) proposed a representation and acquisition of the precedence knowledge of an assembly from the viewpoint of disassembling the product. An assembly is described by an undirected graph called "Feature Mating Operation Graph" (FMOG). This assembly graph $G = (V,E)$ consists of a finite nonempty set of vertices V and a set of edges E connecting them. Two different vertices are present in the graph: square vertices representing components and circle vertices representing feature mating operations between components. Each feature mating operation is represented by a Boolean variable indicating whether the operation is done or undone.

Extending this concept to an assembly, the state of an assembly is described by the conditions of all of its feature mating operations. A geometric mating graph is also developed to include all of the necessary geometric and topological information for the precedence knowledge acquisition. Then, two algorithms are developed to obtain the precedence knowledge from the geometric mating graph.

Ko and Lee (1987) developed a method which used mating conditions as the input and generate an assembly procedure in two steps. First, each component in an assembly is located at a specific vertex of a hierarchical tree. Second, an assembly procedure is generated from the hierarchical tree with the help of interference checking. Hoffman (1989) presented a technique which can take Constructive Solid Geometry representations of two objects along with the relative position of the two objects corresponding to their mated position and discover a path for extracting one object from the other. The reverse of this procedure forms an assembly sequence for the composite object. Miller and Hoffman (1989) used fasteners as the crucial items to determine a valid assembly sequence. Woo and Dutta (1991) proposed an approach by traversing the Disassembly Tree (DT) in pre- and post-order to yield a sequence of minimal number of operations for disassembly and assembly respectively.

All of the methodologies mentioned above generate only a single valid assembly sequence based on the product geometry. Although the sequences generated by using above methodologies are geometrically feasible, they could be infeasible because of process or other constraints, such as designer intent, facility restrictions, cost, fixturing infeasibility, etc. Since only a single valid sequence is generated using the above methodologies, a set of "good" sequences is less likely to be found.

Lin and Chang (1990) developed a planning methodology, which accepts the part and assembly boundary models as inputs, does reasoning on the model, and then automatically generates feasible plans for the assembly of the product. Although this methodology can generate more than one valid assembly sequences, it is restricted to considering only one subassembly during the assembly process. In other words, two or more subassemblies cannot be processed in parallel.

Homem De Mello and Sanderson (1986, 1990) used a decomposition approach to generate an AND/OR graph representation of all possible assembly plans by representing all the possible configurations of the assembly. Based on this approach, Khosla and Mattikali (1989) developed a methodology to automatically determine the assembly sequence from a 3-D solid modeler description of the assembly. Their approach consists of automatically determining a set of assembly operations, through a disassembly procedure, that leads to the given assembly. In both methods, one starts with the completed product and systematically disassembles it by every possible path. The search for every possible path is exhaustive and thus will become too cumbersome to use if there are more than a few parts.

Bourjault (1984) presented a methodology for generating all assembly sequences algorithmically from a set of rules derived from the answers to a series of questions about the mating of part pairs and multiples of parts. Each question is answered with "yes" or "no" and can be phrased so that either (or a mix) answer calls for subsequent action. A modification of Bourjaults's method was presented by De Fazio and Whitney (1987). They represented the assembly as a network graph, where the parts are represented by nodes and liaisons are represented by arcs connecting the related nodes. Such a network representation of assembly is called "liaison diagram" (see Figure 2.1). Once the product is characterized as a network of parts and liaisons, the user has to answer two questions for each liaison. The questions to be answered are as follows. For $i = 1$ to l, where l is total number of liaisons in a product

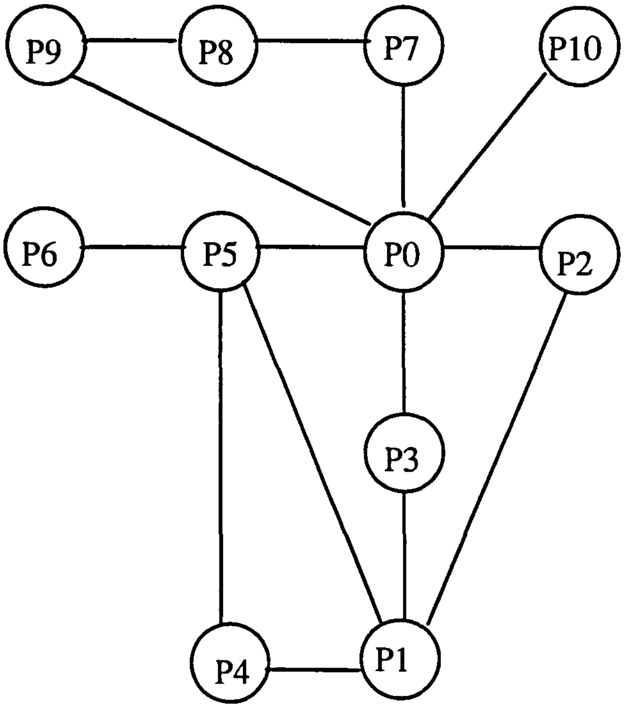


Figure 2.1: An example of liaison diagram

design,

Q1) what liaisons must be done prior to doing liaison i ?

Q2) what liaisons must be left to be done after doing liaison i ?

The questions are to be answered for each liaison. Answers are to be expressed in the form of a precedence relationship between liaisons or between logical combinations of liaisons. There are exactly $2l$ questions to be answered, two associated with each liaison. Based on the user's answers to the questions, a complete set of assembly sequences can be derived. Lui (1989) extended Whitney's work to construct the directed graph of assembly states representing all of the assembly sequences using these precedence relationships and the liaison diagrams.

Both approaches (Bourjaults 1984, De Fazio and Whitney 1987) lead themselves to interactive systems which rely on designer's ability to correctly answer questions. For simple assemblies with a small number of liaisons, these approaches work fairly well. However, with an increasing number of parts, the number of liaisons (and questions) increases fairly quickly, making these approaches infeasible for many products. The number of liaisons for a product with n parts can be n - $1 \leq l \leq \binom{n}{2}$. A product of only 10 components would have between 9 and 45 liaisons, and between 18 to 90 questions to be answered. As the number of liaisons increases, it becomes difficult for a person to guarantee the consistency and correctness of the answers. De Fazio *et al.* (1991) proposed an approach called the "onion-skin" method for assembly sequence generation. The "onion-skin" concept of "peeling" parts from an assembly in layers, much like peeling the skin of an onion, is used to find assembly precedence. This methodology also requires that a human answer questions to determine whether or not the product (or subassembly) can be separated into two components. Thus, it still requires human judgment to determine sequences. This increases the chances of making a judgment error.

Since many factors can affect the selection of assembly sequence, the methodologies generating only a single assembly sequence are not reasonable approaches because process and other constraints may make this single sequence infeasible. Thus, study of all possible alternatives with respect to the sequences of assembly is essential for automatic assembly planning. Because of the possibilities of making errors, human interaction in assembly sequence generation should also be reduced.

Interference Detection for Part Disassembly

When two or more separate parts are brought together in an assembly, the parts should not intersect. To provide a tool to detect design mistakes in a CAD environment, a static interference checking function is often embedded in solid modeling systems. As mentioned in the previous chapter, the approach used in this thesis is a disassembly approach that automatically extracts all of the precedence relationships between liaisons. Thus, the entire assembly has to be disassembled to determine precedences. This raises the need for dynamic interference detection.

The basis principle of dynamic interference detection is to check the intersection between stationary objects and the trajectory of a moving object. Boyse (1979) presented a computer representation for solids and surfaces and algorithms which carry out interference checking among objects. Objects are represented as polyhedra or as piecewise planar surfaces. To detect a collision between two objects, it is sufficient to detect a collision of an edge on one object with a face of the other or vice versa. W.P. Wang and K.K. Wang (1986) proposed a method for modeling swept volume by computing a family of critical curves, which is the boundary of the swept volume, from a moving solid. Roth (1982) developed a method called "Ray Casting" as the methodological basis for a CAD/CAM solid modeling system. In order to visualize and analyze the model, virtual light rays are cast as probes. This method is also applied in assembly planning for dynamic interference detection. Rays are cast from the moving part in its moving direction. If any ray intersects with the stationary part, collision is detected.

For some types of assemblies, especially three-dimensional mechanisms, non-linear mating paths are sometimes needed to insert certain parts. Lozano-Perez (1979, 1983) developed the configuration-space approach which is a fundamental analytical tool used to determine non-linear mating paths from interrelated objects. A configuration of a part is a set of parameters which uniquely specifies the position of every point on the part, and the configuration space is the set of all possible configurations.

In the configuration space, the problem of planning the motion of a part through a space of obstacles is transformed into an equivalent problem of planning the motion of a point through a space of enlarged configuration-space obstacles. Finally, Wang (1990) presented a 3-dimensional collision avoidance algorithm for controlling complicated machine motions.

CHAPTER 3. PROPOSED METHODOLOGY

In this chapter, a methodology for deriving the liaison precedence relationships for a product design which is characterized by a simple-cycle liaison diagram is described. Before the proposed methodology is presented, the background information related to the subject is stated first.

Preliminaries

The related background information including liaison diagram structures, precedence relationship representations, and disassembly directions is described in this section.

Liaison Diagram Structures

A liaison diagram, as shown in Figure 2.1, is an abstract representation of a product assembly. As mentioned in the previous chapter, a liaison diagram is used to represent an assembly product. The parts are represented by nodes and liaisons are represented by arcs connecting the related nodes. Although it may take many different forms, a liaison diagram may be constructed from two basic structures:

1. **Tree:** A diagram T is a tree if, and only if, every two distinct nodes of T are joined by a unique path (see Figure 3.1.(a)).

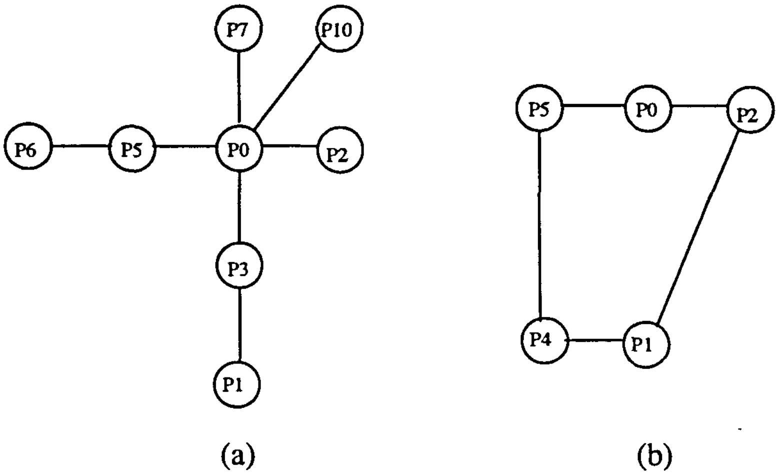


Figure 3.1: Two basic structures for liaison diagrams: (a) tree, (b) simple cycle.

2. **Simple cycle:** A simple cycle is a diagram in which each node (part) is associated exactly with two arcs (liaisons) (see Figure 3.1.(b)).

In this thesis, a methodology is developed to extract all of the precedence relationships for a product design which is characterized by a simple-cycle liaison diagram.

Precedence Relationship Representations

A liaison precedence relationship can be expressed in a form of $A \longrightarrow B$. Both A and B can be a single liaison or a logical combination of liaisons. Several examples are shown as following in which the precedence sign "$\longrightarrow$" is read "must precede".

$$L_4 \longrightarrow (L_1 \wedge L_3)$$
$$L_4 \longrightarrow (L_1 \vee L_3)$$
$$(L_1 \wedge L_3) \longrightarrow L_4$$

where "$\wedge$" and "$\vee$" respectively denote the "and" and "or" operators in Boolean algebra. Each liaison, L_i, in a precedence relationship is regarded as a Boolean variable indicating whether or not the liaison has been established.

$$L_i = \begin{cases} 1 & \text{established} \\ 0 & \text{unestablished} \end{cases}$$

All of the precedence requirements must be met as the assembly process progresses. This means that the following two conditions must be met:

Condition *1*: When the Boolean value of the left hand side is 0, the Boolean value of the right hand side must be 0 too.

Condition *2*: When the Boolean value of the left hand side is 1, the Boolean value of the right hand side can be either 0 or 1.

These two conditions are demonstrated with the example, $L_i \longrightarrow (L_j \vee (L_k \wedge L_l))$. Let B_L and B_R be the Boolean values of the left hand side and right hand side of the precedence sign respectively.

1. Assume $L_i = 0$, $L_j = 1$, $L_k = 0$, $L_l = 1$

   $B_L = L_i = 0$,

   $B_R = L_j \vee (L_k \wedge L_l) = 1 \vee (0 \wedge 1) = 1$

   Condition *1* is violated, so the sequence would be invalid because the precedence requires that L_i is established before (L_j) or $(L_k \wedge L_l)$ is done.

2. Assume $L_i = 1$, $L_j = 0$, $L_k = 1$, $L_l = 1$

   $B_L = L_i = 1$,

   $B_R = L_j \vee (L_k \wedge L_l) = 0 \vee (1 \wedge 1) = 1$

   Neither of the two conditions is violated, so the sequence satisfies this precedence relationship.

Liaison precedence relationships can take several different forms. Two basic forms of precedence relationships are extracted in this thesis:

1. **Simple form** has the form of $L_i \longrightarrow L_j$ that it has only one liaison on each side of the precedence sign.

2. **Complex form** has the form of $L_i \longrightarrow (L_j \wedge L_k \wedge ...)$ that it has one liaison on the left, but the right hand side of the precedence sign is a Boolean combination of several liaisons by using "$\wedge$" Boolean operator only.

Different forms of precedence relationships can be decomposed into the two basic forms stated above. For example, a precedence relationship of the form $(L_i \wedge L_j) \longrightarrow L_k$ can be decomposed into two simple-form precedence relationships:

$L_i \longrightarrow L_k$
$L_j \longrightarrow L_k$

Another example, $L_i \longrightarrow ((L_j \wedge L_k) \vee L_l)$, can be decomposed into a combination of a simple-form and a complex-form precedence relationships:

$L_i \longrightarrow L_l$
$L_i \longrightarrow (L_j \wedge L_k)$

Disassembly Directions

In this thesis, it is assumed that the product design will include only planar and cylindrical faces. In order that the program be able to reason the precedence relationships, the disassembly directions for each liaison must be identified. Different procedures are employed to select the disassembly directions for these two types of mating faces:

1. **Cylindrical** mating faces are constrained such that the disassembly directions must be along the cylindrical axis. Let **n** be the unit vector of the axis of the cylindrical face. Both **n** and **-n** are the possible disassembly directions.

2. For **planar** faces, the procedure to determine the disassembly directions is based on the work of Chen and Woo (1992), and Dutta and Woo (1991). Suppose part P_2 is to be disassembled from part P_1. First, the vectors of the surface normals to P_2's mating faces with respect to P_1 are determined. Note that the vectors always point inward towards the part to be disassembled (P_2 in this case; see Figure 3.2.(a)). Next, the procedure maps all of the surface normals of the mating faces of P_2 onto an unit sphere. To determine a valid set of disassembly directions, the following is done. For each normal vector generated, a hemisphere is created by making a plane cut through the center of the unit sphere and perpendicular to the normal vector of the mating face. The hemisphere consists of all portions of the unit sphere lying to the side of the plane cut on which the unit normal resides. One hemisphere is created for each normal vector. The intersection of these hemispheres represents a space of valid disassembly directions. The disassembly directions are obtained by generating a vector from the center of the unit sphere to any point in the intersection space.

To illustrate this concept, a 2-dimensional example will be used. In the example, a unit circle will replace the unit sphere. In Figure 3.2.(a), n_1 and n_2 are the normals of the mating faces of P_2 with respect to P_1. Figure 3.2.(b) shows n_1 is mapped onto the unit circle, along with the semicircle obtained by cutting the unit circle perpendicular to the direction of n_1. This semicircle is indicated by a shaded area. Figure 3.2.(c) shows the entire unit circle along with the semicircles for both n_1 and n_2. The intersection of the two semicircles (shown most shaded in the figure) represents a space of possible disassembly directions. To generate a disassembly direction, one simply draws a vector from the center of the unit circle to any point in the most shaded region. In the

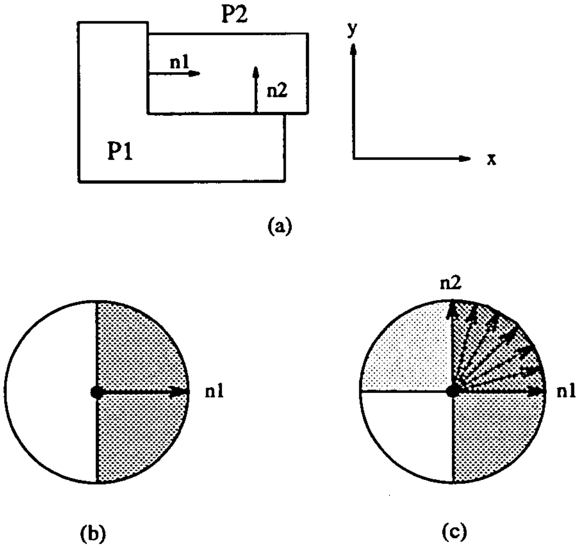


Figure 3.2: Hemisphericity of face normals

figure, the dashed lines all represent valid disassembly directions (this can be verified by examining Figure 3.2.(a)). Finding the intersection of hemispheres in 3-dimensional cases is exactly the same as for the 2-dimensional cases. As the example in Figure 3.2 shows, there can be an infinite number of disassembly directions. However, in the case where the hemispheres all intersect at a single line, there may only be one or two valid disassembly directions. These two cases can be described as follows: (a) the intersection contains an infinite number of valid disassembly directions, and (b) the intersection contains a finite number of valid disassembly directions.

(a) Case (a) was illustrated in Figure 3.2, and will now be explained in more detail. There are five different possible intersections for a disassembly direction set containing an infinite number of disassembly directions: half-plane, plane segment, plane, half-space, and space segment (see Table 3.1). For a part containing an infinite number of possible disassembly directions,

Table 3.1: Equations for representing disassembly direction sets containing an infinite number of directions

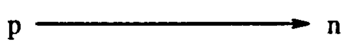
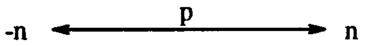
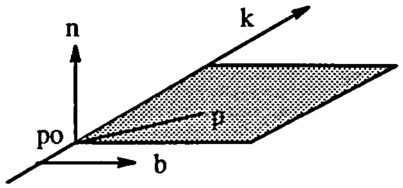
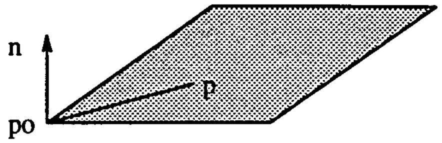
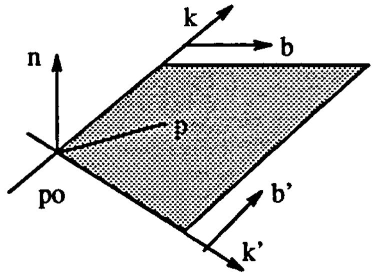
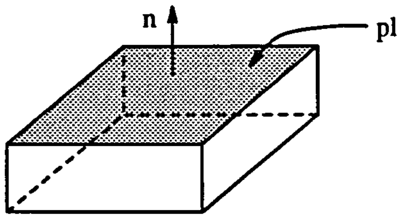
Equation	Space region represented
half-line = (p, n)	 p : a point on the line n : vector of the line
line = (p, n, -n)	 p : a point on the line n : vector of the line
half-plane = (po, p, n, k, b)	 po, p : points on the plane n : plane normal k : vector of an unbounded line ehich divides the plane into two unbounded regions b : vector perpendicular to both n and k and points to the desired plane region
plane = (po, p, n)	 po, p : points on the plane n : plane normal
plane segment = (po, p, n, (k, b), (k', b'), ...)	 region bounded by two half-planes (po, p, n, k, b) and (po, p, n, k', b')
half-space = (pl, n)	 pl : unbounded surface which divides Cartesian space into two unbounded regions n : normal of pl and points to the desired space region
space segment = ((pl, n), (pl', n'),.....)	region bounded by two half-spaces (pl, n) and (pl', n')

Table 3.2: Primary disassembly direction selection

Intersection	Primary disassembly direction (Symbols used follow Figure 3.3)
half-plane	b, k, -k
plane	any vector on the plane
plane segment	θ : angle between b and b'[a] If $0^{\circ} \leq \theta \leq 90^{\circ}$, select b, b', k, k' If $90^{\circ} < \theta < 180^{\circ}$, select $\frac{b+b'}{2}$, k, k'
half plane	n and any vector on pl
space segment	θ : angle between n and n' If $0^{\circ} \leq \theta \leq 90^{\circ}$, select n, n', and any vector on pl and pl' If $90^{\circ} < \theta < 180^{\circ}$, select $\frac{n+n'}{2}$, and any vector on pl and pl'

[a] There are two angles between two vectors, and the sum of these two angles is 360°. The smaller angle is always chosen for angle θ.

only primary disassembly directions are selected. The reason only primary disassembly directions are chosen is to cut down on the size of the solution space. Lin (1990) suggested a way to select primary disassembly directions. The modified procedure is presented in Table 3.2.

(b) Case (b) will now be explained in more detail. If the intersection of the hemispheres is a line, the valid disassembly directions are $\{n, -n\}$, where n is the unit vector of the line. If the intersection of those hemispheres is a half-line, the disassembly direction is n, where n is the unit vector of the half-line. This case is illustrated in Figure 3.3. Suppose part P_2 is to be disassembled from part P_1. In Figure 3.3, it can be seen that the three semicircles intersect along one direction n_2 in the figure. Thus only the single disassembly direction n_2 is valid.

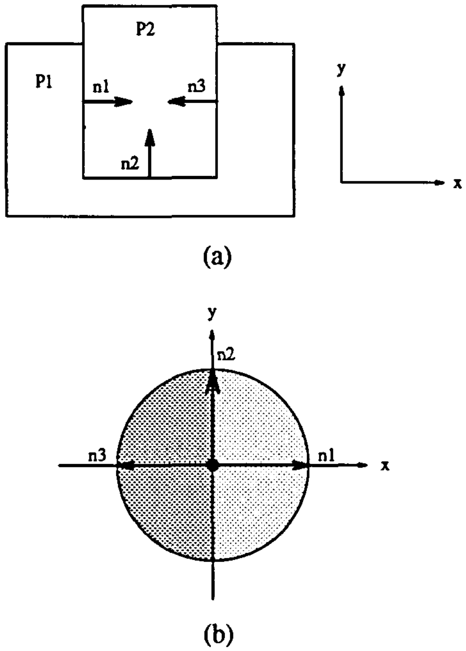


Figure 3.3: The intersection of the semicircles contains a finite number of disassembly directions

Given the above approach, it is quite easy to determine the disassembly directions for each part. The assembly directions for a part is simply the reverse of its disassembly directions.

Methodology

For a given simple-cycle product design with l liaisons, there are l - 2 types of precedence relationships to be derived:

1. Simple form

   $L_i \longrightarrow L_{j_1}$, where $i,j_1 = 1,2,\ldots,l$, and $i \neq j_1$

2. Complex form

   $L_i \longrightarrow (L_{j_1} \wedge L_{j_2})$, where $i,j_1,j_2 = 1,2,\ldots,l$, and $i \neq j_1 \neq j_2$

$\vdots$

$L_i \longrightarrow (L_{j_1} \wedge L_{j_2} \wedge \ldots \wedge L_{j_{l-2}})$, where $i, j_1, j_2, \ldots, j_{l-2} = 1, 2, \ldots, l$, and $i \neq j_1 \neq j_2 \neq \cdots \neq j_{l-2}$

An example product design, as shown in Figure 3.4, is used to illustrate the methodology. Figure 3.5 shows the liaison diagram for the product and Figure 3.3 shows the disassembly directions for each part. Since there are five parts and five liaisons in this product, three types of precedence relationships must be checked:

1. Simple form

   $L_i \longrightarrow L_{j_1}$, where $i, j_1 = 1,2,3,4,5$, and $i \neq j_1$

2. Complex form

   $L_i \longrightarrow (L_{j_1} \wedge L_{j_2})$, where $i, j_1, j_2 = 1,2,3,4,5$, and $i \neq j_1 \neq j_2$

   $L_i \longrightarrow (L_{j_1} \wedge L_{j_2} \wedge L_{j_3})$, where i, $j_1, j_2, j_3 = 1,2,3,4,5$, and $i \neq j_3 \neq j_3 \neq j_3$

The methodology is described as following:

1. To identify the simple-form precedence relationships: $L_i \longrightarrow L_{j_1}$, where $i, j_1 = 1,2,3,4,5$, and $i \neq j_1$

   Assuming that L_i and L_{j_1} are established, if L_i can be disengaged by disassembling either part associated with L_i, then $L_i \longrightarrow L_{j_1}$ does not exist; otherwise, $L_i \longrightarrow L_{j_1}$ exists. The liaison, L_{j_1}, on the right hand side of precedence sign must be maintained when L_i is being disengaged. For example, as shown in Figure 3.5 and 3.6, the existence of $L_2 \longrightarrow L_1$ is checked. L_2 can only be disengaged by disassembling P_3. If L_2 is disengaged by disassembling P_2 instead, L_1 will be disengaged too. In terms of assembly view point rather

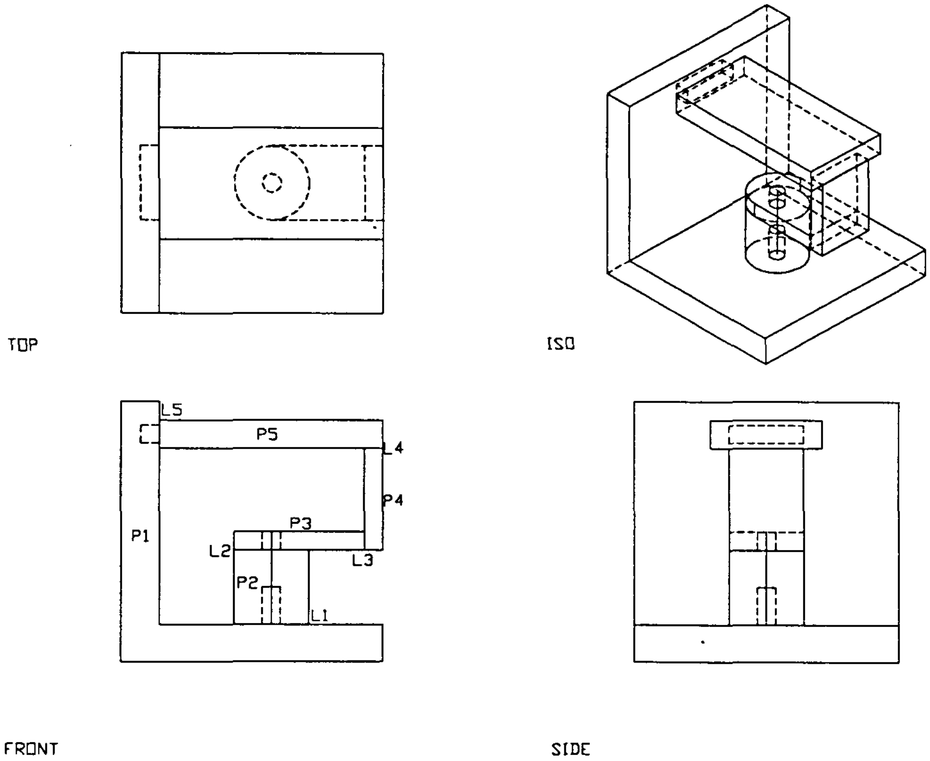


Figure 3.4: Example product with the simple-cycle liaison diagram

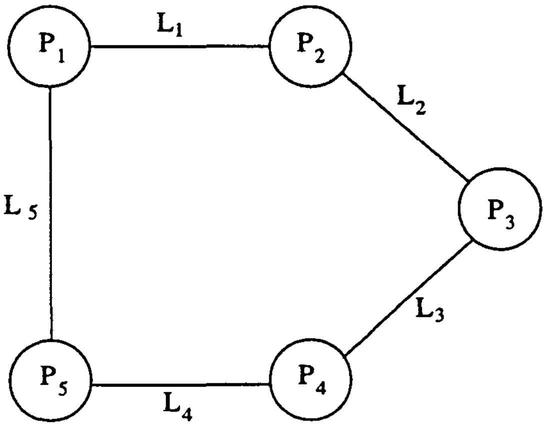


Figure 3.5: The liaison diagram of the example product shown in Figure 3.4

Table 3.3: Part disassembly directions associated with each liaison for the example product

Liaison / Part to be disassembled	L1	L2	L3	L4	L5
P1	(0,-1,0)	——	——	——	(-1,0,0)
P2	(0,1,0)	(0,-1,0)	——	——	——
P3	——	(0,1,0)	(-1,0,0) (0,1,0)(0,-1,0) (0,0,1)(0,0,-1)	——	——
P4	——	——	(1,0,0) (0,1,0)(0,-1,0) (0,0,1)(0,0,-1)	(1,0,0)(-1,0,0) (0,-1,0) (0,0,1)(0,0,-1)	——
P5	——	——	——	(1,0,0)(-1,0,0) (0,1,0) (0,0,1)(0,0,-1)	(1,0,0)

Note : The elements in each cell are the disassembly directions expressed in terms of unit directional vectors. For example, the first cell, (0,-1,0) and (0,-1,0), are the feasible disassembly directions when L1 is disengaged by disassembling P1.

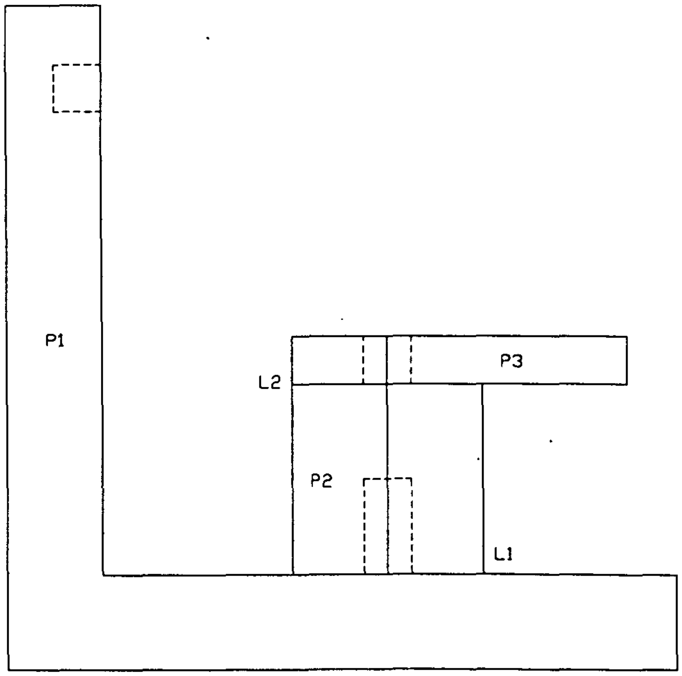


Figure 3.6: The subassembly consisting of liaison L_1 and L_2 only

than disassembly, we are testing that if L_2 can be established by inserting P_3 in its assembly directions with L_1 established. For every L_i, only those L_{j_1} adjacent to L_i need to be checked. So when checking the existence of $L_1 \longrightarrow L_k$, only $k = 2$ and k = 5 need to be checked. The cases that $k = 3$ and $k = 4$ need not to be checked. For example, L_1 is made of P_1 and P_2, and L_4 is made of P_4 and P_5. Both liaisons can be established separately without any difficulty, because they are two separate subassemblies. This is the reason that $k = 3$ and $k = 4$ need not to be checked with L_1.

2. To identify the complex-form precedence relationships

   (a) $L_i \longrightarrow (L_{j_1} \wedge L_{j_2})$, where $i, j_1, j_2 = 1,2,3,4,5$, and $i \neq j_1 \neq j_2$

   With L_i, L_{j_1}, and L_{j_2} established, if L_i can be disengaged without disengaging L_{j_1} and L_{j_2}, then $L_i \longrightarrow (L_{j_1} \wedge L_{j_2})$ does not exist. Otherwise, it exists. When disengaging L_i, the liaisons on the right hand side of precedence sign must always be maintained as well. For a simple-cycle product, this can be done by making sure that at most only one of the liaisons on the right side of precedence sign is adjacent to L_i. For example, when checking the existence of the relationship, $L_2 \longrightarrow (L_1 \wedge L_3)$, it is clear that both L_1 and L_3 are adjacent to L_2 (see Figure 3.4 and 3.5). L_2 can be disengaged by disassembling either P_2 or P_3. No matter which one is disassembled, either L_1 or L_3 would be disengaged and the Boolean value of $(L_1 \wedge L_3)$ would be 0. Then, $L_2 \longrightarrow (L_1 \wedge L_3)$ is no longer valid.

   In addition, two conditions must be met for the liaisons on the right hand side of precedence sign:

   i. They must form a tree.

ii. Exactly one liaison among them must be adjacent to L_i.

For example, when the precedence relationships between L_3 and two other liaisons are checked, only $L_3 \longrightarrow (L_4 \wedge L_5)$ and $L_3 \longrightarrow (L_2 \wedge L_1)$ need to be checked. For $L_3 \longrightarrow (L_1 \wedge L_5)$, because the parts associated with L_1 and L_5 and the parts associated with L_3 are two separate subassemblies, they can be constructed separately without any difficulty. Therefore, there is no need to check the existence of this type of precedence relationships.

(b) $L_i \longrightarrow (L_{j_1} \wedge L_{j_2} \wedge L_{j_3})$, where i,j_1,j_2,j_3 = 1,2,3,4,5 and $i \neq j_1 \neq j_2 \neq j_3$

With L_i, L_{j_1}, L_{j_2}, and L_{j_3} established, if L_i can be disengaged without disengaging any of L_{j_1}, L_{j_2}, and L_{j_3}, then $L_i \longrightarrow (L_{j_1} \wedge L_{j_2} \wedge L_{j_3})$ does not exist. Otherwise, this precedence relationship exists. When disengaging L_i, the liaisons on the right hand side of the precedence sign cannot be disengaged either, and the heuristics presented in 2.(a) should be followed as well.

If $L_i \longrightarrow A$ exists (A could be a single liaison or a logic combination of liaisons using "$\wedge$" only), there is no need to check whether or not $L_i \longrightarrow B$ exists, where A is a subset of B. For example, suppose that $L_1 \longrightarrow L_5$ exists. There is no need to check if $L_1 \longrightarrow (L_5 \wedge L_4)$ exists. If $L_1 \longrightarrow (L_5 \wedge L_4)$ is checked, it will exist too. Because of $L_1 \longrightarrow L_5$, no matter L_4 is established or not, L_1 cannot be established. However, it is incorrect that both of these two precedence relationships exist. Suppose that $L_5 = 1$, and $L_4 = 0$. According to $L_1 \longrightarrow (L_5 \wedge L_4)$, the Boolean value on the right hand side is 0 ($1 \wedge 0 = 0$). Thus, L_1 can be established. But according to $L_1 \longrightarrow L_5$, the Boolean value on the right side is 1. That means L_1 cannot be established. Then, a contradiction is found between these two precedence relationships. In other words, these two precedence relationships cannot both exist. Therefore, If $L_i \longrightarrow A$ exists, it is not necessary to check whether or not $L_i \longrightarrow B$ exists, where A is a subset of B.

There is also no need to check if $L_i \longrightarrow (L_{j_1} \wedge L_{j_2} \wedge L_{j_3} \wedge L_{j_4})$ exists. In this case, there must be exactly two liaisons on the right side of the precedence sign adjacent to L_i for the example product with five liaisons. This case violates the heuristics described in 2.(a) that exactly one liaison among the liaisons on the right hand side of the precedence sign must be adjacent to L_i. Therefore, this is the reason that only l - 2 types of precedence relationships need to be derived with l liaisons in a product design.

Liaison Disengagement

The proposed methodology uses a disassembly approach to check the possibility of liaison disengagement. A liaison can be disengaged only when the associated part can be removed freely without colliding other parts in the disassembly directions. For example, as shown in Figure 3.6, if we want to check the existence of the relationship $L_2 \longrightarrow L_1$ by disassembling P_3 in its disassembly directions, we need to check if P_3 will collide with P_1 and with P_2. An algorithm using swept volume technique is developed to identify the possible collisions. Using this algorithm, all of the possible collisions can be detected between a moving object and a static object. The algorithm is summarized in the following:

1. Assuming that part P_1 is going to be removed at direction D and P_2 is the static part. For each part, find the extent (minimum and maximum coordinates values) in x, y, and z axes, then construct a rectangular solid for each part.

2. Check the interference between the two rectangular solids. Two cases may result from the test: (a) interference exists between the two solids, or (b) interference does not exist.

   (a) No interference is detected between the two rectangular solids

   For example, as shown in Figure 3.7, we want to know if P_2 will collide with P_1 at the +x direction. Two rectangular solids are formulated for the two parts, and there is no interference between these two solids (see Figure 3.8). The following three conditions must be checked next:

   Condition 1: P_2 is completely behind P_1 with respect to +x direction as shown in Figure 3.9.

   Condition 2: P_2 is completely above or below P_1 as shown in Figure 3.10.

   Condition 3: P_2 is completely on the right or left side of P_1 as shown in Figure 3.11.

   If any one of these three conditions is satisfied, P_2 will not collide with P_1 in +x direction. If none of them is satisfied (see Figure 3.8), then the swept volume technique described in (b) is used to test if P_2 will collide with P_1 in +x direction. Creating swept volumes as solids and checking interferences are quite time-consuming. The three tests stated above are used to avoid unnecessary swept volume creations and interference checkings.

   (b) Interference is detected between the two rectangular solids

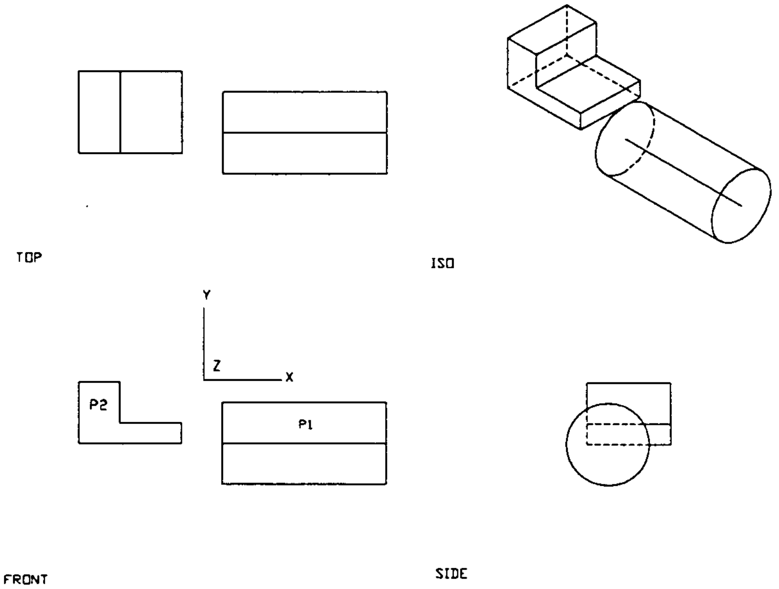


Figure 3.7: Part P_2 is to be removed at +x direction, while part P_1 is stationary

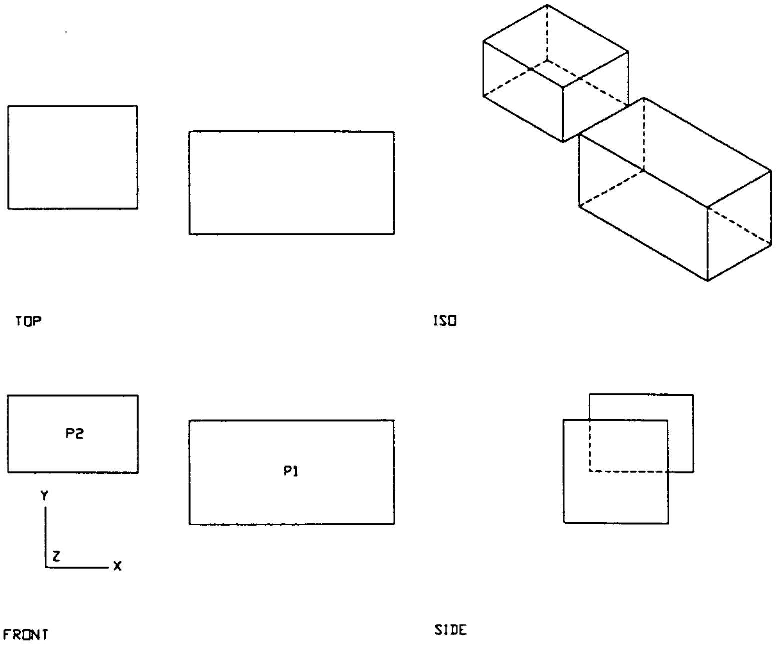


Figure 3.8: The two rectangular solids for parts P_1 and P_2 do not intersect each other

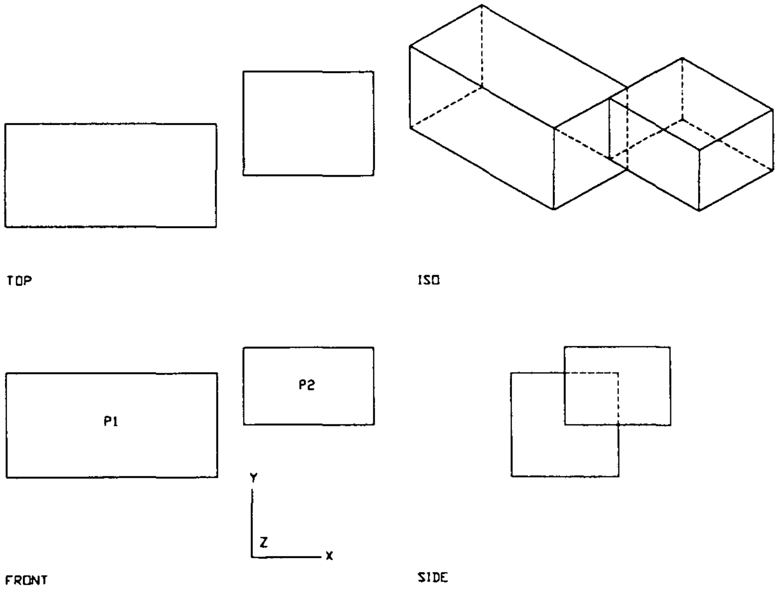


Figure 3.9: The rectangular solid for part P_2 is completely behind the solid for part P_1

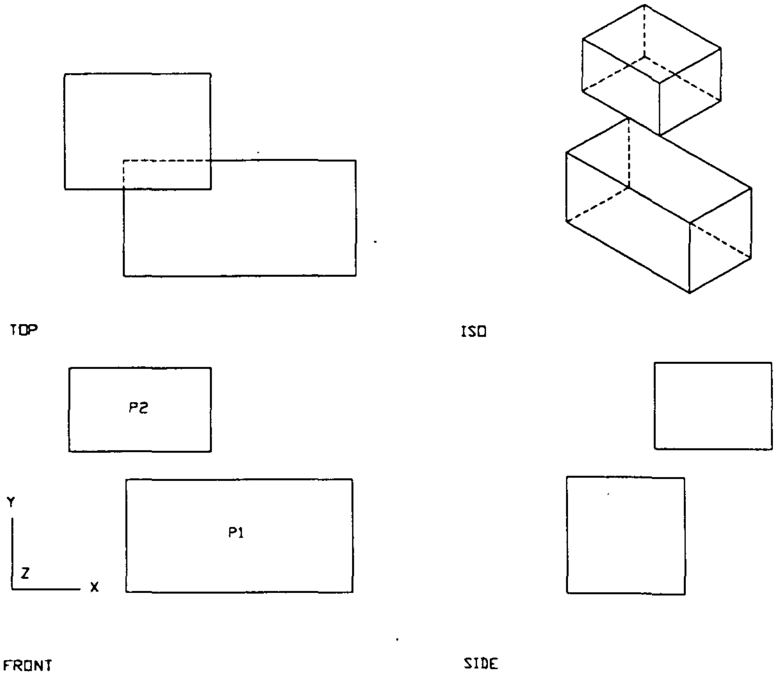


Figure 3.10: The rectangular solid for part P_2 is completely above the solid for part P_1

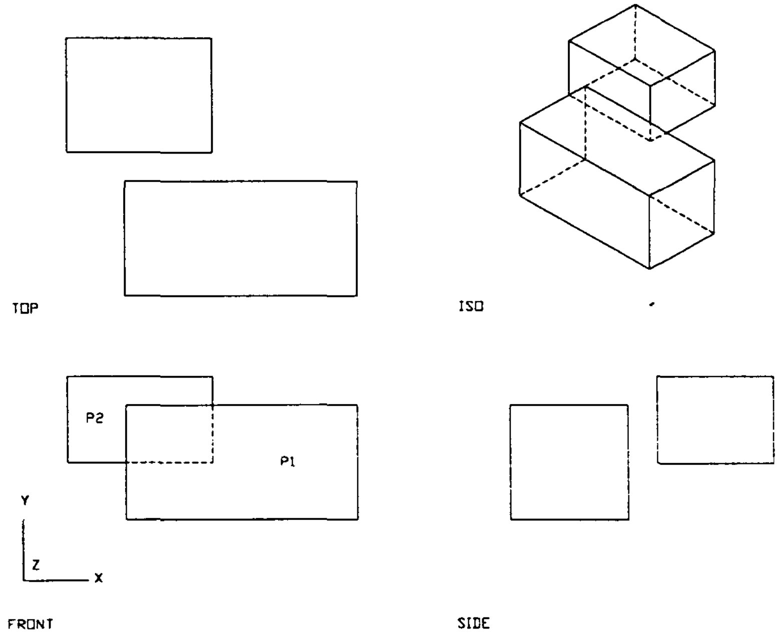


Figure 3.11: The rectangular solid for part P_2 is completely on the right side of the soid for part P_1

As shown in Figure 3.12, we want to know whether or not P_1 will collide with P_2 in +x direction. Two rectangular solids are formulated for the two parts (see Figure 3.13), and there is an interference between them. Although the two solids intersect, it does not mean the actual part movement will interfere each other. Further test is needed to clarify the possibility. This is done by translating every facet of P_1 in +x direction. The volume each facet of P_1 sweeps through becomes a swept solid. If any swept solid has an interference with P_2, then P_1 will collide with P_2 when P_1 is removed at the +x direction. The distance at which the facets are translated must be large enough to cover all the potential parts which might be collided by moving P_1, so a reasonably large value can be chosen arbitrarily for this distance. The procedure to create a swept solid

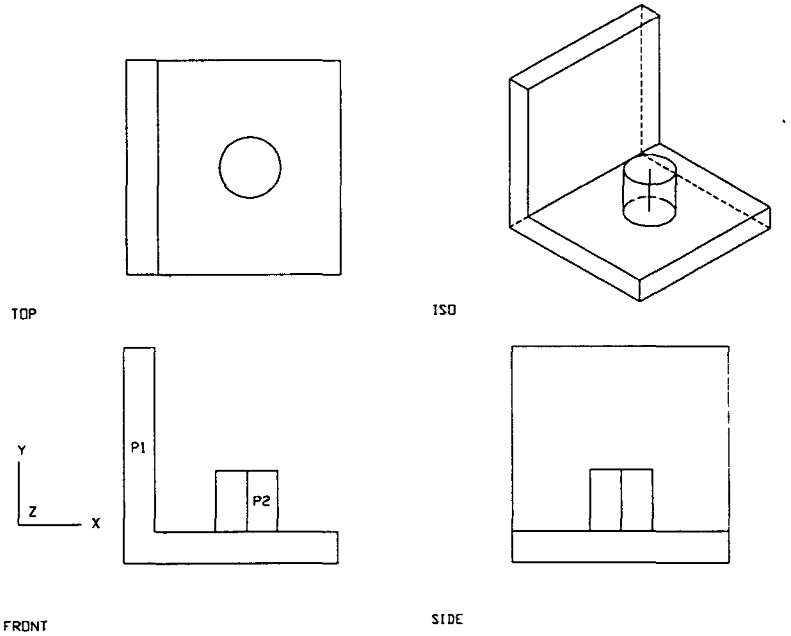


Figure 3.12: Part P_1 is to be removed at +x direction, while part P_2 is stationary

is described as followed:

i. Find the projection of every facet of P_1 on a plane which has a normal vector D (D is the disassembly direction for P_1). If the projection of a facet is a line, the facet is ignored.

ii. For those facets which are perpendicular to direction D, extrude their projections from where the facets are, then go to step v (see Figure 3.14).

iii. For those facets which are not perpendicular to direction D, additional care must be taken during extrusion. This is illustrated with an example of extruding a cylindrical surface in a direction perpendicular to the axis of the cylinder. As shown in Figure 3.15, P_1 is a cylindrical surface which will be extruded in direction D, and Figure 3.15

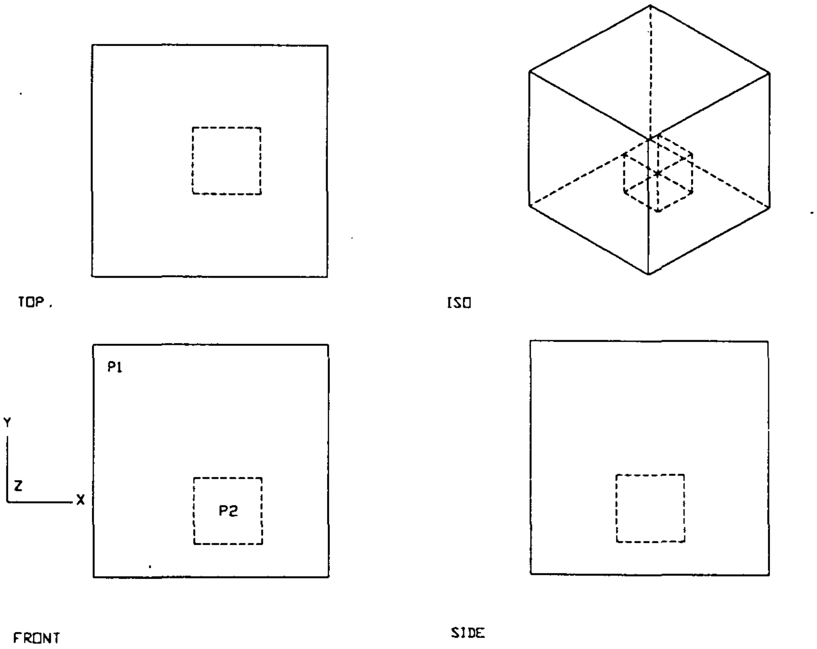


Figure 3.13: The two rectangular solids for parts P_1 and P_2 intersect each other

shows its top view. The cylindrical face is represented by 16 facets and none of these facets is perpendicular to direction D. For any facet i as shown in Figure 3.16, its projection is extruded in direction D to create a swept solid and SP2 is chosen as the starting position to extrude the projection.

iv. Define facet i as the cutting plane to cut the swept solid that was just created. Discard the solid on the right side of facet i, and the solid on the left side of facet i is exactly the swept solid we need. This swept solid is shown with a filled area in Figure 3.16.

v. Check interferences between P_2 and the swept solids. If there is any interference, then P_1 will collide with P_2 in direction D.

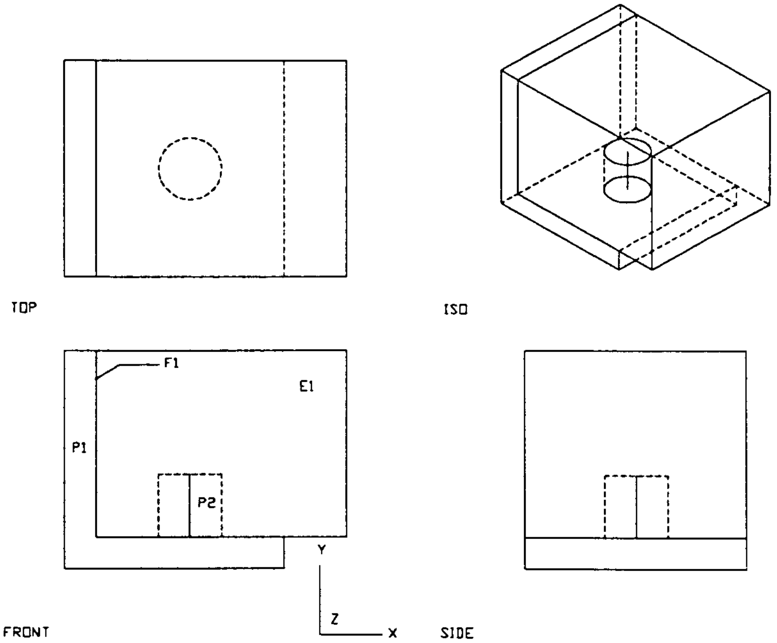


Figure 3.14: Sweep facet F1 of P_1 in +x direction to create a swept volume for collision detection

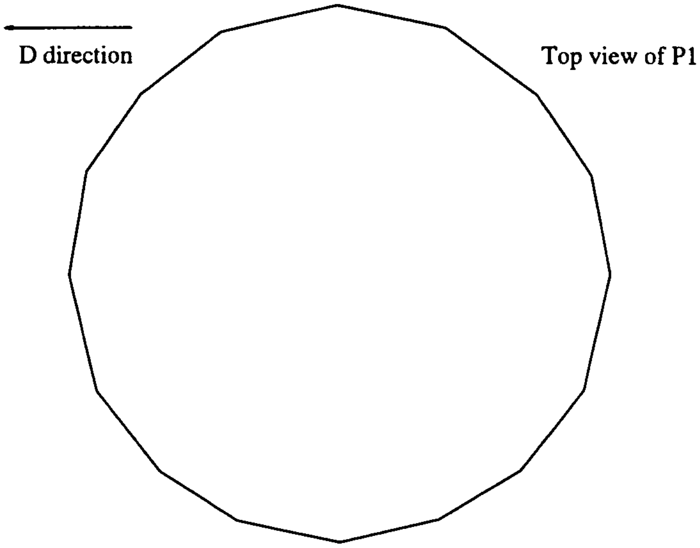


Figure 3.15: Top view of P_1

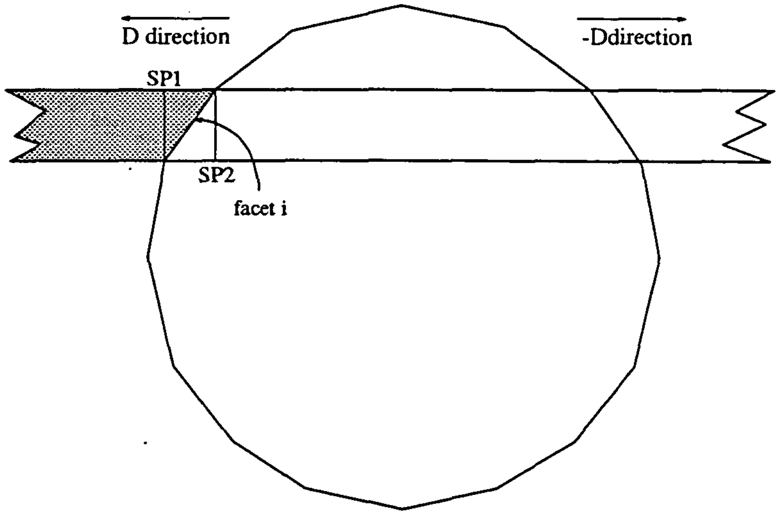


Figure 3.16: Sweep facet i in -x direction to create a swept volume for collision detection

CHAPTER 4. IMPLEMENTATION AND RESULTS

This chapter presents the implementation of the methodology discussed in the previous chapter. A section on assembly sequence generation is also included. The implementation serves two purposes: 1) to validate the proposed methodology, and 2) to demonstrate its feasibility. The general procedures of the program will be described. Results for the example product will be provided to validate the correctness of the methodology and the program.

Implementation

The proposed methodology has been implemented in the UNIX environment using the C programming language. The geometric assembly model was created using I-DEAS on a DEC5000/200 workstation. The program interfaces with the I-DEAS[1] (SDRC, 1990) solid modeling system during execution. The I-DEAS system provides a static interference checking function, which was employed to implement the collision detection using the swept volume technique. As shown in Figure 1.1, the inputs of the program are:

1. The universal files of the solid model that contain the geometric information describing the product design.

[1] Integrated Design Engineering Analysis Software (I-DEAS) is a software product of Structural Dynamics Research Corporation (SDRC), Milford, OH 45150

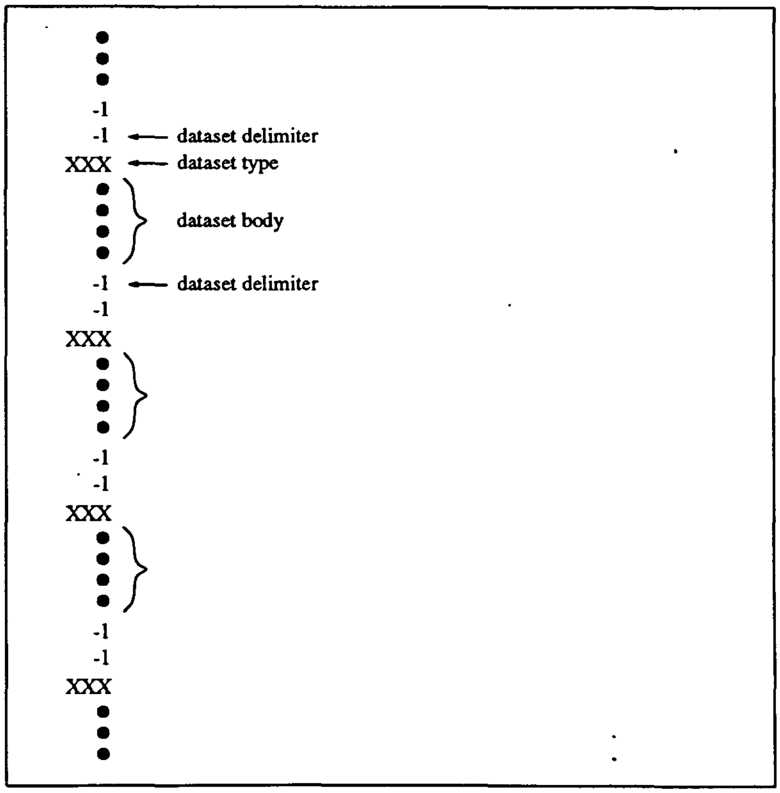


Figure 4.1: Basic structure of a universal file

2. The liaison diagram for the product design.

3. The disassembly directions for each part.

These three inputs will be described in the following sections.

Universal File Structure

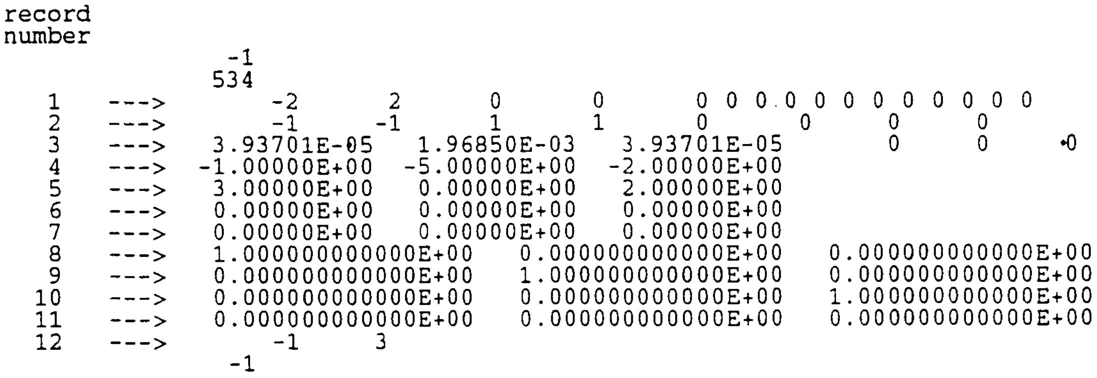
All geometric information describing the product design is retrieved from the universal files. The files are written in ASCII characters. Thus, a user created program such as the one described in this thesis can directly access the information in a universal file. Each universal file is a sequentially formatted file with records having a maximum length of 80 characters. The basic structure of a universal file is shown in Figure 4.1. A universal file is divided into sections called datasets. The first record of each dataset is a dataset delimiter. This is a line containing a minus sign in column 5 and a 1 in column 6. The second record of the dataset contains the dataset type indicating the type of data included in the dataset, such as a transformation matrix, precise surface information of a part, etc. The second record is a number in the range 1 through 32767 right justified in columns 1 through 6. Following the dataset type record is the body of the dataset which contains data dependent on the dataset type. The final record of the dataset contains a delimiter line containing a minus sign in column 5 and a 1 in column 6. Figure 4.2 shows an example of a type 534 dataset in the part P_2's universal file. The first row and the last row are the delimiters. I-DEAS uses the Constructive Solid Geometry (CSG) method to store a list of objects and operations required to define a part. The second row, dataset type 534, means this dataset contains some geometric information about part P_2 after an operation is done on P_2. An operation performed on an object can include: cut, join, scale, reflect, etc. If none of these operations is performed on a part, then there will be no type 534 dataset in its universal file. In other words, this part itself is a primitive. Primitives defined in I-DEAS include block, cone, cylinder, sphere, and tube. If a final geometry of a part comes from a series of operations on a primitive, there will be at least one type 534 dataset in its universal file. Each operation performed on the primitive corresponds to one type 534 dataset, and the sequence of these datasets in the universal file corresponds to the sequence of operations performed on the primitive. Since only the geometric information about the final part geometry is needed for the program, only the last of the type 534 datasets is retrieved. Figure 3.4 shows that P_2 is constructed from a main cylinder via two operations. The first operation joins a smaller cylinder to the top of the main cylinder. The second operation cuts a cylinder shaped cavity out of the bottom of the main cylinder. Thus, there will be two type 534 datasets, corresponding to the two operations in P_2's universal file. Figure 4.2 shows the second of these datasets, which contains the information about P_2's final geometry. The program requires only part of the data in a type 534 dataset. For instance, the information necessary from the dataset in Figure 4.2 is:

- Record 4: Field 1-3 → First point of Brep bounding box. (min)
- Record 5: Field 1-3 → Second point of Brep bounding box. (max)
- Record 8: Field 1-3 → Rotation information (element A-C).
- Record 9: Field 1-3 → Rotation information (element D-F).
- Record 10: Field 1-3 → Rotation information (element G-I).
- Record 11: Field 1-3 → Translation information (element J-L).

$$\begin{bmatrix} A & D & G & J \\ B & E & H & K \\ C & F & I & L \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} XOLD \\ YOLD \\ ZOLD \\ 1 \end{bmatrix} = \begin{bmatrix} XNEW \\ YNEW \\ ZNEW \\ 1 \end{bmatrix}$$

From the list, records 4 and 5 give the coordinates of the diagonal vertices of the rectangular box of a part. Records 8-11 (i.e. elements A-L) are used to form a 4×4 transformation matrix with homogeneous coordinates (Mortenson, 1985). The matrix is used to map a part from the origin to its final position.

There are three types of datasets used by the program, and they are:



```
record
number
                 -1
                534
  1     --->        -2        2        0        0        0 0 0 0 0 0 0 0 0 0 0 0
  2     --->        -1       -1        1        1        0        0        0        0
  3     --->    3.93701E-05   1.96850E-03   3.93701E-05         0        0        0
  4     --->   -1.00000E+00  -5.00000E+00  -2.00000E+00
  5     --->    3.00000E+00   0.00000E+00   2.00000E+00
  6     --->    0.00000E+00   0.00000E+00   0.00000E+00
  7     --->    0.00000E+00   0.00000E+00   0.00000E+00
  8     --->    1.000000000000E+00   0.000000000000E+00   0.000000000000E+00
  9     --->    0.000000000000E+00   1.000000000000E+00   0.000000000000E+00
 10     --->    0.000000000000E+00   0.000000000000E+00   1.000000000000E+00
 11     --->    0.000000000000E+00   0.000000000000E+00   0.000000000000E+00
 12     --->        -1        3
                 -1
```

Figure 4.2: The type 534 dataset in P_2's universal file

1. Dataset type 537

   Dataset type 537 has the same format as dataset type 534, and is retrieved by the program only when dataset type 534 is not written. This occurs if a part itself is a primitive. Primitives have no modeling operations such as cut or join performed upon them. In this case, the necessary information is retrieved from type 537 dataset.

2. Dataset type 534

   If a part itself is not a primitive (e.g. modeling operations are performed upon it), its universal file must contain at least one type 534 dataset. As mentioned before, only the last one needs to be retrieved.

3. Dataset type 544

   When I-DEAS shows objects on screen, it uses Boundary Representation (BREP) to represent objects. The type 544 dataset contains three pieces of information that will be retrieved by the program: 1) the coordinates of all the boundary points, 2) number of facets in the object, and 3) which points each facet is made of. As mentioned in the previous chapter, facet information is needed in order to do collision detections.

Internal Representation of a Liaison Diagram

A liaison diagram is characterized as a network graph in this research. In general, there are two main representations of graphs: 1) adjacency matrix, and 2) adjacency list. Since the liaison diagrams discussed in this research are of simple-cycle structure, a double linked list data structure as shown in Figure 4.3 is used to represent the liaison diagram. Each data set is composed of three integer values. Two of the values represent part numbers of the two parts associated with a liaison, and the third gives the liaison number. As shown in Figure 4.3, the first value in each data set is the part number of the first part in the liaison, the second value in the set contains the liaison number, and the third value in the set is the part number of the second part in the liaison. The leftmost set with digits (1, 1, 2) will now be used to illustrate the format of a data set in Figure 4.3. The middle digit, 1, is the liaison number. The right digit, 2, is the right part number with respect to L_1 in the liaison diagram. The left digit, 1, is the left part number with respect to L_1. The way to determine the left part and the right part with respect to a liaison is to imagine standing inside the liaison diagram (cycle) (see Figure 3.5). Look at one of the liaisons, say L_i, the part number appearing on the right of L_i will be the right digit of i, and the part number appearing on the left of L_i will be the left digit of i. Any two liaisons adjacent to each other in the liaison diagram must have their respective data set adjacent as well. The input file format of liaison diagram is described in the Appendix A.

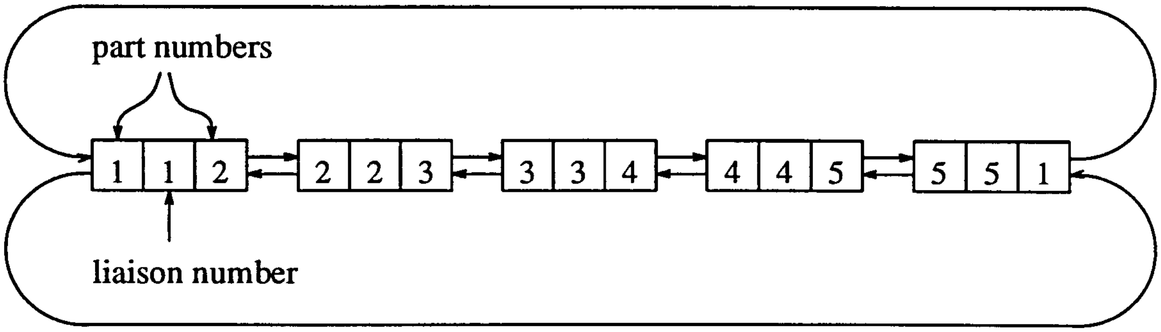


Figure 4.3: Data structure representation of the liaison diagram for the example product

Internal Representation of the Disassembly Directions

A representation, as shown in Figure 4.4, similar to adjacency list is used to store disassembly directions. Each part is associated with a linked list consisting of all of the disassembly directions of the part. Each data set contains the liaison numbers of the two liaisons associated with a disassembly direction, along with the disassembly direction. The first two values in a data set are the liaison numbers involved in the disassembly direction. The third, fourth, and fifth values are the x, y, and z components of the vector of the disassembly direction. For example, the first data set (2, 3, 0, 1, 0) for part 3 shows that disassembly direction (0, 1, 0) is associated with both liaison 2 and 3. The second data set of part 3 shows that direction (-1, 0, 0) for part 3 is associated with liaison 3 only, and so forth. For a simple-cycle liaison diagram, one disassembly direction can be associated with at most two liaisons. Thus, when a disassembly direction is associated with only one liaison, one of the first two values in a data set is set to zero. The input file format of disassembly directions is described in the Appendix A.

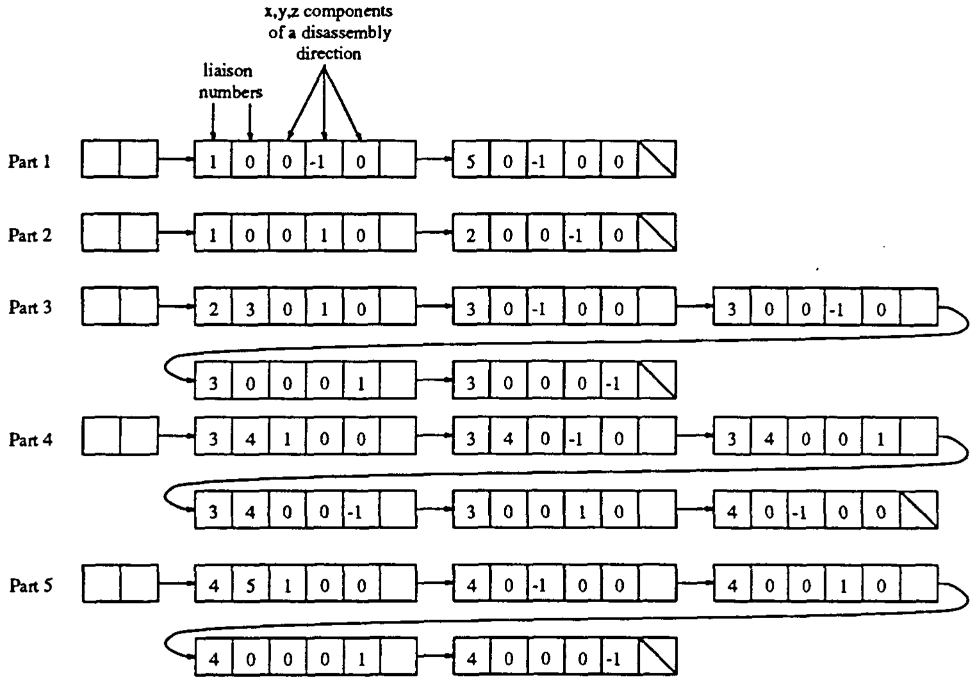


Figure 4.4: Data structure representation for the disassembly directions

Program File of I-DEAS Software

A program file is an external file of I-DEAS commands that can be built from within I-DEAS or as a text file from outside I-DEAS and executed at a later time. Once created, the program file can be used to instruct I-DEAS to execute the commands recorded in the file. The file performs in sequence. Because the file is saved as a text file, it can be created by a text editor or using any programming language (C, Fortran, etc.). Program files are especially useful if you have a long sequence of commands that are used often. The collision detection algorithm described in the previous chapter was implemented using a program file to take advantage of embedded functions of I-DEAS software. Figure 4.5 shows an example of a simple program file. Every line in a program file is started by a signal character consisting of one letter and one space followed by a colon or two letters followed by a colon. There are several signal characters defined in I-DEAS. In this research, there are only two signal characters used:

K :	Normal keyboard input
? :	Interactive interrupt

As shown in Figure 4.5, the keyboard input signal (K :) is always followed by a command. The interactive input signal (? :) will stop the automatic execution of program file commands. When a program file is running, it will stop at the point where the (? :) is entered and wait for user's input. After the required information is entered, the program file will automatically continue. In this implementation, (? :) is used only to pause the program file, so users do not need to input any information. Once the program file pauses, the interference checking result will be output by I-DEAS to the file called **IN_CH.dat** which has been specified by the C program. Then the C program continues to read **IN_CH.dat** in order to get the interference information. To continue the program file, users just need to hit 'RETURN' key instead of inputting any information. The details of running the program are described in the Appendix A. Figure 4.6 shows all of the potential collisions for all the parts moving in their disassembly directions. This output is used by the program to extract precedence relationships.

Implementation Using the Program File

The data structures of liaison diagrams, disassembly directions, universal files, and program files have been explained in the previous sections. In this section, part of the program file for the example product shown in Figure 3.4 is extracted to explain the implementation using I-DEAS program file. Before the implementation

```
K :/CREATE
K :BLOCK
K : 14.000 14.000 14.000
K :/MANAGE
K :STORE
K :B1
? :
```

Figure 4.5: Simple program file

P1 → (P2,-1, 0, 0)
(P5, 0,-1, 0)

P2 → (P1, 0,-1, 0)
(P3, 0, 1, 0)
(P5, 0, 1, 0)

P3 → (P1,-1, 0, 0)
(P1, 0,-1, 0)
(P2,-1, 0, 0)
(P2, 0,-1, 0)
(P2, 0, 0, 1)
(P2, 0, 0,-1)
(P5, 0, 1, 0)

P4 → (P1, 0,-1, 0)
(P1,-1, 0, 0)
(P2, -1, 0, 0)
(P3,-1, 0, 0)
(P5, 0, 1, 0)

P5 → (P1,-1, 0, 0)
(P1, 0, 1, 0)
(P1, 0, 0, 1)
(P1, 0, 0,-1)

Note : P1 → (P2,-1, 0, 0) means P1 will collide with P2 at the (-1, 0, 0) direction

Figure 4.6: Collisions detected for each part of the example product

is discussed, a further introduction to I-DEAS is needed. I-DEAS is made up of a number of "Families", each subdivided further into "Tasks", all executed from a common menu and sharing a common database. The main families are: Solid Modeling, Finite Element Modeling & Analysis, System Dynamics, Test, Drafting, and Manufacturing. In this research, only two tasks in the Solid Modeling family are used: Object Modeling, and Assembly Modeling. The Object Modeling task is the foundation of I-DEAS, since the solid object geometry that is created here is shared by many other applications. This is where the initial design is created. The Assembly Modeling task is used to create complex systems from the objects created in the Object Modeling task. The static interference checking function is embedded in this task. A model of an assembly is called a system in the Assembly Modeling task. In a system, a model of a part is called a component. If the same component is used more than once in a system, the component is not duplicated. Instead, instances of the component are created. An instance is simply a technique for minimizing the size of the database. Instead of duplicating a component, each instance provides a pointer from the component to the system in which it is used. In this way, only one version of the component exists, even though the component is instanced many times in many different systems. The pointer mentioned above includes orientation data to describe the orientation of the instance with respect to the system. When a component is created in the Assembly Modeling task, another type of pointer is also created. This pointer associates an object with the component. This object defines the geometry of the component. Because a pointer is used, the object's geometry is not duplicated, no matter how many times the component is created. This pointer does not include orientation information, therefore the component space is coincident with the object space. Figure 4.7 shows the relationships for a component and its

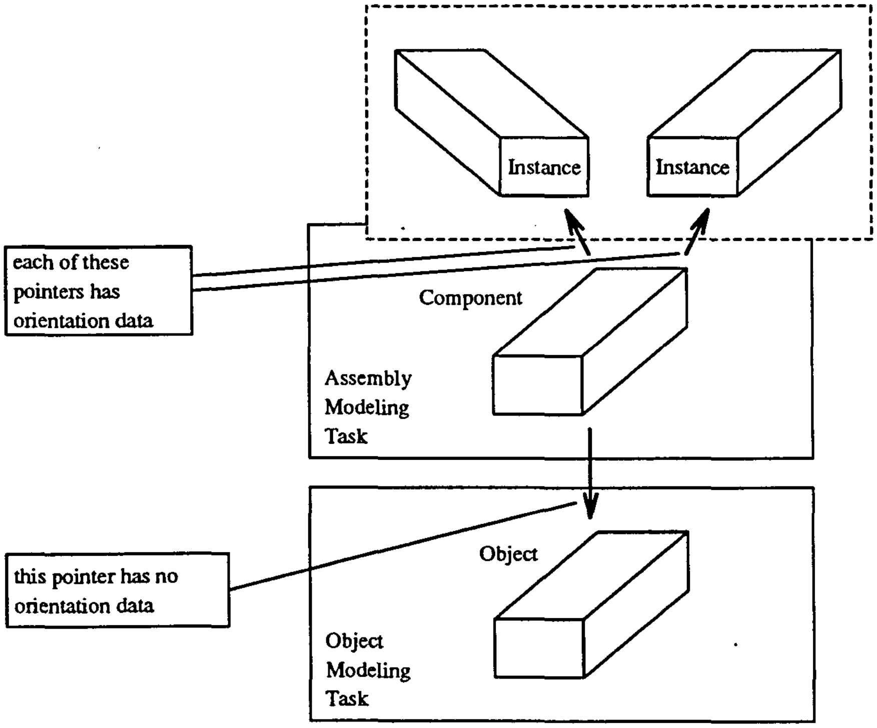

Instance
Instance
each of these pointers has orientation data
Component
Assembly Modeling Task
this pointer has no orientation data
Object
Object Modeling Task

Figure 4.7: The relationships for a component and its instances and object

instances and object. For the example product discussed in this thesis, since each component (part) is used exactly once, each component has only one instance.

The name of the program file generated by the C program has been specified as **cycle.prg** by the C program itself. Suppose that P_1 is going to be removed in (-1, 0, 0) direction and the collision between P_1 and P_2 is checked. As discussed in the previous chapter, two rectangular solids for P_1 and P_2 have to be constructed first. Then the existence of static interference between these two solids has to be checked. This is done in I-DEAS by a segment of **cycle.prg** shown in Figure 4.8. In this segment of the program file **cycle.prg**, two rectangular solids for P_1 and P_2 are created first in Object Modeling task. In order to do the interference checking

```
line number

     1    --->    K :/CREATE
     2    --->    K :BLOCK
     3    --->    K : 14.000  14.000  14.000
     4    --->    K :/MANAGE_STORED
     5    --->    K :STORE
     6    --->    K :B1
     7    --->    K :
     8    --->    K :/CREATE
     9    --->    K :BLOCK
    10    --->    K :  4.000   5.000   4.000
    11    --->    K :/ORIENT
    12    --->    K :TRANSLATE
    13    --->    K :  1.000  -2.500   0.000
    14    --->    K :/MANAGE_STORED
    15    --->    K :STORE
    16    --->    K :B2
    17    --->    K :
    18    --->    K :/TASK
    19    --->    K :ASSEMBLY_MODELING
    20    --->    K :/CREATE
    21    --->    K :COMPONENT_&_INSTANCE
    22    --->    K :
    23    --->    K :B1
    24    --->    K :
    25    --->    K :B1
    26    --->    K :
    27    --->    K :
    28    --->    K :DONE
    29    --->    K :/CREATE
    30    --->    K :COMPONENT_&_INSTANCE
    31    --->    K :
    32    --->    K :B2
    33    --->    K :
    34    --->    K :B2
    35    --->    K :
    36    --->    K :DONE
    37    --->    K :DRAW
    38    --->    K :/LIST
    39    --->    K :CHECK_INTERFERENCE
    40    --->    K :VOLUME_COMPUTE_SW
    41    --->    K :ON
    42    --->    K :COMPONENT_CHECK
    43    --->    K :LABEL
    44    --->    K :B1
    45    --->    K :
    46    --->    K :LIST_LAST_RESULTS
    47    --->    K :YES
    48    --->    K :IN_CH
    49    --->    K :YES
    50    --->    ? :
```

Figure 4.8: Part of the program file **cycle.prg**(line numbers and arrows do not exist in the program file, but are used here for clarification)

Table 4.1: Explanation of part of the program file **cycle.prg** shown in Figure 4.8

Line number	Explanation for the I-DEAS commands
1 - 7	The circumscribed rectangular solid for P1 is created, and is stored in I-DEAS as B1.
8 - 17	The circumscribed rectangular solid for P2 is created, and is stored in I-DEAS as B2.
18 - 19	Switch the task from the Object Modeling to the Assembly Modeling.
20 - 28	A component is created for B1 just constructed in Object Modeling task, and is named B1.
29 - 37	A component is created for B2 just constructed in Object Modeling task, and is named B2.
38 - 49	Check the interference between components B1 and B2, and then output the interference checking result to the file IN_CH.dat.
50	Pause the program file cycle.prg.

between the two solids just created, we have to switch the task from Object Modeling task to Assembly Modeling task, and create one component for each solid. Then the interference is checked between these two components and the result is output to the file **IN_CH.dat**. The details of this segment of **cycle.prg** are explained in Figure 4.1.

Although the objects B1 and B2 in the Object Modeling task have the same name as the components B1 and B2 in the Assembly Modeling task, I-DEAS regards these two names as different. For example, the B1 in Object Modeling task is not the same as the B1 in Assembly Modeling task because these two names are defined in two different tasks. Since the static interference checking function is embedded in the Assembly Modeling task, the program must switch tasks from Object Modeling to Assembly Modeling when interference checking is needed.

The source code of the C program is listed in Appendix B.

Sequence Generation

Based on the extracted precedence relationships, a complete set of valid assembly sequences can be generated. Since the proposed methodology for deriving the precedence relationships is based on the assumption that one liaison is disengaged at a time, the generated sequences based on the extracted precedence relationships will be valid if only one liaison is established at a time. Before (l - 2) liaisons are established, a liaison assembly sequence is generated by choosing one liaison at a time. Every chosen liaison must be checked to determine if the establishment of this liaison will violate any of the precedence relationships. If the establishment of this liaison violates any of the precedence relationships, another unestablished liaison must be selected to be checked. So, when only (l - 2) liaisons are established without violating any of the precedence relationships, every sequence ranging from the first to the (l-2^{nd}) liaison is valid.

Once (l - 2) liaisons have been successfully established, the last two unestablished liaisons will be established at the same time because of the following two reasons: 1) all of the parts are rigid, and 2) the liaison diagram is in the simple-cycle structure. Therefore, additional care must be taken, and two cases need to be considered: 1) the (l - 2) liaisons established are made of two subassemblies, and 2) the (l - 2) liaisons established are made of only one subassembly.

1. The (l - 2) liaisons established are made of two subassemblies.

   Combining the two subassemblies will establish two liaisons, say L_i and L_j, at the same time. In this case, L_i and L_j are not adjacent to each other in the liaison diagram. A simple-cycle example is shown in Figure 4.9 to illustrate

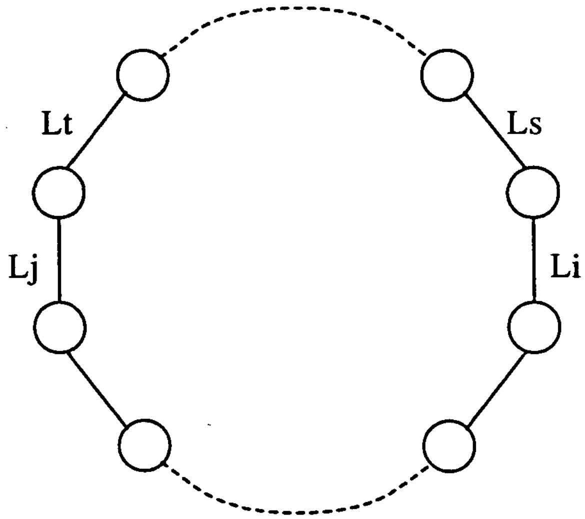


Figure 4.9: An example of simple-cycle liaison diagram

this special case. If the following two precedence relationships exist, then the product cannot be assembled successfully.

(a) $L_i \longrightarrow (L_s \wedge \ldots \wedge L_t \wedge L_j)$

(b) $L_j \longrightarrow (L_t \wedge \ldots \wedge L_s \wedge L_i)$

The liaisons on the right hand side of the precedence sign of the two precedence relationships above include all of the intermediate liaisons between L_s and L_t. For example, suppose L_3 and L_4 have been established. Now L_5 is going to be established, but doing so will violate the precedence requirement $L_1 \longrightarrow L_5$. Thus L_1 has to be established before L_5. With L_1, L_3, and L_4 established, the next step is to combine the two subassemblies, one made of P_1 and P_2, and the other one made of P_3, P_4, and P_5. Combining them will establish two liaisons, L_2 and L_5, at the same time. But doing so will violate the precedence requirements $L_2 \longrightarrow (L_3 \wedge L_4 \wedge L_5)$ and $L_5 \longrightarrow (L_4 \wedge L_3 \wedge L_2)$ (see Figure 3.5), and the product cannot be assembled successfully.

2. The (l - 2) liaisons established are made of only one subassembly.

   In this case, the subassembly consists of (l - 1) parts, so the last part inserted to this subassembly will also establish two liaisons at the same time. The last two unestablished liaisons must be adjacent to each other in the liaison diagram. The final part can always be inserted regardless of violating any precedence relationship or not.

   Case 1 above explains why there is no need to check if the establishment of any one of the last two liaisons will violate any precedence relationships. For example, after L_1, L_2, and L_3 have been established, the last part (P_5 in this case) inserted will establish L_4 and L_5 at the same time. According to the statement in case 1, if any one of the following two set of precedence relationships exist, the product cannot be assembled successfully (see Figure 3.5).

   (a) $L_4 \longrightarrow L_5$
   $L_5 \longrightarrow L_4$

   (b) $L_4 \longrightarrow (L_3 \wedge L_2 \wedge L_1 \wedge L_5)$
   $L_5 \longrightarrow (L_1 \wedge L_2 \wedge L_3 \wedge L_4)$

   It is obvious that neither of the two sets of precedence relationships above will exist. Therefore, the product can always be assembled successfully in this case.

Results

Once the collision information is derived by the collision detection function, it is used as the input to the precedence relationship extraction function. The result of the precedence relationship extraction function for the example product is shown as following:

1). $L_1 \longrightarrow L_5$

2). $L_2 \longrightarrow (L_1 \wedge L_5)$

3). $L_1 \longrightarrow (L_2 \wedge L_3 \wedge L_4)$

4). $L_2 \longrightarrow (L_3 \wedge L_4 \wedge L_5)$

5). $L_3 \longrightarrow (L_4 \wedge L_5 \wedge L_1)$

6). $L_5 \longrightarrow (L_4 \wedge L_3 \wedge L_2)$

Note that the precedence relationships determined conform exactly with the relationships generated by hand. This gives a validation of both the program and the methodology.

The set of sequences for the example product is shown in Figure 4.10. This graphical representation was introduced by De Fazio and Whitney (1987). Each block in the graph is a liaison, and the order from left to right corresponds with the liaison number in a ascending order. Blank blocks represent unestablished liaisons, the shaded blocks represent established liaisons. By traversing from one rank to the next through the solid lines, the final state can be reached and a valid sequence can be found. If a dashed line is followed, the sequence is eventually blocked and the final state cannot be reached. There are a total of 28 valid liaison sequences generated by the program:

1). $L_1 \longrightarrow L_2 \longrightarrow L_3 \longrightarrow L_4 \longrightarrow L_5$
2). $L_1 \longrightarrow L_2 \longrightarrow L_3 \longrightarrow L_5 \longrightarrow L_4$
3). $L_1 \longrightarrow L_2 \longrightarrow L_4 \longrightarrow L_3 \longrightarrow L_5$
4). $L_1 \longrightarrow L_2 \longrightarrow L_4 \longrightarrow L_5 \longrightarrow L_3$
5). $L_1 \longrightarrow L_2 \longrightarrow L_5 \longrightarrow L_3 \longrightarrow L_4$
6). $L_1 \longrightarrow L_2 \longrightarrow L_5 \longrightarrow L_4 \longrightarrow L_3$
7). $L_1 \longrightarrow L_3 \longrightarrow L_2 \longrightarrow L_4 \longrightarrow L_5$
8). $L_1 \longrightarrow L_3 \longrightarrow L_2 \longrightarrow L_5 \longrightarrow L_4$
9). $L_1 \longrightarrow L_4 \longrightarrow L_2 \longrightarrow L_3 \longrightarrow L_5$

10). $L_1 \longrightarrow L_4 \longrightarrow L_2 \longrightarrow L_5 \longrightarrow L_3$
11). $L_2 \longrightarrow L_1 \longrightarrow L_3 \longrightarrow L_4 \longrightarrow L_5$
12). $L_2 \longrightarrow L_1 \longrightarrow L_3 \longrightarrow L_5 \longrightarrow L_4$
13). $L_2 \longrightarrow L_1 \longrightarrow L_4 \longrightarrow L_3 \longrightarrow L_5$
14). $L_2 \longrightarrow L_1 \longrightarrow L_4 \longrightarrow L_5 \longrightarrow L_3$
15). $L_2 \longrightarrow L_1 \longrightarrow L_5 \longrightarrow L_3 \longrightarrow L_4$
16). $L_2 \longrightarrow L_1 \longrightarrow L_5 \longrightarrow L_4 \longrightarrow L_3$
17). $L_2 \longrightarrow L_3 \longrightarrow L_1 \longrightarrow L_4 \longrightarrow L_5$
18). $L_2 \longrightarrow L_3 \longrightarrow L_1 \longrightarrow L_5 \longrightarrow L_4$
19). $L_2 \longrightarrow L_4 \longrightarrow L_1 \longrightarrow L_3 \longrightarrow L_5$
20). $L_2 \longrightarrow L_4 \longrightarrow L_1 \longrightarrow L_5 \longrightarrow L_3$
21). $L_3 \longrightarrow L_1 \longrightarrow L_2 \longrightarrow L_4 \longrightarrow L_5$
22). $L_3 \longrightarrow L_1 \longrightarrow L_2 \longrightarrow L_5 \longrightarrow L_4$
23). $L_3 \longrightarrow L_2 \longrightarrow L_1 \longrightarrow L_4 \longrightarrow L_5$
24). $L_3 \longrightarrow L_2 \longrightarrow L_1 \longrightarrow L_5 \longrightarrow L_4$
25). $L_4 \longrightarrow L_1 \longrightarrow L_2 \longrightarrow L_3 \longrightarrow L_5$
26). $L_4 \longrightarrow L_1 \longrightarrow L_2 \longrightarrow L_5 \longrightarrow L_3$
27). $L_4 \longrightarrow L_2 \longrightarrow L_1 \longrightarrow L_3 \longrightarrow L_5$
28). $L_4 \longrightarrow L_2 \longrightarrow L_1 \longrightarrow L_5 \longrightarrow L_3$

Discussion

In Figure 4.10, if only the first (l - 2) liaisons are considered within each sequence, all of the sequences involving solid lines, dashed lines, or both are feasible up to the (l - 2^{nd}) liaison. This can be verified by examining Figure 3.4. Once state 5 with (l - 2) established liaisons is reached, the sequence cannot go beyond this state. In other words, the last two liaisons cannot be established.

In order to prevent traveling from state 1 to state 2, state 3 to state 5, and state 4 to state 5, there should be some precedence relationship that prevent both L_3 and L_4 from being established before other liaisons are established. Thus, the precedence relationship should be in the following format: $L_i \longrightarrow (L_3 \wedge L_4)$. The precedence

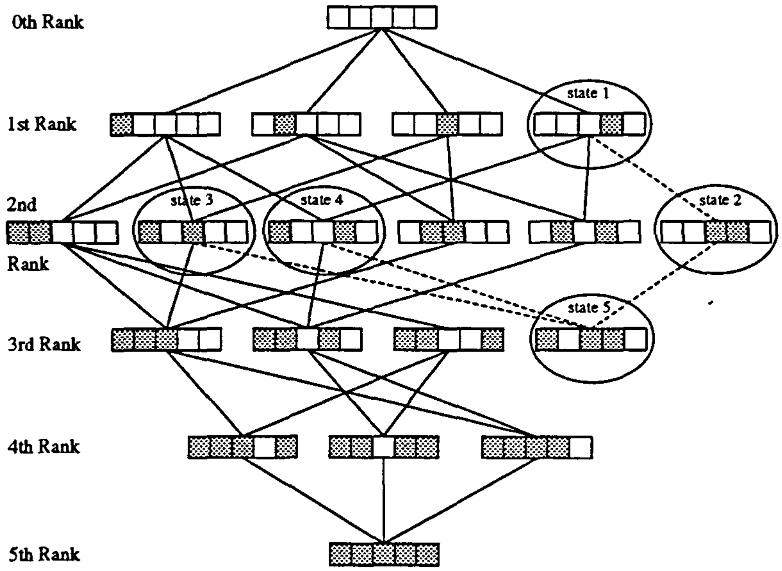


Figure 4.10: Graphical representation of all of valid liaison assembly sequences of the example product

relationships in this format mean both L_3 and L_4 cannot be established before some liaisons L_i is established. L_i can be found by examining Figure 4.10. If state 5 is reached, L_2 and L_5 cannot be established. Therefore, L_i can be either L_2 or L_5. In other words, the following two precedence relationships are needed to eliminate the dashed lines in Figure 4.10:

$$L_2 \longrightarrow (L_3 \wedge L_4) \qquad (1)$$

$$L_5 \longrightarrow (L_3 \wedge L_4) \qquad (2)$$

which are not extracted by the methodology. Precedences (1) and (2) can be combined into one precedence relationship:

$$(L_2 \wedge L_5) \longrightarrow (L_3 \wedge L_4) \qquad (3)$$

Precedence (3) means the Boolean value of $(L_3 \wedge L_4)$ cannot become one before the Boolean value of $(L_2 \wedge L_5)$ becomes one; otherwise, the product cannot be assembled successfully. Verification by hand shows that this is true. However, the proposed methodology does not identify the existence of precedences (1) and (2). For example, when $L_2 \longrightarrow (L_3 \wedge L_4)$ is checked, Figure 4.11 shows that L_2 can be disengaged by disassembling P_2, so this precedence relationship does not exist. This is because: A) the single liaison on the left hand side of the precedence sign is disengaged by disassembling one part only, and B) precedences (1) and (2) are checked separately. Thus, the forms for precedence relationships should not be restricted to the two basic forms only. Precedence (3) shows that a precedence relationship can be in the following general form:

$$(L_i \wedge L_j \wedge \ldots) \longrightarrow (Lx \wedge Ly \wedge \ldots)$$

There could be one or more liaisons on both sides of the precedence sign by using "$\wedge$" Boolean operator only. For example, the followings are different versions of precedence relationships:

$$L_i \longrightarrow L_j$$
$$L_i \longrightarrow (L_j \wedge \ldots)$$
$$(L_j \wedge \ldots) \longrightarrow L_i$$
$$(L_i \wedge \ldots) \longrightarrow (L_j \wedge \ldots)$$

In general, the disassembly approach should not assume removing only single part, or disengaging only single liaison. Subassemblies and multiple liaisons need to be considered as well.

In the case above for the simple-cycle liaison diagram, the final two liaisons are not adjacent to each other in the liaison diagram. As mentioned in the previous section, another case is that the final two liaisons are adjacent to each other in the liaison diagram. In this case, the last part inserted will establish the final two liaisons at the same time, and the following form is a possible precedence relationship for this case:

$$(L_{l-1} \vee (L_1 \wedge L_2 \wedge \ldots \wedge L_{l-3} \wedge L_{l-2})) \longrightarrow L_l \qquad (4)$$

which L_1, L_2, ..., and L_{l-2} are made of one subassembly. The last part inserted will establish L_{l-1} and L_l at the same time. First, $(L_1 \wedge L_2 \wedge \ldots \wedge L_{l-3} \wedge L_{l-2})) \longrightarrow L_l$ has to be checked. If this precedence relationship is true, all of L_1, L_2, ..., and L_{l-2} have to be established before L_l is established. Once L_1, L_2, ..., and L_{l-2} are established, there are only two liaisons (L_{l-1} and L_l) left. Then, it does not matter which one of L_{l-1} and L_l are established first because the Boolean value of the left hand side of the precedence sign is one already. An example shown in Figure 4.12 is used to illustrate this form. This is part of the example product used in the paper written by De Fazio and Whitney (1987). This example product is made of four parts which are all rotational parts, and its liaison diagram is in the simple cycle structure. In this example, the axis direction is the only disassembly (assembly) direction for these rotational parts. Following precedence relationship is true in this case:

$$(L_6 \vee (L_3 \wedge L_8)) \longrightarrow L_1$$

This precedence relationship means that either L_6 or $(L_3 \wedge L_8)$ has to be established before L_1 is established. If both L_3 and L_8 are established, the Boolean value of the left hand side of the precedence sign is one already. So, it does not matter L_1 or L_6 is established first because these two liaisons will be established at the same time. If L_6 is established first, then the sequence between L_3, L_8, and L_1 can be in any order. If L_1 is established first, the Boolean value of the left side of the precedence sign is still zero, but the Boolean value of the right side is one already. Thus, none of L_6, L_3, and L_8 can be established. The three cases stated above can be verified

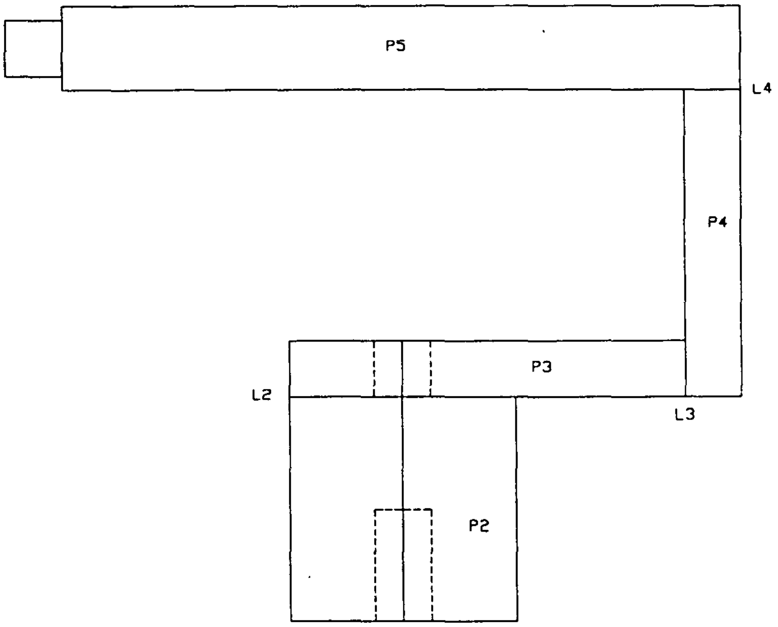


Figure 4.11: The subassembly consisting of L_2, L_3, and L_4

by examining Figure 4.12. Therefore, precedence (4) is another possible precedence which need to be considered.

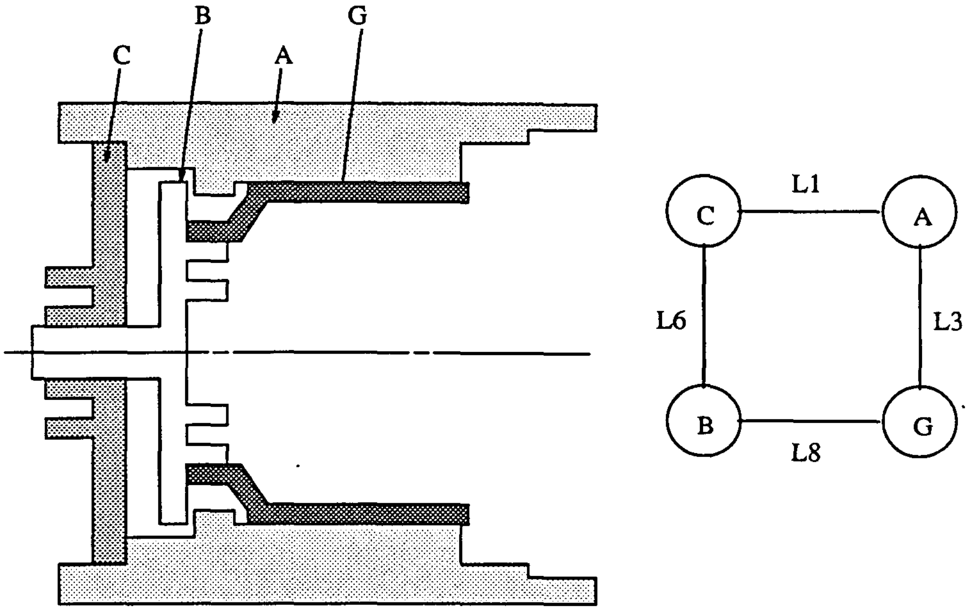


Figure 4.12: An example product with a simple-cycle liaison diagram

CHAPTER 5. CONCLUSION

A product design requires a great deal of analysis and trade-offs between cost and quality. The designer's initial ideas often does not work as intended. Therefore, the designer must make modifications to the original design. As the design progresses, these early modifications have a large impact on the cost and direction of the design. Analyses and experiments must be carried out to verify the design. As the design gradually works its way toward acceptability, the decisions on design changes become more interdependent. Often, later design decisions are affected by a decision made previously. Thus, earlier decisions have the most influence on the later course of the design.

The integrative nature of assembly seems to be a powerful force in raising the level of integration in all aspects of early product design (Nevins and Whitney, 1989). Generation of all the assembly sequences in the early design stage can help designers in two way: 1) to evaluate the assemblability of a product, and 2) to help the designer search for a better assembly sequence under a set of constraints. Different assembly sequences have different requirements for assembly fixturing, number of orientation changes, convenience of access, time of assembly, and possibilities of part damage during part mating. The difficulty of identifying the complete set of valid assembly sequences increases as the product complexity and part count increase. Thus, automatic precedence relationship extraction for assembly sequence generation becomes an important activity in the concurrent design environment and the implementation of an integrated manufacturing system.

We have developed a methodology that transforms a three-dimensional product design characterized by a simple-cycle liaison diagram under a solid modeling environment into a set of assembly precedence requirements. This methodology provides a tool which will assist both product designers and manufacturing engineers to identify valid assembly sequences. This research would also benefit engineers in the process and system aspects, so a product design can be evaluated during the early product design phase.

In this study, a methodology has been developed to perform geometric reasoning on the solid models of the assembly with a simple-cycle liaison diagram. Information concerning collision constraints is derived. This information is used as basic data for precedence relationship extraction.

So far, the proposed methodology can only ensure that the generated sequences are geometrically feasible up to the (l - 2^{nd}) liaison. In this research, only two basic forms for precedence relationships are extracted. For the two basic forms, there is only one liaison on the left hand side of the precedence sign, and one or more liaisons on the right hand side. The results show that considering only these precedences cannot get the complete set of precedence relationships. Multiple liaisons on both sides of the precedence sign should be considered.

Future Work

The activities for future research work are stated in the following:

1. Modification of proposed precedence extraction methodology by considering disassembling subassemblies and single part

2. Extend the methodology to include tree structure, and composite of simple cycles and trees together

   The methodology described in this thesis assumes that product design can be characterized by simple-cycle liaison diagram only. After the methodology is extended, product design of different liaison diagram structures can be evaluated.

3. Incorporate non-geometric constraints into the methodology

   The current methodology focuses on analyzing geometric constraints on the solid model of a product design. Some of the sequences generated by the methodology might not be feasible in reality because of some physical process constraints. To enhance the methodology, other factors need to be considered such as assembly stability, and fixturing requirements, etc.

BIBLIOGRAPHY

Bourjault, A., Contribution a une approche méthodologique de Fassemblage automatisé: Elaboration automatique des séquences opératiores. Thesis to obtain Grade de Docteur es Sciences Physiques at L'Université de Franche-Comté, Nov. 1984.

Boyse, John W., Interference Detection Among Solids and Surfaces. *Communication of the ACM*, 22(1), 1979, 3-9.

Chang,T.C., Wysk, R.A., and Wang, H.P., *Computer-Aided Manufacturing*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

Chen, L.L. and Woo, T.C., Computational Geometry on the Sphere with Applications to Automated Machining. *ASME Journal of Mechanical Design*, 114(2), 1992, 288-295

De Fazio, Thomas L. and Whitney, D.E., Simplified Generation of All Mechanical Assembly Sequences. *IEEE Journal of Robotics and Automation*, RA-3(6), 1987, 640-658.

Hoffman, Richard, Automated Assembly in a CSG Domain. *IEEE International Conference on Robotics and Automation.*, 1, 1989, 210-215.

Homem de Mello, L.S. and Sanderson, A.C., AND/OR Graph Representation of Assembly Plans. *AAAS-86 Proc. of the Fifth Nat'l Conf. on Artificial Intelligence*, August 11-15, 1986, Philadelphia, PA.

Homem de Mello, L.S. and Sanderson, A.C., AND/OR Graph Representation of Assembly Plans. *IEEE Transactions on Robotics and Automation*, April, 1990, 6(2), 188-199.

Homem de Mello, L.S. and Sanderson, A.C., Assembly Sequence Planning. *Artificial Intelligence Magazine*, Spring, 1990, 62-81.

Huang, Y.F. and Lee, C.S.G., Precedence Knowledge in Feature Mating Operation Assembly Planning. *1989 Proc. of IEEE International Conf. on Robotics & Automation*, 1, 1989, 216-221.

I-DEAS : Version 5, User's Guide. Structural Dynamics Research Corporation, Milford, Ohio, 1990.

Khosla, Pradeep k. and Mattikali, Raju, Determining the Assembly Sequence from a 3-D Model. *Journal of Mechanical Working Technology*, 20, 1989, 153-162.

Ko, Heedong and Lee, Kunwoo, Automatic Assembling Procedure Generation from Mating Conditions. *Computer-Aided Design*, 19, 1987, 3-10.

Lin, Alan C., Automatic Assembly Planning for Three-Dimensional Mechanical Products. PhD dissertation, Purdue University, West lafayette, Indiana, 1990.

Lin, Alan C. and Chang, T.C., Automated Assembly Planning for 3-Dimensional Mechanical Products. *Proceedings of NSF Design and manufacturing Systems Conf.*, Austin TX, Jan. 1990.

Lozano-Perez, T. and Wesley, M.A., An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles. *Communications of the ACM*, 22(10), 1979, 560-570.

Lozano-Perez, T., Spatial Planning : A Configuration Space Approach. *IEEE Transactions on Computers*, C-32(1), 1983, 108-120.

Lui, M.M., Generation and Evaluation of Mechanical Assembly Sequences Using the Liaison-Sequence Method. Master's thesis, Dept. of Mech. Eng., MIT, May, 1988. Also Report CSDL-T-990, The Charles Stark Draper Lab.

Miller, Joseph M. and Hoffman, Richard L., Automatic Assembly Planning with Fasteners. *1989 Proc. of IEEE International Conf. on Robotics & Automation*, 1, 1989, 69-74.

Mortenson, Michael E., *Geometric Modeling*, John Wiley & Sons, Inc., New York, NY, 1985.

Nevins, J.L. and Whitney, D.E., *Concurrent Design of Product and Processes*, McGraw-Hill, Inc., New York, NY, 1989.

Roth, Scott D., Ray Casting for Modeling Solids. *Computer Graphics and Image Processing*, 18, 1982, 109-144.

Wang, W.P. and Wang, K.K., Geometric Modeling for Swept Volume of Moving Solids. *IEEE Comput. Graph. & Appl.*, 6(12), 1986, 8-17.

Wang, W.P., Three-Dimensional Collision Avoidance in Production Automation. *Computer in Industry*, 1990, 169-174.

Whitney, D.E., De Fazio, T.L., Gustavson, R.E., Graves, S.C., Baldwin, D.F., Jastrzebski, M.J., Milner, J.M., Tung, K.K., and Whipple, R.W., Tools for Strategic Product Design. *Proc. of the 1991 NSF Design and Manufacturing Systems Conference*, Jan. 9-11, 1991, 1179-1186.

Woo, T.C. and Dutta, D., Automatic Disassembly and Total Ordering in Three Dimensions. *Trans. of the ASME, Journal of Engineering for Industry*, 113, 1991, 207-213.

APPENDIX A. USER MANUAL

Introduction

This program extracts liaison precedence relationships by reasoning the solid model of a product design in I-DEAS, and then generates all of the valid liaison assembly sequences. The product design must be developed in I-DEAS, and should be characterized by a simple-cycle liaison diagram only. The extracted precedence relationships and all of the valid assembly sequences can be output either on the screen or to the file called **pre_seq.out**. This manual describes the inputs to the program, the program structure, and the operations of the program.

Inputs

The inputs to the program include: 1) the universal files containing the geometric information describing the product design, 2) the liaison diagram for the product design, and 3) the disassembly directions for each part.

1. Users have to create the solid model for the product design by using I-DEAS, and generate a universal file for each part. I-DEAS provides the command to generate the universal files. There are five parts for the example product, and the universal file names with respect to each part are list below:

   (a) P_1: part1.unv

(b) P_2: part2.unv

(c) P_3: part3.unv

(d) P_4: part4.unv

(e) P_5: part5.unv

2. The liaison diagram for the product design that contains the relationships between parts and liaisons must be provided in a file called **cycle.dat**. The liaison data of the example product stored in the file **cycle.dat** are input as:

```
1 1 2
2 2 3
3 3 4
4 4 5
5 5 1
```

The first row will now be used to illustrate the format of **cycle.dat**. The middle digit, 1, is the liaison number. The right digit, 2, is the right part number with respect to L_1 in the liaison diagram. The left digit, 1, is the left part number with respect to L_1. The way to determine the left part and the right part with respect to a liaison is to imagine standing inside the liaison diagram (cycle) (see Figure 3.5). Look at one of the liaisons, say L_i, the part number appearing on the right of L_i will be the right digit of i, and the part number appearing on the left of L_i will be the left digit of i. Any two liaisons adjacent to each other in the liaison diagram must have their respective data row adjacent as well.

3. The disassembly directions for each part are stored in the file **parts.dat**. Disassembly directions tell the program what the valid disassembly directions for each part. The **parts.dat** of the example product is given in the following:

```
1 0 0 -1 0   5 0 -1 0 0   0 0 0 0 0
```

```
1 0 0 1 0   2 0 0 -1 0   0 0 0 0 0
2 3 0 1 0   3 0 -1 0 0   3 0 0 -1 0   3 0 0 0 1   3 0 0 0 -1   0 0 0 0 0
3 4 1 0 0   3 4 0 -1 0   3 4 0 0 1   3 4 0 0 -1   3 0 0 1 0   4 0 -1 0 0   0 0 0 0 0
4 5 1 0 0   4 0 -1 0 0   4 0 0 1 0   4 0 0 0 1   4 0 0 0 -1   0 0 0 0 0          .
```

The selection of these disassembly directions follows the proposed methodology described in Chapter 3. The first row is the disassembly directions of P_1, the second row is the disassembly directions for P_2, and so forth. When the program reads these data, each group of five digits is read as one unit. Each unit contains the liaison numbers of the two liaisons associated with a disassembly direction, along with the disassembly direction. The first two values in a unit are the liaison numbers involved in the disassembly direction. The third, fourth, and fifth values are the x, y, and z components of the vector of the disassembly direction. For example, the first unit (2, 3, 0, 1, 0) of row 3 shows that disassembly direction (0, 1, 0) is associated with both liaison 2 and 3. The second unit in row 3 shows that direction (-1, 0, 0) for part 3 is associated with liaison 3 only, and so forth. For a simple-cycle liaison diagram, one disassembly direction can be associated with at most two liaisons. Thus, when a disassembly direction is associated with only one liaison, one of the first two values in a data set is set to zero. The last unit in a row consists of five zero digits. When the program reaches this unit, it goes to the next row.

Program Structure

There are six major functions in this program:

1. liaison_diagram(): read liaison-diagram information.

2. disassembly_direction(): read the disassembly directions for each part and the liaison that each disassembly direction is associated with.

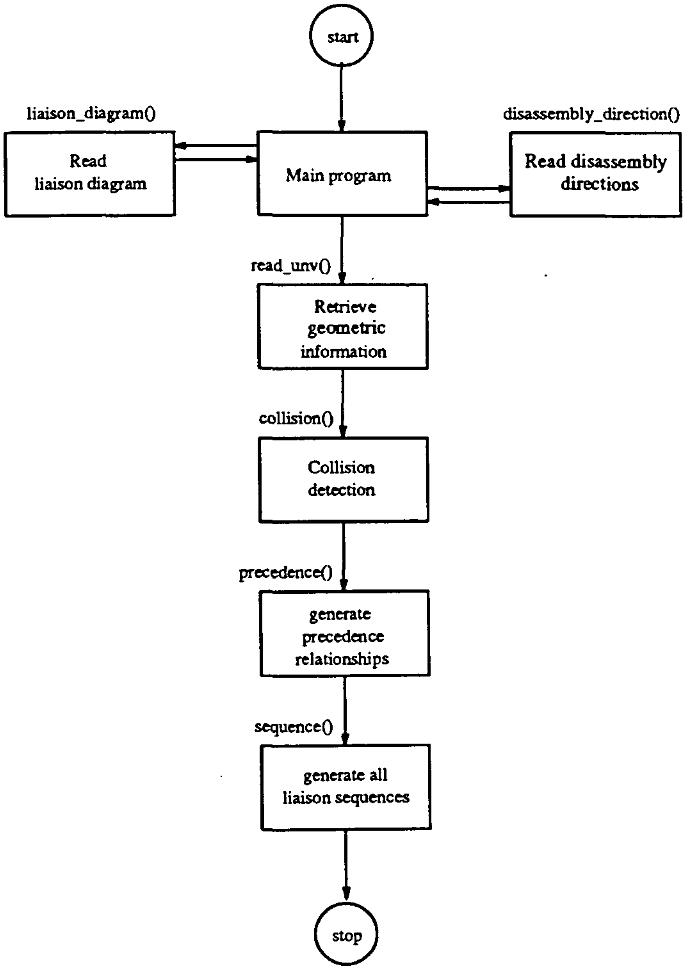
3. read_unv(): retrieve the geometric information of the product design from I-DEAS.

4. collision(): detect the potential collisions.

5. precedence(): derive precedence relationships.

6. sequence(): generate all of the valid assembly sequences based on the precedence relationships derived.

The functional relationships of the six major functions are shown in the following flow chart:



First, the main program retrieves all the input data it needs including: liaison diagram, disassembly directions, and geometric information of the product design. Secondly, those inputs retrieved are imported into the collision detection function. Once the potential collision information is generated, it is input to the precedence relationship extraction function. Based on the extracted precedence relationships, the 'sequence' function then generates all the valid assembly sequences.

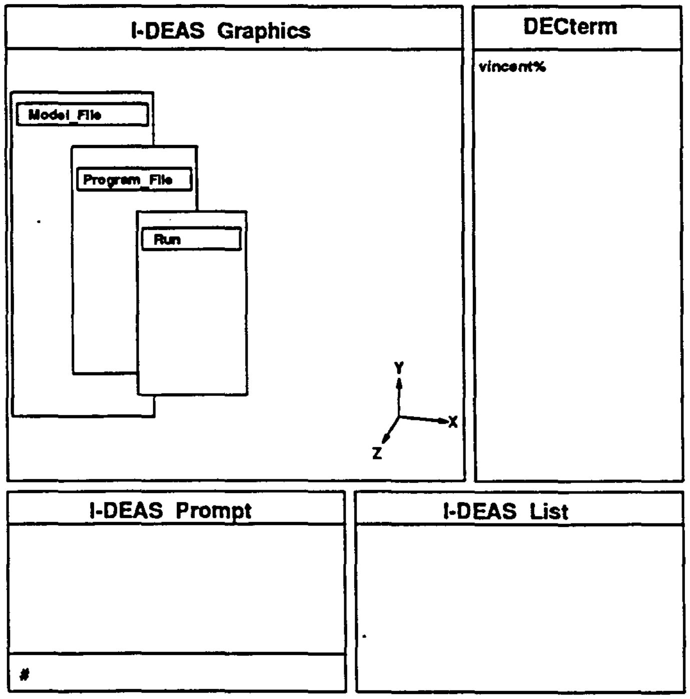
Operations

The program is coded in the C programming language, and runs in the UNIX environment. The source code is stored under the file name **pre_seq.c**. The **pre_seq.c** needs to use mathematical functions defined in math.h, so the following command must be used to compile the program.

```
vincent% cc pre_seq.c -lm
```

Where vincent% is the prompt of the window. If the executable file name is not specified, **a.out** will be the default executable file name.

Once the proper input files have been created, the user must log into I-DEAS. I-DEAS interfaces with the C program, and thus I-DEAS must be running in order that the C program runs. Once we get into I-DEAS, I-DEAS will automatically create three windows: I-DEAS Graphics, I-DEAS Prompt, and I-DEAS List. I-DEAS Graphics is used for graphical display purpose, I-DEAS Prompt is where users input commands, and I-DEAS List is used to show users operational information. In order to run the C program, a DECterm window has to be created, and these four windows can be arranged in the positions as shown in the following figure.



The procedure for running this program is described in following steps:

1. Input **a.out** in the DECterm window, and hit the 'RETURN' key.

   vincent% **a.out**

   Then the program will show following information on the screen:

```
*****************************************************************
*                                                               *
*       AUTOMATIC PRECEDENCE RELATIONSHIP EXTRACTION FOR        *
*                ASSEMBLY SEQUENCE GENERATION                   *
*                                                               *
*****************************************************************

Output the result to a file(pre_seq.out) or sreen ?
(1). file.
(2). screen.
(3). both(file and screen).
--> (1/2/3) 3

=================================================================
*      Please start to run the program_file ==> cycle.prg.      *
* ------------------------------------------------------------- *
*      Once the program file stop running, hit 'RETURN' in      *
*      DECterm window to resume the C program ....              *
=================================================================
```

2. The next step is to invoke the Object Modeling task under the Solid Modeling family in I-DEAS. Next, go to the main menu in I-DEAS Graphics window and select 'Model File', 'Program File', and 'Run' (see the figure in the previous page). Now input the program file name by typing **cycle** in I-DEAS Prompt window and hit the 'RETURN' key.

3. After the I-DEAS program file stops running, the interference checking result has been output to the file called **IN_CH.dat**. Activate the DECterm window by hitting the 'RETURN' key. This activates the C program, which then processes the interference checking result written to the **IN_CH.dat** file.

4. After the C program stops running, the interference checking result in **IN_CH.dat** has been resolved, and the I-DEAS commands for the next step have been put into **cycle.prg**. Go to I-DEAS Prompt window and hit the 'RETURN' key, reactivating the I-DEAS program file. Then go to step 3, and continue this procedure until the C program terminates.

The purpose of using I-DEAS is to do the interference checking, so step 3 and 4 are repeated until all the potential collisions are examined. Once the collision detections are done, the C program will generate the precedence relationships and all of the valid liaison assembly sequences.

APPENDIX B. PROGRAM LIST

```
/* ****************************************************************** */
/*                                                                    */
/*   AUTOMATIC PRECEDENCE RELATIONSHIP EXTRACTION FOR ASSEMBLY       */
/*                    SEQUENCE GENERATION                             */
/*                                                                    */
/*                         Hung-Yi Tu                                 */
/*                                                                    */
/*       Department of Industrial and Manufacturing Systems Engineering */
/*             Iowa State University of Science and Technology        */
/*                      Ames, Iowa 50011 USA                          */
/*                                                                    */
/* ------------------------------------------------------------------ */
/*                                                                    */
/*    This program can derive the precedence relationships from a    */
/*   solid model in a CAD system called I-DEAS, and generate all the */
/*   valid assembly sequences based on these precedence relationships. */
/*   Following information is the inputs of this program :           */
/*     (1). Liaison diagram.                                          */
/*          This information is stored in the "cycle.dat" file.       */
/*     (2). Disassembly directions for each part.                     */
/*          This information is stored in the "parts.dat" file.       */
/*                                                                    */
/*--------------------------------------------------------------------*/
/*                                                                    */
/*   Following variables are defined before the main program :       */
/*     (1). N  : Number of liaisons.                                  */
/*     (2). lm : The length of the square work plane defined in I-DEAS. */
/*     (3). EL : The length of the swept volume.                      */
/*                                                                    */
/*   In collision detection function, the projection of each facet of */
/*   every part is found, so lm is just used to define the size of pro- */
/*   jection plane in I-DEAS. Once a projection is found, it has to be */
/*   extruded to create a swept volume. So we need to determine EL to */
/*   be the length of the swept volume. When a different product design */
/*   is tested, N, lm, and EL might be changed. We can simply change  */
/*   the defined values of these variables at the beginning of the   */
/*   program and recompile the program.                               */
/*                                                                    */
/* ------------------------------------------------------------------ */
/*                                                                    */
/*   There are nine functions in this program :                       */
/*     (1). liaison_diagram() : read liaison-diagram information.     */
/*     (2). disassembly_direction() : read the disassembly directions for */
/*          each part and the liaisons that each diassembly           */
/*          direction is associated with.                             */
/*      (3). read_unv() : retrieve the geometrical information of the          */
/*           product design from I-DEAS.                                       */
/*      (4). collision() : detect the potential collisions.                    */
/*       (5). collision_store() : store the potential collision information.   */
/*      (6). int_check_res() : read the interference checking result           */
/*           generated from I-DEAS.                                            */
/*      (7). N_intersection_check() : to determine whether a moving part       */
/*            will collide with a static part or not when their                */
/*             rectangular boxes do not have intersection.                     */
/*      (8). precedence() : derive precedence relationships.                   */
/*      (9). sequence() : generate all the sequences based on precedence       */
/*            relationships.                                                   */
/*                                                                             */
/* ******************************************************************        */

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#define N 5
#define lm 20
#define EL 25.0

/* ========================================== */
/*   This structure is used to store the liaison diagram     */
/*   information described as following :                   */
/*    (1) All the contact parts for each part.               */
/*     (2) The liaisons that each part is associated with.   */
/* ========================================== */
typedef struct listnode1
     {
          int lia;
          int lpart;
          int rpart;
           struct listnode1 *right, *left;
          } NODE1; .

/* ============================================ */
/*   This structure is used to store                          */
/*     (1) All the disassembly directions for each part.      */
/*     (2) The liaison that each disassembly direction is     */
/*          assocaited with.                                  */
/* ============================================ */
typedef struct listnode2
     {
          int   asso[2];
          float dir[3];
           struct listnode2 *next;
          } NODE2;

/* ============================================ */
/*    This structure is used to store the collision           */
/*    information.                                            */
/* ============================================ */
typedef struct listnode3
     {
          int part;
          float dd[3];
```

```
        struct listnode3 *next;
       } NODE3;

/* ============================================= */
/*   This structure is used to store precedence   */
/*   relationship information.                    */
/* ============================================= */
typedef struct plistnode
     {
       int   liaison;
       struct plistnode *next;
     } NODE4;

NODE1 *liaison_diagram();
disassembly_direction();
read_unv();
collision();
collision_store();
int_check_res();
N_intersection_check();
precedence();
sequence();

int   option;
FILE *output;
NODE4 *pr[N+1][N-1][3],*prtail[N+1][N-1][3];

/* ============================================= */
/*            ****  MAIN PROGRAM  ****            */
/* ============================================= */
main()
{
 char c;
  int i,j,k,rank,seq[N+1];

 NODE1 *clist;
 NODE2 *part[N];
  NODE3 *colli[N],*cltail[N];

   printf("****************************************************************\n");
 printf("*                                                              *\n");
 printf("*        AUTOMATIC PRECEDENCE RELATIONSHIP EXTRACTION FOR      *\n");
 printf("*                 ASSEMBLY SEQUENCE GENERATION                 *\n");
 printf("*                                                              *\n");
   printf("****************************************************************\n\n");

 do
  {
    printf("Output the result to a file(pre_seq.out) or sreen ?\n");
     printf("(1). file.\n(2). screen.\n(3). both(file and screen).\n");
    printf("--> (1/2/3) ");
    scanf("%d",&option);
   scanf("%c",&c);

    if (option != 1 && option != 2 && option != 3)
          printf("\nPlease select 1 or 2 or 3 !!!\n\n");
    else printf("\n");
  } while (option != 1 && option != 2 && option != 3);
  if (option == 1 || option == 3)
    output = fopen("pre_seq.out","w");

  clist = liaison_diagram();
   disassembly_direction(part);

   collision(part,colli,cltail);

  for (i = 1; i <= N; i++)
   for (j = 1; j <= (N-2); j++)
     for (k = 1; k <= 2; k++)
      {
         pr[i][j][k] = prtail[i][j][k] = malloc(sizeof(NODE4));
         pr[i][j][k]->liaison = i+1;
        pr[i][j][k]->next = NULL;
      }

   precedence(clist,part,colli);

  if (option == 1 || option == 3)
    fprintf(output,"\n\nAll the possible sequences :\n\n");
  if (option == 2 || option == 3)
    printf("\n\nAll the possible sequences :\n\n");

  rank = 1;
  sequence(rank,seq);

  if (option == 1 || option == 3)
    fclose(output);
}

/* ------------------------------------------------------------ */
/*                  Input the liaision diagram                  */
/* ------------------------------------------------------------ */
NODE1 *liaison_diagram()
{
 FILE *stream;
  NODE1 *list,*tail,*p;

  stream = fopen("cycle.dat","r");
  list = tail = malloc(sizeof(NODE1));

  while (! feof(stream))
   {
     p = malloc(sizeof(NODE1));
      fscanf(stream,"%d %d %d",&p->lia,&p->lpart,&p->rpart);
      tail->right = p;
     p->left = tail;
     tail = p;
   }

  fclose(stream);
  p = list;
  list = list->right;
  tail->right = list;
  list->left = tail;
  free(p);
  return(list);
}
```

```
/* ------------------------------------------------------------ */
/*      Input the disassembly direction information      */
/* ------------------------------------------------------------ */
disassembly_direction(node)
NODE2 *node[N];
{
 FILE *stream;
 char c;
 int x, y, i, j, k;
 NODE2 *p,*tail;

  stream = fopen("parts.dat","r");

 for (i = 0; i < N; i++)
  {
    node[i] = tail = malloc(sizeof(NODE2));

    p = malloc(sizeof(NODE2));
    for (j = 0; j < 2; j++)
          fscanf(stream,"%d",&p->asso[j]);
    for (j = 0; j < 3; j++)
          fscanf(stream,"%f",&p->dir[j]);

     while (p->asso[0] != 0 || p->asso[1] != 0)
     {
            p->next = NULL;
       tail->next = p;
       tail = p;

       p = malloc(sizeof(NODE2));

            for (j = 0; j < 2; j++)
              fscanf(stream,"%d",&p->asso[j]);
       for (j = 0; j < 3; j++)
              fscanf(stream,"%f",&p->dir[j]);
     }
  }
  fclose(stream);
}

/* ------------------------------------------------------------------ */
/*      Check the potential collisions for each part in its       */
/*     disassembly directions.                                      */
/* ------------------------------------------------------------------ */
collision(part,colli,cltail)
NODE2 *part[N];
NODE3 *colli[N],*cltail[N];
{
 FILE *prg;
  NODE2 *p,*r,*pp,*qq,*head,*head_pp;
 NODE3 *q,*rr;
  char   TR,con,con1,line,jump,jump1,skip,stack,delete,result,ret;
  int     a,b,i,j,s,t,u,k1,k2,ss,add,flag,key1,key2,seq,code,mark,
        index,c_code2,point_no,g[3],sub[3],count[N],face_no[N],
        f_p_no[N][50],p_label[N][50][100],temp;
  float e,step,s1,s2,as1,as2,dx1,dy1,p1[40],p2[40],D1,D2,AD1,AD2,
         diff,front,back,inner_p,length_1,length_2,cos_theta,theta,
        vector_1[3],vector_2[3],normal[3],md[3],trans[3],Mt[N][4][4],
     box[N][2][4],nbox[N][2][4],box_d[2][3],point[N][100][4],
    npoint[N][100][4];

static char code1[] = "   534";
static char code2[] = "   537";
static char code3[] = "   544";

for (i = 0; i < N; i++)
  count[i] = 0;

for (i = 0; i < N; i++)
  colli[i] =  cltail[i] = malloc(sizeof(NODE3));

index = 0;
e = 1.0E-5;
for (i = 0; i < 5; i++)
 {
  if (count[i] == 0)
         read_unv(i,count,face_no,f_p_no,p_label,nbox,npoint);

  for (j = 0; j < 5; j++)
         if (i != j)
           {
            if (count[j] == 0)
                   read_unv(j,count,face_no,f_p_no,p_label,nbox,npoint);

            if (index == 0)
                   {
                    index = 1;
                    prg = fopen("cycle.prg","w");
                   }
            else
                   {
                    prg = fopen("cycle.prg","a");
                    if (delete == 'Y')
                      fprintf(prg,"K :/MA\nK :DE\nK :E1\nK :\n");
                   }

            if (i != 0)
                   fprintf(prg,"K :/DE\nK :\n");
            else if (i == 0 && j != 1)
                   fprintf(prg,"K :/DE\nK :\nK :/TA\nK :O\n");

            if (i == 0)
                   {
                    for (s = 0; s < 3; s++)
                     {
                      if (j == 1)
                             box_d[0][s] = fabs(nbox[i][0][s] - nbox[i][1][s]);
                        box_d[1][s] = fabs(nbox[j][0][s] - nbox[j][1][s]);
                     }

                    if (j == 1)
                     {
                       fprintf(prg,"K :/CR\nK :B\nK :");
                      for (t = 0; t <= 2; t++)
                             fprintf(prg,"%7.3f  ",box_d[0][t]);
                       fprintf(prg,"\n");
```

```
        TR = 'N';
        for (t = 0; t < 3; t++)
            {
             trans[t] = (nbox[i][0][t] + nbox[i][1][t])/2;
             if (fabs(trans[t]) > e)
              TR = 'Y';
            }

        if (TR == 'Y')
            {
             fprintf(prg,"K :/OR\nK :TR\nK :");
             for (t = 0; t < 3; t++)
               fprintf(prg,"%7.3f ",trans[t]);
              fprintf(prg,"\n");
            }

         fprintf(prg,"K :/MA\nK :STO\nK :");
          fprintf(prg,"B1\nK :\n");
       }

      fprintf(prg,"K :/CR\nK :B\nK :");
     for (t = 0; t <= 2; t++)
        fprintf(prg,"%7.3f ",box_d[1][t]);
      fprintf(prg,"\n");

     TR = 'N';
     for (t = 0; t < 3; t++)
      {
        trans[t] = (nbox[j][0][t] + nbox[j][1][t])/2;
       if (fabs(trans[t]) > e)
            TR = 'Y';
      }

     if (TR == 'Y')
      {
        fprintf(prg,"K :/OR\nK :TR\nK :");
       for (t = 0; t < 3; t++)
            fprintf(prg,"%7.3f ",trans[t]);
        fprintf(prg,"\n");
      }

      fprintf(prg,"K :/MA\nK :STO\nK :");
     switch (j)
      {
       case 1 : fprintf(prg,"B2\nK :\n");
       break;
       case 2 : fprintf(prg,"B3\nK :\n");
       break;
       case 3 : fprintf(prg,"B4\nK :\n");
       break;
       case 4 : fprintf(prg,"B5\nK :\n");
       break;
      }
    }

if (i == 0)
     fprintf(prg,"K :/TA\nK :AS\n");

if (i == 0)
```

```
        {
         if (j == 1)
           fprintf(prg,"K :/CR\nK :C\nK :\nK :B1\nK :\nK :B1\nK :\nK :\nK :D\n");
          else fprintf(prg,"K :/I\nK :A\nK :\nK :B1\nK :\nK :\nK :D\n");

          fprintf(prg,"K :/CR\nK :C\nK :\nK :");
         switch (j)
          {
           case 0 : fprintf(prg,"B1\nK :\nK :B1\nK :\nK :D\n");
            break;
           case 1 : fprintf(prg,"B2\nK :\nK :B2\nK :\nK :D\n");
            break;
           case 2 : fprintf(prg,"B3\nK :\nK :B3\nK :\nK :D\n");
            break;
           case 3 : fprintf(prg,"B4\nK :\nK :B4\nK :\nK :D\n");
            break;
           case 4 : fprintf(prg,"B5\nK :\nK :B5\nK :\nK :D\n");
            break;
          }
        }
else
        {
         fprintf(prg,"K :/I\nK :A\n");
         for (s = 0; s < 2; s++)
          {
           if (s == 0)
                  u = i;
           else u = j;

           switch (u)
                  {
                  case 0 : fprintf(prg,"K :\nK :B1\nK :\nK :\n");
                    break;
                  case 1 : fprintf(prg,"K :\nK :B2\nK :\nK :\n");
                    break;
                  case 2 : fprintf(prg,"K :\nK :B3\nK :\nK :\n");
                    break;
                  case 3 : fprintf(prg,"K :\nK :B4\nK :\nK :\n");
                    break;
                  case 4 : fprintf(prg,"K :\nK :B5\nK :\nK :\n");
                    break;
                  }
          }
          fprintf(prg,"K :D\n");
        }

if (i == 0)
        fprintf(prg,"K :DR\n");
 fprintf(prg,"K :/L\nK :CH\n");
if (i == 0 && j == 1)
        fprintf(prg,"K :V\nK :On\n");
 fprintf(prg,"K :C\nK :L\nK :");
switch (i)
        {
        case 0 : fprintf(prg,"B1\n");
          break;
        case 1 : fprintf(prg,"B2\n");
          break;
        case 2 : fprintf(prg,"B3\n");
```

```
          break;
        case 3 : fprintf(prg,"B4\n");
          break;
        case 4 : fprintf(prg,"B5\n");
          break;
        }
 fprintf(prg,"K :\nK :L\nK :Y\nK :IN_CH\nK :Y\n? :\n");
 fclose(prg);

if (i == 0 && j == 1)
        {
          printf("=============================================================");
          printf("\n*     Please start to run the program_file ==> cycle.prg.     *\n");
          printf("* ------------------------------------------------------- *");
          printf("\n*     Once the program file stop running, hit 'RETURN' in     *");
          printf("\n*     DECterm window to resume the C program ....            *\n");

printf("=============================================================\n");
          scanf("%c",&ret);
        }
else
        {
          printf("=============================================================");
          printf("\n*     Check potential collisions between P%d and P%d.          *",i+1,j+1);
          printf("\n* ------------------------------------------------------- *");
          printf("\n*     Please CONTINUE the 'I-DEAS program file'.             *\n");
          printf("* ------------------------------------------------------- *");
          printf("\n*     Once the program file stop running, hit 'RETURN' in     *");
          printf("\n*     DECterm window to resume the C program ....            *\n");

printf("=============================================================\n");
          scanf("%c",&ret);
        }

  int_check_res(i,j,&result);

if (result == 'N')
        {
          head = p = part[i]->next;
          do
           {
             for (s = 0; s < 3; s++)
                  md[s] = p->dir[s];

               N_intersection_check(i,j,md,&result,nbox);

             if (result == 'Y')
                  collision_store(i,j,cltail,md);

             p = p->next;
           } while (p != NULL);

          printf("\n");
          for (u = 0; u < 5; u++)
           {
             printf("P%d : ",u+1);
             q = colli[u]->next;
             temp = 0;
             while (q != NULL)
```

```
                              {
                               if (temp != 0)
                                printf("     ");
                               printf("P%d --> (%5.2f,%5.2f,%5.2f)\n",q->part,q->dd[0],q->dd[1],q->dd[2]);
                               q = q->next;
                               temp++;
                              }
                      }
                      printf("\n");
                     delete = 'N';
                    }
             else
                    {
                     key1 = 0;
                     flag = 0;
                     head = p = part[i]->next;
                     do
                      {
                        skip = 'N';
                        for (s = 0; s < 3; s++)
                              md[s] = p->dir[s];

                        if (i > j)
                              {
                               r = part[j]->next;
                               while (r != NULL && skip == 'N')
                                {
                                 add = 0;
                                 for (t = 0; t < 3; t++)
                                        {
                                         if (md[t] == -r->dir[t])
                                          add++;
                                        }

                                 if (add == 3)
                                        {
                                         if ((p->asso[0] != 0) && (r->asso[0] == p->asso[0] || r->asso[1] == p-
>asso[0]))

                                           skip = 'Y';
                                         else if ((p->asso[1] != 0) && (r->asso[0] == p->asso[1] || r->asso[1] == p-
>asso[1]))

                                           skip = 'Y';
                                         else
                                           {
                                            rr = colli[j]->next;
                                            while (rr != NULL)
                                                  {
                                                   add = 0;
                                                   for (t = 0; t < 3; t++)
                                                    {
                                                     if (md[t] == -rr->dd[t])
                                                            add++;
                                                    }

                                                   if (add == 3 && rr->part == i+1)
                                                    {
                                                      skip = 'Y';
                                                      q = malloc(sizeof(NODE3));
                                                      q->part = j + 1;
                                         for (t = 0; t < 3; t++)
                                              q->dd[t] = -rr->dd[t];
                                         q->next = NULL;
                                         cltail[i]->next = q;
                                         cltail[i] = q;

                                          printf("\n");
                                         for (u = 0; u < 5; u++)
                                              {
                                               printf("P%d : ",u+1);
                                               q = colli[u]->next;
                                               temp = 0;
                                               while (q != NULL)
                                                {
                                                 if (temp != 0)
                                                      printf("      ");
                                                  printf("  P%d --> (%5.2f,%5.2f,%5.2f)\n",q->part,q-
>dd[0],q->dd[1],q->dd[2]);

                                                 q = q->next;
                                                 temp++;
                                                }
                                              }
                                          printf("\n");
                                         delete = 'N';
                                        }
                                       rr = rr->next;
                                      }
                                 }
                               }
                          r = r->next;
                         }
                       }

                  if (skip == 'N')
                       {
                        for (s = 0; s < face_no[i]; s++)
                         {
                          if (md[0] != 0)
                               {
                                seq = 0;
                                sub[0] = 2;
                                sub[1] = 1;
                                sub[2] = 0;
                               }
                          else if (md[1] != 0)
                               {
                                seq = 1;
                                sub[0] = 0;
                                sub[1] = 2;
                                sub[2] = 1;
                               }
                          else
                               {
                                seq = 2;
                                sub[0] = 0;
                                sub[1] = 1;
                                sub[2] = 2;
                               }
```

```
u = 0;
con = 'Y';
do
     {
       k1 = p_label[i][s][u];
       k2 = p_label[i][s][u+1];
       D1 = npoint[i][k2][sub[0]] - npoint[i][k1][sub[0]];
       D2 = npoint[i][k2][sub[1]] - npoint[i][k1][sub[1]];
       AD1 = fabs(D1);
       AD2 = fabs(D2);

       if (AD1 > e || AD2 > e)
        con = 'N';
       u++;
     } while (con == 'Y');

line = 'Y';
add = 0;
for (t = 0; t < f_p_no[i][s]; t++)
     {
       a = p_label[i][s][t];

       stack = 'Y';
       if (t > 0)
        {
         switch (seq)
              {
              case 0 :
               if (fabs(p1[mark] - -npoint[i][a][sub[0]]) > e || fabs(p2[mark] -
npoint[i][a][sub[1]]) > e)
                stack = 'N';
               break;
              case 1 :
               if (fabs(p1[mark] - npoint[i][a][sub[0]]) > e || fabs(p2[mark] -
npoint[i][a][sub[1]]) > e)
                stack = 'N';
               break;
              case 2 :
               if (fabs(p1[mark] - npoint[i][a][sub[0]]) > e || fabs(p2[mark] -
npoint[i][a][sub[1]]) > e)
                stack = 'N';
               break;
              }
        }

       if (t == 0 || stack == 'N')
        {
         switch (seq)
              {
              case 0 :
               {
                p1[add] = -npoint[i][a][sub[0]];
                p2[add] = npoint[i][a][sub[1]];
               }
               break;
              case 1 :
               {
                p1[add] = npoint[i][a][sub[0]];
```

```
                          p2[add] = npoint[i][a][sub[1]];
                        }
                        break;
                      case 2 :
                        {
                          p1[add] = npoint[i][a][sub[0]];
                          p2[add] = npoint[i][a][sub[1]];
                        }
                        break;
                      }
             mark = add;
             add++;
           }

          if (t >= (u + 1) && line == 'Y')
           {
              b = p_label[i][s][t-1];
               s1 = npoint[i][a][sub[0]] - npoint[i][b][sub[0]];
               s2 = npoint[i][a][sub[1]] - npoint[i][b][sub[1]];
              as1 = fabs(s1);
              as2 = fabs(s2);

             if (as1 < e && as2 < e)
                      line = 'Y';
             else
                      {
                        if (AD1 > e && AD2 > e)
                          {
                            if (as1 > e && as2 > e && ((s1/s2 - D1/D2) < e))
                                     line = 'Y';
                            else
                                     line = 'N';
                          }
                        else if (AD1 > e && AD2 < e)
                          {
                            if (as1 > e && as2 < e)
                                     line = 'Y';
                            else
                                     line = 'N';
                          }
                        else
                          {
                            if (as1 < e && as2 > e)
                                     line = 'Y';
                            else
                                     line = 'N';
                          }
                       }
            }
         }

if (line == 'N')
         {
           flag++;
           if (flag == 1)
             {
                prg = fopen("cycle.prg","a");
                fprintf(prg,"K :/DE\nK :\nK :/TA\nK :CG\n");
             }
```

```
else
 {
   prg = fopen("cycle.prg","a");
   fprintf(prg,"K :/DE\nK :\nK :/MA\nK :DE\nK :E1\nK :\nK :/TA\nK
:CG\n");
 }

if (i == 0 && s == 0 && key1 == 0 && key2 == 0)
 fprintf(prg,"K :/W\nK :K :AT\nK :SE\nK :K\nK :%d %d\nK :%d
%d\n",lm,lm,-lm,-lm);

switch (seq)
 {
  case 0 : fprintf(prg,"K :/W\nK :RE\nK :YZ\nK :K\nK :\n");
   break;
  case 1 : fprintf(prg,"K :/W\nK :RE\nK :XZ\nK :K\nK :\n");
   break;
  case 2 : fprintf(prg,"K :/W\nK :RE\nK :XY\nK :K\nK :\n");
   break;
 }

 fprintf(prg,"K :/CR\nK :PR\nK :K\n");

for (t = 0; t < add; t++)
 {
    fprintf(prg,"K :%10.6f %10.6f\n",p1[t],p2[t]);
  if (t != 0)
         fprintf(prg,"K  :F\n");
 }

 fprintf(prg,"K :C\nK :\n");

 a = p_label[i][s][0];
 front = npoint[i][a][sub[2]];
 back = npoint[i][a][sub[2]];

 switch (seq)
  {
  case 0 :
   if (md[0] > 0)
          for (t = 1; t < f_p_no[i][s]; t++)
           {
             a = p_label[i][s][t];
             if (npoint[i][a][sub[2]] > front)
               front = npoint[i][a][sub[2]];
             else if (npoint[i][a][sub[2]] < back)
               back = npoint[i][a][sub[2]];
           }
   else
          for (t = 1; t < f_p_no[i][s]; t++)
           {
             a = p_label[i][s][t];
             if (npoint[i][a][sub[2]] < front)
               front = npoint[i][a][sub[2]];
             else if (npoint[i][a][sub[2]] > back)
               back = npoint[i][a][sub[2]];
           }
   break;
  case 1 :
```

```
  if (md[1] > 0)
        for (t = 1; t < f_p_no[i][s]; t++)
         {
           a = p_label[i][s][t];
           if (npoint[i][a][sub[2]] > front)
             front = npoint[i][a][sub[2]];
           else if (npoint[i][a][sub[2]] < back)
             back = npoint[i][a][sub[2]];
         }
  else
        for (t = 1; t < f_p_no[i][s]; t++)
         {
           a = p_label[i][s][t];
           if (npoint[i][a][sub[2]] < front)
             front = npoint[i][a][sub[2]];
           else if (npoint[i][a][sub[2]] > back)
             back = npoint[i][a][sub[2]];
         }
  break;
 case 2 :
  if (md[2] > 0)
        for (t = 1; t < f_p_no[i][s]; t++)
         {
           a = p_label[i][s][t];
           if (npoint[i][a][sub[2]] > front)
             front = npoint[i][a][sub[2]];
           else if (npoint[i][a][sub[2]] < back)
             back = npoint[i][a][sub[2]];
         }
  else
        for (t = 1; t < f_p_no[i][s]; t++)
         {
           a = p_label[i][s][t];
           if (npoint[i][a][sub[2]] < front)
             front = npoint[i][a][sub[2]];
           else if (npoint[i][a][sub[2]] > back)
             back = npoint[i][a][sub[2]];
         }
  break;
 }

switch (seq)
 {
 case 0 :
   fprintf(prg,"K :/W\nK :TR\nK :0 0 %6.2f\nK :/MA\nK :PR\nK :STO\nK
:PF1\nK :\n",back);
  break;
 case 1 :
   fprintf(prg,"K :/W\nK :TR\nK :0 0 %6.2f\nK :/MA\nK :PR\nK :STO\nK
:PF1\nK :\n",-back);
  break;
 case 2 :
   fprintf(prg,"K :/W\nK :TR\nK :0 0 %6.2f\nK :/MA\nK :PR\nK :STO\nK
:PF1\nK :\n",back);
  break;
 }

fprintf(prg,"K :/DE\nK :P\nK :\nK :/TA\nK :O\n");
```

```
switch (seq)
 {
 case 0 :
  {
        fprintf(prg,"K :/CR\nK :E\n");
        fprintf(prg,"K :PF1\n");

        if (md[0] > 0)
          fprintf(prg,"K :%6.2f\n",EL);
        else fprintf(prg,"K :%6.2f\n",-EL);

        if (fabs(front - back) > e)
         {
          for (t = 0; t < 3; t++)
            g[t] = p_label[i][s][t];

          for (t = 0; t < 3; t++)
           {
                vector_1[t] = npoint[i][g[0]][t] - npoint[i][g[1]][t];
                vector_2[t] = npoint[i][g[1]][t] - npoint[i][g[2]][t];
           }

           normal[0] = vector_1[1] * vector_2[2] - vector_1[2] *
vector_2[1];
           normal[1] = vector_1[2] * vector_2[0] - vector_1[0] *
vector_2[2];
           normal[2] = vector_1[0] * vector_2[1] - vector_1[1] *
vector_2[0];

           inner_p = length_1 = length_2 = 0;

          for (t = 0; t < 3; t++)
           {
                inner_p += md[t] * normal[t];
                length_1 += pow(md[t],2.);
                length_2 += pow(normal[t],2.);
           }

           length_1 = sqrt(length_1);
           length_2 = sqrt(length_2);

           cos_theta = inner_p/(length_1 * length_2);

          fprintf(prg,"K :/CO\nK :P\nK :T\nK :K\n");
          if (cos_theta > 0)
           for (t = 0; t < 3; t++)
                {
                  fprintf(prg,"K :");
                 for (u = 0; u < 3; u++)
                    fprintf(prg,"%10.6f  ",npoint[i][g[t]][u]);
                  fprintf(prg,"\n");
                }
          else
           for (t = 2; t >= 0; t--)
                {
                  fprintf(prg,"K :");
                 for (u = 0; u < 3; u++)
                    fprintf(prg,"%10.6f  ",npoint[i][g[t]][u]);
                  fprintf(prg,"\n");
```

```
                                              }
                                          fprintf(prg,"K :P\n");
                                         }

                                        fprintf(prg,"K :/MA\nK :STO\nK :E1\nK :\nK :\n");

                                        fprintf(prg,"K :/TA\nK :AS\nK :/CR\nK :C\nK :\nK :E1\nK :\nK
:E1\nK :\nK :D\n");
                                        fprintf(prg,"K :DR\nK :/I\nK :A\n");

                                        switch (j)
                                         {
                                          case 0 : fprintf(prg,"K :\nK :P1\nK :\nK :\n");
                                           break;
                                          case 1 : fprintf(prg,"K :\nK :P2\nK :\nK :\n");
                                           break;
                                          case 2 : fprintf(prg,"K :\nK :P3\nK :\nK :\n");
                                           break;
                                          case 3 : fprintf(prg,"K :\nK :P4\nK :\nK :\n");
                                           break;
                                          case 4 : fprintf(prg,"K :\nK :P5\nK :\nK :\n");
                                           break;
                                         }
                                        fprintf(prg,"K :D\n");
                                        fprintf(prg,"K :/L\nK :CH\nK :C\nK :L\nK :E1\nK :\nK :L\nK
:Y\nK :IN_CH\nK :Y\n? :\n");

                                        fclose(prg);

        printf("==============================================================================");
                                        printf("\n*     Please CONTINUE the 'I-DEAS program file'.
*\n");
                                        printf("* -------------------------------------------------------- *");
                                        printf("\n*     Once the program file stop running, hit 'RETURN' in
*");
                                        printf("\n*     DECterm window to resume the C program ....
*\n");

        printf("==============================================================================\n");
                                        scanf("%c",&ret);

                                        int_check_res(i,j,&result);

                                        if (result == 'Y')
                                         {
                                          s = 999;
                                            collision_store(i,j,cltail,md);
                                         }
                                    }
                                    break;
                                   case 1 :
                                    {
                                        fprintf(prg,"K :/CR\nK :E\n");
                                        fprintf(prg,"K :PF1\n");

                                        if (md[1] > 0)
                                          fprintf(prg,"K :%6.2f\n",-EL);
                                        else fprintf(prg,"K :%6.2f\n",EL);

                                        if (fabs(front - back) > e)
```

```
{
 for (t = 0; t < 3; t++)
   g[t] = p_label[i][s][t];

 for (t = 0; t < 3; t++)
  {
        vector_1[t] = npoint[i][g[0]][t] - npoint[i][g[1]][t];
        vector_2[t] = npoint[i][g[1]][t] - npoint[i][g[2]][t];
  }

  normal[0] = vector_1[1] * vector_2[2] - vector_1[2] *
vector_2[1];
  normal[1] = vector_1[2] * vector_2[0] - vector_1[0] *
vector_2[2];
  normal[2] = vector_1[0] * vector_2[1] - vector_1[1] *
vector_2[0];

  inner_p = length_1 = length_2 = 0;

 for (t = 0; t < 3; t++)
  {
        inner_p += md[t] * normal[t];
        length_1 += pow(md[t],2.);
        length_2 += pow(normal[t],2.);
  }

  length_1 = sqrt(length_1);
  length_2 = sqrt(length_2);

  cos_theta = inner_p/(length_1 * length_2);

  fprintf(prg,"K :/CO\nK :P\nK :T\nK :K\nK :");
 if (cos_theta > 0)
  for (t = 0; t < 3; t++)
        {
         fprintf(prg,"K :");
        for (u = 0; u < 3; u++)
           fprintf(prg,"%10.6f  ",npoint[i][g[t]][u]);
         fprintf(prg,"\n");
        }
 else
  for (t = 2; t >= 0; t--)
        {
         fprintf(prg,"K :");
        for (u = 0; u < 3; u++)
           fprintf(prg,"%10.6f  ",npoint[i][g[t]][u]);
         fprintf(prg,"\n");
        }
  fprintf(prg,"K :P\n");
}

fprintf(prg,"K :/MA\nK :STO\nK :E1\nK :\nK :\n");

fprintf(prg,"K :/TA\nK :AS\nK :/CR\nK :C\nK :\nK :E1\nK :\nK
:E1\nK :\nK :D\n");

fprintf(prg,"K :DR\nK :/I\nK :A\n");

switch (j)
 {
```

```
                                  case 0 : fprintf(prg,"K :\nK :P1\nK :\nK :\n");
                                   break;
                                  case 1 : fprintf(prg,"K :\nK :P2\nK :\nK :\n");
                                   break;
                                  case 2 : fprintf(prg,"K :\nK :P3\nK :\nK :\n");
                                   break;
                                  case 3 : fprintf(prg,"K :\nK :P4\nK :\nK :\n");
                                   break;
                                  case 4 : fprintf(prg,"K :\nK :P5\nK :\nK :\n");
                                   break;
                                 }

                                fprintf(prg,"K :D\n");
                                fprintf(prg,"K :/L\nK :CH\nK :C\nK :L\nK :E1\nK :\nK :L\nK
:Y\nK :IN_CH\nK :Y\n? :\n");
                                fclose(prg);

     printf("=========================================================");
                                printf("\n*    Please CONTINUE the 'I-DEAS program file'.
*\n");
                                printf("* ---------------------------------------------------- *");
                                printf("\n*    Once the program file stop running, hit 'RETURN' in
*");
                                printf("\n*    DECterm window to resume the C program ....
*\n");

     printf("=========================================================\n");
                                scanf("%c",&ret);

                                int_check_res(i,j,&result);

                                if (result == 'Y')
                                 {
                                  s = 999;
                                    collision_store(i,j,cltail,md);
                                 }
                            }
                            break;
                           case 2 :
                            {
                                fprintf(prg,"K :/CR\nK :E\n");
                                fprintf(prg,"K :PF1\n");

                                if (md[2] > 0)
                                  fprintf(prg,"K :%6.2f\n",EL);
                                else fprintf(prg,"K :%6.2f\n",-EL);

                                if (fabs(front - back) > e)
                                 {
                                  for (t = 0; t < 3; t++)
                                    g[t] = p_label[i][s][t];

                                  for (t = 0; t < 3; t++)
                                   {
                                        vector_1[t] = npoint[i][g[0]][t] - npoint[i][g[1]][t];
                                        vector_2[t] = npoint[i][g[1]][t] - npoint[i][g[2]][t];
                                   }
```

```
   normal[0] = vector_1[1] * vector_2[2] - vector_1[2] *
vector_2[1];
   normal[1] = vector_1[2] * vector_2[0] - vector_1[0] *
vector_2[2];
   normal[2] = vector_1[0] * vector_2[1] - vector_1[1] *
vector_2[0];

   inner_p = length_1 = length_2 = 0;

  for (t = 0; t < 3; t++)
   {
         inner_p += md[t] * normal[t];
         length_1 += pow(md[t],2.);
         length_2 += pow(normal[t],2.);
   }

   length_1 = sqrt(length_1);
   length_2 = sqrt(length_2);

    cos_theta = inner_p/(length_1 * length_2);

   fprintf(prg,"K :/CO\nK :P\nK :T\nK :K\nK :");
   if (cos_theta > 0)
    for (t = 0; t < 3; t++)
         {
           fprintf(prg,"K :");
          for (u = 0; u < 3; u++)
             fprintf(prg,"%10.6f  ",npoint[i][g[t]][u]);
            fprintf(prg,"\n");
         }
  else
    for (t = 2; t >= 0; t--)
         {
           fprintf(prg,"K :");
          for (u = 0; u < 3; u++)
             fprintf(prg,"%10.6f  ",npoint[i][g[t]][u]);
            fprintf(prg,"\n");
         }
   fprintf(prg,"K :P\n");
 }

fprintf(prg,"K :/MA\nK :STO\nK :E1\nK :\nK :\n");

fprintf(prg,"K :/TA\nK :AS\nK :/CR\nK :C\nK :\nK :E1\nK :\nK
:E1\nK :\nK :D\n");
fprintf(prg,"K :DR\nK :/I\nK :A\n");

switch (j)
 {
  case 0 : fprintf(prg,"K :\nK :P1\nK :\nK :\n");
   break;
  case 1 : fprintf(prg,"K :\nK :P2\nK :\nK :\n");
   break;
  case 2 : fprintf(prg,"K :\nK :P3\nK :\nK :\n");
   break;
  case 3 : fprintf(prg,"K :\nK :P4\nK :\nK :\n");
   break;
  case 4 : fprintf(prg,"K :\nK :P5\nK :\nK :\n");
   break;
```

```
                                         }

                                        fprintf(prg,"K :D\n");
                                        fprintf(prg,"K :/L\nK :CH\nK :C\nK :L\nK :E1\nK :\nK :L\nK
:Y\nK :IN_CH\nK :Y\n? :\n");
                                        fclose(prg);

        printf("===========================================================================");
                                        printf("\n*     Please CONTINUE the 'I-DEAS program file'.
*\n");
                                        printf("* ------------------------------------------------------- *");
                                        printf("\n*     Once the program file stop running, hit 'RETURN' in
*");
                                        printf("\n*     DECterm window to resume the C program ....
*\n");

        printf("===========================================================================\n");
                                        scanf("%c",&ret);

                                        int_check_res(i,j,&result);

                                        if (result == 'Y')
                                         {
                                          s = 999;
                                           collision_store(i,j,cltail,md);
                                         }
                                   }
                                   break;
                                  }
                                }
                          }

                          printf("\n");
                         for (u = 0; u <= i; u++)
                          {
                           printf("P%d : ",u+1);
                           q = colli[u]->next;
                           temp = 0;
                           while (q != NULL)
                                {
                                 if (temp != 0)
                                  printf("    ");
                                 printf("P%d --> (%5.2f,%5.2f,%5.2f)\n",q->part,q->dd[0],q->dd[1],q-
>dd[2]);
                                 q = q->next;
                                 temp++;
                                }
                          }
                          printf("\n");
                         delete = 'Y';
                        }
                    p = p->next;
                   } while (p != NULL);
                 }
          }
  }
}
/* ---------------------------------------------------------------- */
/*    Read the output file for interference checking result      */
/*    from I-DEAS.                                                */
/* ------------------------------------------------------------- */
int_check_res(i,j,result)
int  i,j;
char *result;
{
 FILE *stream;
 char c;
 int s;
 static char str1[] = "Interference detected for";
 static char str2[] = "NONE";

 stream = fopen("IN_CH.dat","r");

 c = fgetc(stream);
 s = 999;
 do
  {
   if (c == 'I')
          for (s = 1; s <= 24; s++)
            {
              c = fgetc(stream);
              if (c != str1[s])
                s = 999;
            }
   c = fgetc(stream);
  } while(!(s == 25));

 while (c != '-')
   c = fgetc(stream);
 while (c == '-')
   c = fgetc(stream);

 for (s = 0; s < 2; s++)
   c = fgetc(stream);

 for (s = 0; s < 4; s++)
  {
   c = fgetc(stream);
   if (c != str2[s])
          s = 999;
  }

 if (s != 1000)
  {
   *result = 'N';
   printf("\n            ***** NO interference *****\n");
  }
 else
  {
   *result = 'Y';
   printf("            ***** Interference exists *****\n");
  }
 fclose(stream);
}

/* ---------------------------------------------------------------- */
/*     Check which of the following conditions that the              */
/*   . rectangular box of the moving part is in :                    */
/*      (1). Completely above or below the rectangular box of        */
/*           the static part.                                        */
/*      (2). Completely on the right side or left side of the        */
/*           rectangular box of the static part.                     */
/*      (3). Completely behind the rectangular box of the            */
/*           static part.                                            */
/*      (4). None of the above three conditions.                     */
/* ----------------------------------------------------------------- */
N_intersection_check(i,j,md,inter_f,nbox)
int i,j;
char *inter_f;
float md[3],nbox[N][2][4];
{
  int s,seq;
   float  e,back1,front2,up1,up2,down1,down2,right1,right2,left1,left2;

  e = 1.0E-5;

  if (md[0] != 0)
    seq = 0;
  else if (md[1] != 0)
    seq = 1;
  else
    seq = 2;

  switch (seq)
   {
   case 0 :
    if (md[0] > 0)
          {
           if (nbox[i][0][0] > nbox[i][1][0])
             back1  = nbox[i][1][0];
           else back1  = nbox[i][0][0];

           if (nbox[j][0][0] > nbox[j][1][0])
             front2 = nbox[j][0][0];
           else front2  = nbox[j][1][0];

           if (back1 >= front2 || fabs(back1 - front2) < e)
             *inter_f = 'N';
           else
            {
              if (nbox[i][0][1] > nbox[i][1][1])
                    {
                     up1 = nbox[i][0][1];
                     down1 = nbox[i][1][1];
                    }
             else
                    {
                     up1 = nbox[i][1][1];
                     down1 = nbox[i][0][1];
                    }

              if (nbox[j][0][1] > nbox[j][1][1])
                    {
                     up2 = nbox[j][0][1];
                     down2 = nbox[j][1][1];
                              }
            else
                    {
                      up2 = nbox[j][1][1];
                      down2 = nbox[j][0][1];
                    }

            if (up1 <= down2 || fabs(up1 - down2) < e || down1 >= up2 || fabs(down1 - up2) < e)
                    *inter_f = 'N';
            else
                    {
                      if (nbox[i][0][2] > nbox[i][1][2])
                       {
                          right1 = nbox[i][1][2];
                          left1 = nbox[i][0][2];
                       }
                      else
                       {
                          right1 = nbox[i][0][2];
                          left1 = nbox[i][1][2];
                       }

                      if (nbox[j][0][2] > nbox[j][1][2])
                       {
                          right2 = nbox[j][1][2];
                          left2 = nbox[j][0][2];
                       }
                      else
                       {
                          right2 = nbox[j][0][2];
                          left2 = nbox[j][1][2];
                       }

                      if (right1 >= left2 || fabs(right1 - left2) < e || left1 <= right2 || fabs(left1 - right2) < e)
                        *inter_f = 'N';
                      else *inter_f = 'Y';
                    }
           }
        }
else
        {
          if (nbox[i][0][0] > nbox[i][1][0])
           back1  = nbox[i][0][0];
          else back1 = nbox[i][1][0];

          if (nbox[j][0][0] > nbox[j][1][0])
           front2 = nbox[j][1][0];
          else front2 = nbox[j][0][0];

          if (back1 <= front2 || fabs(back1 - front2) < e)
           *inter_f = 'N';
          else
           {
             if (nbox[i][0][1] > nbox[i][1][1])
                    {
                      up1 = nbox[i][0][1];
                      down1 = nbox[i][1][1];
                    }
            else
                    {
```

```
                   up1 = nbox[i][1][1];
                   down1 = nbox[i][0][1];
                  }

            if (nbox[j][0][1] > nbox[j][1][1])
                  {
                   up2 = nbox[j][0][1];
                   down2 = nbox[j][1][1];
                  }
           else
                  {
                   up2 = nbox[j][1][1];
                   down2 = nbox[j][0][1];
                  }

           if (up1 <= down2 || fabs(up1 - down2) < e || down1 >= up2 || fabs(down1 - up2) < e)
                  *inter_f = 'N';
           else
                  {
                   if (nbox[i][0][2] > nbox[i][1][2])
                    {
                      right1 = nbox[i][0][2];
                      left1 = nbox[i][1][2];
                    }
                   else
                    {
                      right1 = nbox[i][1][2];
                      left1 = nbox[i][0][2];
                    }

                   if (nbox[j][0][2] > nbox[j][1][2])
                    {
                      right2 = nbox[j][0][2];
                      left2 = nbox[j][1][2];
                    }
                   else
                    {
                      right2 = nbox[j][1][2];
                      left2 = nbox[j][0][2];
                    }

                   if (right1 <= left2 || fabs(right1 - left2) < e || left1 >= right2 || fabs(left1 - right2) < e)
                     *inter_f = 'N';
                   else *inter_f = 'Y';
                  }
          }
        }
 break;
case 1 :
 if (md[1] > 0)
        {
         if (nbox[i][0][1] > nbox[i][1][1])
           back1  = nbox[i][1][1];
         else back1  = nbox[i][0][1];

         if (nbox[j][0][1] > nbox[j][1][1])
           front2 = nbox[j][0][1];
         else front2  = nbox[j][1][1];
```

```
          if (back1 >= front2 || fabs(back1 - front2) < e)
            *inter_f = 'N';
          else
           {
              if (nbox[i][0][2] > nbox[i][1][2])
                     {
                       up1 = nbox[i][1][2];
                       down1 = nbox[i][0][2];
                     }
             else
                     {
                       up1 = nbox[i][0][2];
                       down1 = nbox[i][1][2];
                     }

              if (nbox[j][0][2] > nbox[j][1][2])
                     {
                       up2 = nbox[j][1][2];
                       down2 = nbox[j][0][2];
                     }
             else
                     {
                       up2 = nbox[j][0][2];
                       down2 = nbox[j][1][2];
                     }

             if (up1 >= down2 || fabs(up1 - down2) < e || down1 <= up2 || fabs(down1 - up2) < e)
                     *inter_f = 'N';
             else
                     {
                       if (nbox[i][0][0] > nbox[i][1][0])
                        {
                           right1 = nbox[i][0][0];
                           left1 = nbox[i][1][0];
                        }
                      else
                        {
                           right1 = nbox[i][1][0];
                           left1 = nbox[i][0][0];
                        }

                       if (nbox[j][0][0] > nbox[j][1][0])
                        {
                           right2 = nbox[j][0][0];
                           left2 = nbox[j][1][0];
                        }
                      else
                        {
                           right2 = nbox[j][1][0];
                           left2 = nbox[j][0][0];
                        }

                      if (right1 <= left2 || fabs(right1 - left2) < e || left1 >= right2 || fabs(left1 - right2) < e)
                       *inter_f = 'N';
                      else *inter_f = 'Y';
                     }
           }
        }
else
```

```
{
 if (nbox[i][0][1] > nbox[i][1][1])
   back1  = nbox[i][0][1];
 else back1 = nbox[i][1][1];

 if (nbox[j][0][1] > nbox[j][1][1])
   front2 = nbox[j][1][1];
 else front2 = nbox[j][0][1];

 if (back1 <= front2 || fabs(back1 - front2) < e)
   *inter_f = 'N';
 else
  {
    if (nbox[i][0][2] > nbox[i][1][2])
          {
           up1 = nbox[i][1][2];
           down1 = nbox[i][0][2];
          }
   else
          {
           up1 = nbox[i][0][2];
           down1 = nbox[i][1][2];
          }

    if (nbox[j][0][2] > nbox[j][1][2])
          {
           up2 = nbox[j][1][2];
           down2 = nbox[j][0][2];
          }
   else
          {
           up2 = nbox[j][0][2];
           down2 = nbox[j][1][2];
          }

   if (up1 >= down2 || fabs(up1 - down2) < e || down1 <= up2 || fabs(down1 - up2) < e)
          *inter_f = 'N';
   else
          {
           if (nbox[i][0][0] > nbox[i][1][0])
            {
              right1 = nbox[i][1][0];
              left1 = nbox[i][0][0];
            }
          else
            {
              right1 = nbox[i][0][0];
              left1 = nbox[i][1][0];
            }

           if (nbox[j][0][0] > nbox[j][1][0])
            {
              right2 = nbox[j][1][0];
              left2 = nbox[j][0][0];
            }
          else
            {
              right2 = nbox[j][0][0];
              left2 = nbox[j][1][0];
```

```
                }

                if (right1 >= left2 || fabs(right1 - left2) < e || left1 <= right2 || fabs(left1 - right2) < e)
                 *inter_f = 'N';
                else *inter_f = 'Y';
               }
        }
      }
 break;
case 2 :
 if (md[2] > 0)
      {
       if (nbox[i][0][2] > nbox[i][1][2])
        back1  = nbox[i][1][2];
       else back1  = nbox[i][0][2];

       if (nbox[j][0][2] > nbox[j][1][2])
        front2 = nbox[j][0][2];
       else front2  = nbox[j][1][2];

       if (back1 >= front2 || fabs(back1 - front2) < e)
        *inter_f = 'N';
       else
        {
         if (nbox[i][0][1] > nbox[i][1][1])
              {
               up1 = nbox[i][0][1];
               down1 = nbox[i][1][1];
              }
         else
              {
               up1 = nbox[i][1][1];
               down1 = nbox[i][0][1];
              }

         if (nbox[j][0][1] > nbox[j][1][1])
              {
               up2 = nbox[j][0][1];
               down2 = nbox[j][1][1];
              }
         else
              {
               up2 = nbox[j][1][1];
               down2 = nbox[j][0][1];
              }

         if (up1 <= down2 || fabs(up1 - down2) < e || down1 >= up2 || fabs(down1 - up2) < e)
              *inter_f = 'N';
         else
              {
               if (nbox[i][0][0] > nbox[i][1][0])
                {
                 right1 = nbox[i][0][0];
                 left1 = nbox[i][1][0];
                }
               else
                {
                 right1 = nbox[i][1][0];
                 left1 = nbox[i][0][0];
```

```
            }

            if (nbox[j][0][0] > nbox[j][1][0])
            {
              right2 = nbox[j][0][0];
              left2 = nbox[j][1][0];
            }
            else
            {
              right2 = nbox[j][1][0];
              left2 = nbox[j][0][0];
            }

            if (right1 <= left2 || fabs(right1 - left2) < e || left1 >= right2 || fabs(left1 - right2) < e)
             *inter_f = 'N';
            else *inter_f = 'Y';
           }
      }
    }
else
    {
     if (nbox[i][0][2] > nbox[i][1][2])
      back1  = nbox[i][0][2];
     else back1 = nbox[i][1][2];

     if (nbox[j][0][2] > nbox[j][1][2])
      front2 = nbox[j][1][2];
     else front2 = nbox[j][0][2];

     if (back1 <= front2 || fabs(back1 - front2) < e)
      *inter_f = 'N';
     else
      {
       if (nbox[i][0][1] > nbox[i][1][1])
          {
           up1 = nbox[i][0][1];
           down1 = nbox[i][1][1];
          }
      else
          {
           up1 = nbox[i][1][1];
           down1 = nbox[i][0][1];
          }

       if (nbox[j][0][1] > nbox[j][1][1])
          {
           up2 = nbox[j][0][1];
           down2 = nbox[j][1][1];
          }
      else
          {
           up2 = nbox[j][1][1];
           down2 = nbox[j][0][1];
          }

      if (up1 <= down2 || fabs(up1 - down2) < e || down1 >= up2 || fabs(down1 - up2) < e)
          *inter_f = 'N';
      else
          {
                 if (nbox[i][0][0] > nbox[i][1][0])
                  {
                    right1 = nbox[i][1][0];
                    left1 = nbox[i][0][0];
                  }
                 else
                  {
                    right1 = nbox[i][0][0];
                    left1 = nbox[i][1][0];
                  }

                 if (nbox[j][0][0] > nbox[j][1][0])
                  {
                    right2 = nbox[j][1][0];
                    left2 = nbox[j][0][0];
                  }
                 else
                  {
                    right2 = nbox[j][0][0];
                    left2 = nbox[j][1][0];
                  }

                 if (right1 >= left2 || fabs(right1 - left2) < e || left1 <= right2 || fabs(left1 - right2) < e)
                  *inter_f = 'N';
                 else *inter_f = 'Y';
                }
          }
        }
    break;
  }
}

/* ---------------------------------------------------------------- */
/*     Retrieve the geometrical information of the product         */
/*     design from the universal files in I-DEAS.                  */
/* ---------------------------------------------------------------- */
read_unv(i,count,face_no,f_p_no,p_label,nbox,npoint)
int i,count[N],face_no[N],f_p_no[N][50],p_label[N][50][100];
float nbox[N][2][4],npoint[N][100][4];
{
 FILE *inf;
 char c,c2,multi[2];
  int s,t,u,code,c_code2,point_no;
  float e,step,box[N][2][4],Mt[N][4][4],point[N][100][4];

 static char code1[] = "   534";
 static char code2[] = "   537";
 static char code3[] = "   544";

 count[i]++;

 switch (i)
  {
   case 0 : inf = fopen("part1.unv","r");
    break;
   case 1 : inf = fopen("part2.unv","r");
    break;
   case 2 : inf = fopen("part3.unv","r");
    break;
  case 3 : inf = fopen("part4.unv","r");
   break;
  case 4 : inf = fopen("part5.unv","r");
   break;
 }

c_code2 = 0;
do
 {
  t = 0;
  c2 = 'N';
  code = 999;
  c = fgetc(inf);
  do
          {
           if (c == ' ')
            for (t = 1; t <= 5; t++)
             {
                     c = fgetc(inf);
                     if (c == code1[t] && c2 == 'N')
                      {
                       if (t == 5 )
                          code = 534;
                      }
                     else if (c == code2[t] && c_code2 == 0)
                      {
                       c2 = 'Y';
                       c_code2++;
                       if (t = 5)
                         code = 537;
                      }
                     else if (c == code3[t])
                      {
                       c2 = 'Y';
                       if (t == 5)
                         code = 544;
                      }
                     else
                      {
                       c2 = 'N';
                       t = 999;
                      }
             }
           c = fgetc(inf);
          } while (!(t == 6));

   if (code == 534 || code == 537)
          {
           for (t = 1; t <= 30; t++)
              fscanf(inf,"%f",&step);

           for (s = 0; s < 2; s++)
            for (t = 0; t < 3; t++)
               fscanf(inf,"%f",&box[i][s][t]);
           box[i][0][3] = box[i][1][3] = 1;

           for (t = 1; t <= 6; t++)
              fscanf(inf,"%f",&step);
```

```
        for (s = 0; s <= 3; s++)
         for (t = 0; t <= 2; t++)
           fscanf(inf,"%f",&Mt[i][t][s]);

        Mt[i][3][3] = 1;
        Mt[i][3][0] = Mt[i][3][1] = Mt[i][3][2] = 0;

        multi[0] = 'N';
        for (s = 0; s < 4; s++)
         for (t = 0; t < 4; t++)
          {
                if (s == t)
                 {
                  if (Mt[i][s][t] != 1)
                   {
                        s = t = 4;
                        multi[0] = 'Y';
                   }
                 }
                else if (Mt[i][s][t] != 0)
                 {
                  s = t =4;
                  multi[0] = 'Y';
                 }
          }

        if (multi[0] == 'Y')
         {
          for (s = 0; s < 2; s++)
                for (t = 0; t < 4; t++)
                  nbox[i][s][t] = 0;

          for (s = 0; s < 2; s++)
                for (t = 0; t <= 3; t++)
                 for (u = 0; u <= 3; u++)
                   nbox[i][s][t] = nbox[i][s][t] + Mt[i][t][u]*box[i][s][u];
         }
        else
         for (s = 0; s <= 1; s++)
          for (t = 0; t <= 3; t++)
                nbox[i][s][t] = box[i][s][t];
       }
 } while (code != 544);

for (s = 1; s <= 3; s++)
  fscanf(inf,"%d",&face_no[i]);
 fscanf(inf,"%d",&point_no);

for (s = 1; s <= 8; s++)
  fscanf(inf,"%f",&step);

for (s = 0; s < point_no; s++)
 {
   fscanf(inf,"%f",&step);
  for (t = 0; t < 3; t++)
        fscanf(inf,"%f",&point[i][s][t]);
  point[i][s][3] = 1;

  if (multi[0] == 'Y')
```

```
        {
         for (t = 0; t <= 3; t++)
           npoint[i][s][t] = 0;

         for (t = 0; t <= 3; t++)
          for (u = 0; u <= 3; u++)
            npoint[i][s][t] = npoint[i][s][t] + Mt[i][t][u]*point[i][s][u];
        }
   else
        for (t = 0; t < 3; t++)
          npoint[i][s][t] = point[i][s][t];

    fscanf(inf,"%f",&step);
  }

 for (s = 0; s < face_no[i]; s++)
  {
   for (t = 0; t < 2; t++)
        fscanf(inf,"%d",&f_p_no[i][s]);
   for (t = 0; t < f_p_no[i][s]; t++)
        {
          fscanf(inf,"%d",&p_label[i][s][t]);
          p_label[i][s][t]--;
        }
  }
 fclose(inf);
}

/* ----------------------------------------------------------------- */
/*      Store the collision information in the linked list          */
/*     data structure.                                               */
/* ----------------------------------------------------------------- */
collision_store(i,j,cltail,md)
int  i,j;
float md[3];
NODE3 *cltail[N];
{
 int s;
 NODE3 *q;

 q = malloc(sizeof(NODE3));
 q->part = j + 1;
 for (s = 0; s < 3; s++)
  q->dd[s] = md[s];
 q->next = NULL;
 cltail[i]->next = q;
 cltail[i] = q;
}

/* ----------------------------------------------------------------- */
/*      Derive the precedence relationships based on the            */
/*     collision infromation.                                        */
/* ------------------------------------------------------- -------- */
precedence(list, node, colli)
NODE1 *list;
NODE2 *node[N];
NODE3 *colli[N];
{
 FILE *inf;
```

```
NODE1 *p, *q;
NODE2 *r;
NODE3 *c, *cltail[N];
NODE4 *u;
 int a, e, i, j, k, s, t, x, add, md[3], dis_part,
    pot_c_p[N-1], count[N+1][N-1], liaison[N-1];
 char repeat, collision;

 if (option == 1 || option == 3)
  {
     fprintf(output,"\nPrecedence Relationships :\n");
    fprintf(output,"---------------------------------------------");
  }
 if (option == 2 || option == 3)
  {
     printf("\nPrecedence Relationships :\n");
    printf("---------------------------------------------");
  }
 for (i = 1; i <= N; i++)
  for (j = 1; j <= (N-2); j++)
    count[i][j] = 0;

 for (a = 1; a <= (N-2); a++)
  {
   p = list;
   do
          {
           for (i = 1; i <= 2; i++)
            {
             q = p;
             if (i == 1)
                    {
                     dis_part = p->lpart;
                     pot_c_p[0] = p->rpart;
                     for (t = 1; t <= a; t++)
                      {
                       q = q->right;
                        liaison[t] = q->lia;
                        pot_c_p[t] = q->rpart;
                      }
                    }
             else
                    {
                     dis_part = p->rpart;
                     pot_c_p[0] = p->lpart;
                     for (t = 1; t <= a; t++)
                      {
                       q = q->left;
                        liaison[t] = q->lia;
                        pot_c_p[t] = q->lpart;
                      }
                    }

             repeat = 'n';
             if (a >= 2)
                    {
                     u = pr[p->lia][1][1]->next;
                     while (u != NULL && repeat == 'n')
                      {
```

```
         for (t = 1; t <= a; t++)
              if (u->liaison == liaison[t])
               {
                t = 999;
                repeat = 'y';
               }
         if (repeat == 'n')
              u = u->next;
        }

       if (repeat == 'n')
        {
         for (t = 2; t < a; t++)
              {
               for (s = 1; s <= count[p->lia][t]; s++)
                {
                 add = 0;
                  u = pr[p->lia][t][s]->next;
                 while (u != NULL)
                      {
                       for (x = 1; x <= a; x++)
                         if (u->liaison == liaison[x])
                         {
                              add++;
                              x = 999;
                         }
                       u = u->next;
                      }

                 if (add == t)
                      {
                       repeat = 'y';
                       s = 999;
                      }
                }
               if (s == 1000)
                t = 999;
              }
        }
     }

if (repeat == 'n')
     {
      collision = 'y';
      r = node[dis_part-1]->next;
      do
       {
         if (r->asso[0] == p->lia || r->asso[1] == p->lia)
              {
               for (t = 0; t < 3; t++)
                md[t] = r->dir[t];

               collision = 'n';
               c = colli[dis_part-1]->next;
              do
               {
                if (md[0]  == c->dd[0] && md[1]  == c->dd[1] && md[2]  == c->dd[2])
                     {
                      for (t = 0; t <= a; t++)
```

```
                          if (c->part == pot_c_p[t])
                           {
                                   t = 999;
                                   collision = 'y';
                           }
                       }

                if (collision == 'n')
                       c = c->next;
              } while (c != NULL && collision != 'y');

            if (collision == 'y')
              r = r->next;
           }
   else r = r->next;
  } while (r != NULL && collision == 'y');

if (collision == 'y' && a == 1)
 {
  e = p->lia;
  u = malloc(sizeof(NODE4));
  u->liaison = q->lia;
  u->next = NULL;
  prtail[e][a][1]->next = u;
  prtail[e][a][1] = u;
   if (option == 1 || option == 3)
          fprintf(output,"\n    ****   L%d --> L%d",p->lia,q->lia);

   if (option == 2 || option == 3)
          printf("\n    ****   L%d --> L%d",p->lia,q->lia);
 }
else if (collision == 'y')
 {
  e = p->lia;
  count[e][a]++;

  for (t = 1; t <= a; t++)
          {
           u = malloc(sizeof(NODE4));
            u->liaison = liaison[t];
           u->next = NULL;
           prtail[e][a][ count[e][a] ]->next = u;
           prtail[e][a][ count[e][a] ] = u;
          }

   if (option == 1 || option == 3)
          {
           fprintf(output,"\n    ****   L%d --> ( ",p->lia);
           for (t = 1; t < a; t++)
              fprintf(output,"L%d and ",liaison[t]);
            fprintf(output,"L%d )",liaison[a]);
          }

   if (option == 2 || option == 3)
          {
           printf("\n    ****   L%d --> ( ",p->lia);
           for (t = 1; t < a; t++)
              printf("L%d and ",liaison[t]);
            printf("L%d )",liaison[a]);
```

```
                              }
                      }
                    }
            }
           p = p->right;
          } while (p != list);
  }
 if (option == 1 || option == 3)
   fprintf(output,"\n----------------------------------------------\n\n");
 if (option == 2 || option == 3)
   printf("\n----------------------------------------------\n\n");
}

/* ---------------------------------------------------------- */
/*     Generate all the possible assembly sequences based      */
/*     on the precedence relationships.                 */
/* ---------------------------------------------------------- */
sequence(rank,seq)
int rank,seq[N+1];
{
   int  i,j,k,m,n,x,y,last,count,count1,check,temp,temp1,temp2,temp3,length[3];
  char  ans,jump,quit,skip,found,change,keep_search;
 NODE4 *u,*w[N-1];

 if (rank < N)
  {
   for (i = 1; i <= N; i++)
    {
      if (rank == 1)
       {
        for (j = 1; j <= N; j++)
         {
          jump = 'n';
          u = pr[j][1][1]->next;
          while (u != NULL && jump == 'n')
           {
             if (i == u->liaison)
             {
              j = N;
               jump = 'y';
             }
             else u = u->next;
           }
         }

        if (jump == 'n')
         {
          seq[rank] = i;
          rank++;
          sequence(rank,seq);
          rank--;
         }
       }
      else
       {
        skip = 'n';
        for (j = 1; j <= (rank - 1); j++)
         if (i == seq[j])
          {
```

```
    j = rank;
    skip = 'y';
  }

if (skip == 'n')
{
  seq[rank] = i;
  for (j = 1; j <= N; j++)
  {
    jump = 'n';
    change = 'n';
    u = pr[j][1][1]->next;
    while (u != NULL && jump == 'n')
      if (i == u->liaison)
      {
        jump = 'y';

        for (k = 1; k < rank; k++)
         if(j == seq[k])
         {
           k = rank;
           change = 'y';
         }
      }
      else u = u->next;

    if (change == 'y')
     jump = 'n';

    if (jump == 'y')
     j = N;
  }

  if (jump == 'n')
  {
    keep_search = 'y';
    if (rank < (N-1))
      temp3 = rank;
    else temp3 = rank - 1;

    for (x = 2; x <= temp3; x++)
    {
      for (j = 1; j <= N; j++)
      {
        k = 1;
        skip = 'n';
        keep_search = 'y';
        w[1] = pr[j][x][k]->next;

        while (w[1] != NULL && skip == 'n')
        {
          for (m = 1; m <= (x - 1); m++)
            w[m+1] = w[m]->next;

          count = 0;
          for (m = 1; m <= x; m++)
            for (n = 1; n <= rank; n++)
            {
              if (w[m]->liaison == seq[n])
```

```
                 {
                    count++;
                    n = rank;
                 }
             }

           if (count != x)
           {
              k++;
              if (k > 2)
               w[1] = NULL;
                else w[1] = pr[j][x][k]->next;
           }
          else
           {
               skip = 'y';
               keep_search = 'n';

               for (y = 1; y < rank; y++)
                if (j == seq[y])
                 {
                    y = rank;
                     keep_search = 'y';
                 }
           }
        }

         if (skip == 'y' && keep_search == 'n')
        {
           if (rank == (N-1))
             check = j;
          j = N;
          x = rank;
        }
    }
}

if (keep_search == 'y')
{
   rank++;
    sequence(rank,seq);
   rank--;
}
else if (rank == (N-1))
{
    for (n = 1; n <= N; n++)
                                {
                                  found = 'n';
                                  for (x = 1; x <= rank; x++)
                                             if (n == seq[x])
                                               {
                                                 found = 'y';
                                                 x = rank;
                                               }

                                  if (found == 'n')
                                             {
                                               last = n;
                                              n = N;
        }
 }

temp = abs(last - seq[rank]);
if (temp == 1 || temp == (N-1))
 {
  rank++;
  sequence(rank,seq);
  rank--;
 }
else
 {
  quit = 'n';
  temp1 = last;
  temp2 = seq[rank];
   length[1] = abs(temp1 - temp2);
   length[2] = N - length[1];

  for (n = 1; n <= 2; n++)
        {
         count = 0;
         for (x = 1; x <= 2; x++)
          {
           k = 1;
           found = 'n';
            u = pr[temp1][ length[x] ][k]->next;

           while (u != NULL)
                 {
                  do
                   {
                    if (u->liaison == temp2)
                          found = 'y';
                    else u = u->next;
                   } while (u != NULL && found == 'n');

                  if (found == 'y')
                   u=NULL;
                  else
                   {
                    k++;
                    if (k > 2)
                          u=NULL;
                     else u = pr[temp1][ length[x] ][k]->next;
                   }
                 }
           if (found == 'y')
                 count++;
          }

         if (count == 2)
          {
           quit = 'y';
           n = 999;
          }

         if (quit == 'n')
          if (n == 1)
           {
```

```
                                         temp = temp1;
                                         temp1 = temp2;
                                         temp2 = temp;
                                    }
                                   else
                                    {
                                         rank++;
                                         sequence(rank,seq);
                                         rank--;
                                    }
                                   else n = 999;
                                  }
                           }
            }
          }
        }
      }
    }
  }
 else
  {
   for (m = 1; m <= N; m++)
    {
     ans = 'n';
     for (n = 1; n < rank; n++)
      if (m == seq[n])
       {
        ans = 'y';
        n = rank;
       }

     if (ans == 'n')
      {
       seq[rank] = m;
       m = 999;
      }
    }

   if (option == 1 || option == 3)
        {
         for (j = 1; j <= (N - 1); j++)
           fprintf(output,"L%d --> ",seq[j]);
          fprintf(output,"L%d\n",seq[N]);
        }

   if (option == 2 || option == 3)
        {
         for (j = 1; j <= (N - 1); j++)
           printf("L%d --> ",seq[j]);
          printf("L%d\n",seq[N]);
        }
  }
}
```