GENERATION AND EVALUATION
OF MECHANICAL ASSEMBLY SEQUENCES
USING THE LIAISON-SEQUENCE METHOD

by

MAN-CHEUNG MAX LUI

S.B. Mechanical Engineering, M.I.T.
(1986)

SUBMITTED TO THE DEPARTMENT OF
MECHANICAL ENGINEERING
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1988

© Man-Cheung M. Lui, 1988

The author hereby grants to M.I.T. and The Charles Stark Draper Laboratory, Inc.
permission to reproduce and to distribute copies of this thesis document in whole or in
part.

Signature of Author ____________________________________________

Department of Mechanical Engineering

May, 1988

Certified by ____________________________________________________

Daniel E. Whitney, PhD.
Thesis Supervisor

Accepted by ____________________________________________________

MAY 25 1988

ARCHIVES

Massachusetts Institute
of Technology
GENERATION AND EVALUATION
OF MECHANICAL ASSEMBLY SEQUENCES
USING THE LIAISON-SEQUENCE METHOD

by

Man-Cheung Max Lui

Submitted to the Department of Mechanical Engineering
on May 27, 1988 in partial fulfillment of the
requirements for the Degree of Master of Science in
Mechanical Engineering

ABSTRACT

Sequence of assembly is a determining factor in the design of a manufacturing system. A good choice of assembly sequence can improve product quality, and reduce assembly costs. Since it may be costly to overlook a potential candidate sequence, it is desirable to select a sequence from the set of all possible sequences. However, the determination of all assembly sequences of a product by an engineer's experience or imagination alone is often impractical. Liaison Sequence Analysis, developed by De Fazio and Whitney [1986], is a practical, algorithmic method for generating all assembly sequences for a large parts-count product. This thesis describes the implementation of Liaison Sequence Analysis.

For a given design, the set of precedence relations is derived manually and used as the generating logic for all assembly sequences. Graph theory is used to describe the network of assembly, called liaison diagram, where a node represents a part, a subassembly, or a non-assembly task, and a liaison represents the relationship between two nodes, such as mating [Bourjault, 1984]. In order to eliminate any human error in the creation of the sequences from the liaison diagram and the set of precedence relations, the process is automated. Algorithmic solutions in the generation process are discussed, such as identification of a set of next-executable liaisons from a assembly state and determination of the set of liaisons that needs to be established simultaneously based on the liaison diagram. The code has been used to generate 32,436 sequences, 220 states, 724 state transitions for a mechanical product with 11 parts and 18 liaisons in about 40 seconds on an IBM XT computer. The support for editing of the sequence graph is also automated. The data structure of states in the sequence graph for facilitating the editing process is discussed. The work from this research makes it possible to generate accurately all assembly sequences from the precedence relations and to edit the sequences.

Thesis Supervisor:  Dr. Daniel E. Whitney
Section Chief
Charles Stark Draper Laboratory, Inc.
Cambridge, MA

Title:  Lecturer in Mechanical Engineering
DISCLAIMER

The views and conclusions expressed in this thesis are those of the author and should not be interpreted as necessarily representing the position of the National Science Foundation, The Charles Stark Draper Laboratory, Inc., or the Massachusetts Institute of Technology.
GENERATION AND EVALUATION
OF MECHANICAL ASSEMBLY SEQUENCES
USING THE LIAISON-SEQUENCE METHOD

by

Man-Cheung Max Lui

Submitted to the Department of Mechanical Engineering
on May 27, 1988 in partial fulfillment of the
requirements for the Degree of Master of Science in
Mechanical Engineering

ABSTRACT

Sequence of assembly is a determining factor in the design of a manufacturing system. A good choice of assembly sequence can improve product quality, and reduce assembly costs. Since it may be costly to overlook a potential candidate sequence, it is desirable to select a sequence from the set of all possible sequences. However, the determination of all assembly sequences of a product by an engineer's experience or imagination alone is often impractical. Liaison Sequence Analysis, developed by De Fazio and Whitney [1986], is a practical, algorithmic method for generating all assembly sequences for a large parts-count product. This thesis describes the implementation of Liaison Sequence Analysis.

For a given design, the set of precedence relations is derived manually and used as the generating logic for all assembly sequences. Graph theory is used to describe the network of assembly, called liaison diagram, where a node represents a part, a subassembly, or a non-assembly task, and a liaison represents the relationship between two nodes, such as mating [Bourjault, 1984]. In order to eliminate any human error in the creation of the sequences from the liaison diagram and the set of precedence relations, the process is automated. Algorithmic solutions in the generation process are discussed, such as identification of a set of next-executable liaisons from an assembly state and determination of the set of liaisons that needs to be established simultaneously based on the liaison diagram. The code has been used to generate 32,436 sequences, 220 states, 724 state transitions for a mechanical product with 11 parts and 18 liaisons in about 40 seconds on an IBM XT computer. The support for editing of the sequence graph is also automated. The data structure of states in the sequence graph for facilitating the editing process is discussed. The work from this research makes it possible to generate accurately all assembly sequences from the precedence relations and to edit the sequences.

Thesis Supervisor: Dr. Daniel E. Whitney
Section Chief
Charles Stark Draper Laboratory, Inc.
Cambridge, MA

Title: Lecturer in Mechanical Engineering
ACKNOWLEDGEMENTS

First of all, I would like to thank my academic advisor, Dr. Daniel Whitney, for his guidance and support in the development of this thesis research. His teaching has made my learning an enjoyable and rewarding experience.

I would also like to thank Dr. Thomas De Fazio, a staff member at the Charles Stark Draper Laboratory, for his input in the development and application of this research and for his help in proofreading this thesis. My appreciation is extended to other staff members, Mr. Richard Gustavson and Mr. Thomas Stepien, for their generous support in this research and in the writing of this thesis.

I would like to thank other students at Draper Lab., Rob Smith, Matt Brown, Tom Kurfess, Alan Todtenkopf, Eugene Tung, Charlie Klein, and Tom Abell for their help through this research. I wish to express my sincere gratitude to my good friends whom I met from the MIT Student House for their countless help. They are Arthur Chin, Tom Liu, Joe Nardone, Donald Yee, and Stephen Ng.

Finally, I would like to thank my parents sincerely for their endless love and support over the years. Their understanding and care always gives me the extra strength to go on in difficult times. This thesis is dedicated to my parents.

This work was funded by the National Science Foundation under Grant No. DMC-8417949 and DMC-8715800.

Man-Cheung M. Lui
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>4</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>5</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>1.  INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>13</td>
</tr>
<tr>
<td>1.2 The Need For Computerization</td>
<td>16</td>
</tr>
<tr>
<td>2.  CURRENT SYSTEM OVERVIEW</td>
<td>18</td>
</tr>
<tr>
<td>3.  COMPUTERIZED GENERATION OF ASSEMBLY SEQUENCES</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Program Structure Of liaison Sequence Generation</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Precedence Relations</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1 A Special Form of Precedence Relations</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2 Transformation of Precedence Relations</td>
<td>29</td>
</tr>
<tr>
<td>3.2.3 Simplification and Combination of Precedence Relations</td>
<td>31</td>
</tr>
<tr>
<td>3.2.4 An Alternative Method of Coding Precedence Relations</td>
<td>31</td>
</tr>
<tr>
<td>3.3 Data Structure Of Liaison Diagram</td>
<td>33</td>
</tr>
<tr>
<td>3.4 Liaison Diagram With A Loop(s)</td>
<td>35</td>
</tr>
<tr>
<td>3.4.1 Non-Redundant Cycles</td>
<td>37</td>
</tr>
<tr>
<td>3.4.2 Constructing A Spanning Tree</td>
<td>41</td>
</tr>
<tr>
<td>3.4.3 Making Basic Cycles</td>
<td>43</td>
</tr>
<tr>
<td>3.4.4 Generating All Cycles</td>
<td>44</td>
</tr>
<tr>
<td>3.4.5 Eliminating Redundant Cycles</td>
<td>45</td>
</tr>
</tbody>
</table>
3.5 Assembly Sequences Generation .................................................. 46
3.6 Chapter Summary ........................................................................ 50

4. SEQUENCE GRAPH — DISPLAY AND EDIT ........................................... 53
   4.1 Program Structure Of SED ......................................................... 53
   4.2 Structure Of Sequence Graph .................................................. 57
      4.2.1 Data Structure of States and State Transitions ......................... 59
      4.2.2 Number of Sequences ..................................................... 62
   4.3 Display Of The Assembly Sequence Graph ................................... 65
      4.3.1 Color Selection .............................................................. 65
      4.3.2 The Sequence Graph ..................................................... 68
      4.3.3 View A Selected State or State Transition ............................. 70
   4.4 Edit The Sequence Graph ........................................................ 72
      4.4.1 Criteria for Deleting States or State Transitions .................... 72
      4.4.2 Managing Deleted Items .................................................. 73
   4.5 Delete And Append (Undelete) .................................................... 79
      4.5.1 A state ........................................................................ 81
      4.5.2 A State Transition Between Two Specific States ..................... 84
   4.6 Chapter Summary ...................................................................... 84

5. RECOMMENDATIONS FOR FUTURE WORK ......................................... 87
   5.1 Providing More Features For SED ............................................. 87
   5.2 Interfacing Future Software With Other Analytical Programs ........ 95
   5.3 Automating The Generation Of The Precedence Relations ............. 95

6. CONCLUSIONS ............................................................................. 99

APPENDIX A: LIAISON DIAGRAM INPUT DATA ..................................... 101
APPENDIX B: PRECEDENCE RELATIONS SUBROUTINE CODE ............... 103
APPENDIX C: PARTIAL LISTING OF SOFTWARE CODE ......................... 107
REFERENCES ............................................................................... 135
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>26</td>
</tr>
<tr>
<td>3-2</td>
<td>30</td>
</tr>
<tr>
<td>3-3</td>
<td>30</td>
</tr>
<tr>
<td>3-4</td>
<td>38</td>
</tr>
<tr>
<td>3-5</td>
<td>46</td>
</tr>
</tbody>
</table>

- 3-1: The underlying logic of evaluating precedence relations.
- 3-2: A True-False Table of a PR, EXP → 3 AND 4.
- 3-3: A True-False Table of two transformed PR.
- 3-4: All the six possible cycles of Figure 3-7.
- 3-5: The thirteen non-redundant cycles from Figure 3-5.
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>15</td>
</tr>
<tr>
<td>Liaison Diagram of a sample assembly that consists of 10 parts and 18 liaisons</td>
<td></td>
</tr>
<tr>
<td>1-2</td>
<td>16</td>
</tr>
<tr>
<td>A network of sequence graph that represents all valid assembly paths</td>
<td></td>
</tr>
<tr>
<td>2-1</td>
<td>19</td>
</tr>
<tr>
<td>Current software system</td>
<td></td>
</tr>
<tr>
<td>3-1</td>
<td>24</td>
</tr>
<tr>
<td>Schematic diagram of the program LSG</td>
<td></td>
</tr>
<tr>
<td>3-2</td>
<td>27</td>
</tr>
<tr>
<td>Precedence Relations of an assembly of 18 liaisons</td>
<td></td>
</tr>
<tr>
<td>3-3</td>
<td>28</td>
</tr>
<tr>
<td>Flow chart of the present subroutine which tests potentially executable liaison using restricted format of PR</td>
<td></td>
</tr>
<tr>
<td>3-4</td>
<td>32</td>
</tr>
<tr>
<td>Flow chart of an alternative algorithm which tests potentially executable liaison using general form of PR</td>
<td></td>
</tr>
<tr>
<td>3-5</td>
<td>36</td>
</tr>
<tr>
<td>A sample liaison diagram. There are 11 parts and 18 liaisons in the assembly</td>
<td></td>
</tr>
<tr>
<td>3-6</td>
<td>36</td>
</tr>
<tr>
<td>A state with no subassembly</td>
<td></td>
</tr>
<tr>
<td>3-7</td>
<td>38</td>
</tr>
<tr>
<td>A graph of 6 nodes and 8 links</td>
<td></td>
</tr>
<tr>
<td>3-8</td>
<td>39</td>
</tr>
<tr>
<td>Using part-names to represent a cycle, the relationship between RC and NRC is that NRC is a subset of RC</td>
<td></td>
</tr>
<tr>
<td>3-9</td>
<td>39</td>
</tr>
<tr>
<td>Using link-number to represent a cycle, NRC can be identified by having a link-count of one plus the number of common links with a RC</td>
<td></td>
</tr>
<tr>
<td>3-10</td>
<td>40</td>
</tr>
<tr>
<td>A state with 2 subassemblies</td>
<td></td>
</tr>
<tr>
<td>3-11</td>
<td>40</td>
</tr>
<tr>
<td>Schematic diagram of the non-redundant-cycle generation module</td>
<td></td>
</tr>
<tr>
<td>3-12</td>
<td>42</td>
</tr>
<tr>
<td>Test of two nodes, x and y, for a possible loop formation</td>
<td></td>
</tr>
<tr>
<td>3-13</td>
<td>43</td>
</tr>
<tr>
<td>A spanning tree constructed from adding links of increasing numerical order based on Figure 3-5</td>
<td></td>
</tr>
<tr>
<td>3-14</td>
<td>43</td>
</tr>
<tr>
<td>Test if there is a path between nodes x and y through the building graph (not necessarily connected)</td>
<td></td>
</tr>
</tbody>
</table>
3-15 Two of the eight basic cycles constructed from Figure 3-9 by adding back deleted links. ................................................................. 44

3-16 Flow chart of the assembly-sequence generation module .................. 48

3-17 The content of the information recorded when a new state is made from the current rank ........................................................................................................... 49

4-1 Screen design of the program Sequence Edit (SED) ............................... 55

4-2 User interface of SED ............................................................................. 56

4-3 Initial screen setup of SED ....................................................................... 56

4-4 Format of the input data of SED ............................................................... 58

4-5 Structures of selected-state-list and deleted-state-list of an arbitrary rank. (a) Before any movement of states between the two lists. (b) After some movements of states between the two lists ........................................... 60

4-6 Structure of a state in the state diagram .................................................. 63

4-7 Flow chart of the function "count_seq" ................................................... 64

4-8 Flow chart of selecting the color for displaying a state .............................. 66

4-9 Flow chart of selecting the color of displaying a state transition ............... 67

4-10 Flow chart of the graphics display module which draws states and transitions from a given starting-rank to an end-rank .................................................. 69

4-11 Coordinates of the window frame for displaying a segment of the sequence graph ........................................................................................................ 70

4-12 Flow chart of displaying the View-State Window ..................................... 71

4-13 Schematic diagram of the Delete-List Window Managing Module .......... 73

4-14 Undeleting a state from the state diagram (case 1) ............................... 76

4-15 Undeleting a state from the state diagram (case 2) ............................... 77

4-16 Illustrations of pointers movement between the selected segment and deleted segment .................................................................................................................. 80

4-17 Flow chart of the subroutine "edparent" in DELETE mode ....................... 82

4-18 Flow chart of the subroutine "edparent" in APPEND mode ........................ 83

5-1 Consolidating LSG and SED through the computer operating system ....... 88

9
5-2 Incorporating a precedence-relation interpreter into the future software system ................................................................. 90

5-3 Tree representation of a Boolean Expression ......................................................................................................................... 91

5-4 Postorder representation of the expression depicted in Figure 5-1 .............................................................. 91
CHAPTER 1

INTRODUCTION

An intelligent product design is critical in today's competitive market. A product designer must consider a variety of factors such as functional goals, fabrication, manufacture, ease of use, and future repair of a product. A determining factor in the planning of a manufacturing system is the sequence of assembly, itself a function of product design. Selecting a good assembly sequence has many significant advantages, including easier assembly moves, reduced need for fixtures, less rework, better testing opportunities, less risky part mating, and lower unit assembly cost [De Fazio and Whitney, 1986]. Therefore, it is important to look for the most effective assembly sequence for a given product.

Seeking a good assembly sequence can also offer help in many different ways. Whitney et al. [1986] suggested that evaluating sequences of assembly helps a product designer to consider aspects of design other than functional goals. This approach, called the Strategic Approach to Product Design (SAPD), is feasible because assembly is an inherently integrative process that coordinates different activities such as product design, process design, fabrication, testing, quality control, and inventory control.

SAPD can reduce the lead time of the traditional design-manufacture method in which the design of the manufacturing process does not start until the final product design has been completed and product prototypes have been produced. Therefore a manufacturing engineer can only look for a good assembly sequence within the constraints of the designed product. When the sequential engineering process is replaced by an integrated one such as SAPD, the design of an assembly system can be carried out concurrently with the product design. Thus, any changes or modification in
the process and product can be made early as opposed to late in the process when a design change is either costly or impossible to make. The correction to problems detected at an early stage results in tremendous savings in the long run. Therefore, it is very useful to explore all possible assembly alternatives during the process of product design or redesign.

The final decisions associated with the production activities are highly dependent upon the chosen assembly sequence of a product. The sole factor in determining all of the possible sequences of a product, however, is the design of the product. Therefore the alternative sequences can and should be generated even before the product is fabricated or produced on the production line. Given all the generated assembly sequences, a production engineer can evaluate them based on criteria such as assembly time, ease of assembly, testing of subassembly, rework on subassembly, and assembly unit cost [De Fazio and Whitney, 1986]. This selection process involves personnel from each activity. Conflicts among the goals of the different activities are inevitable. However, at the end of the selection process, each activity group should be aware of the trade-offs being made, and a compromise decision on the assembly sequence can then be reached.

In light of the assembly sequence's importance, exploring assembly sequence alternatives becomes the first step towards an integrative design approach. People from each group can then participate in the evaluation of the design of the assembly system or in the modification of the product design. The number of sequences of a product can sometimes be so numerous that seeking all of them by imagination alone can be very difficult, if not impossible. Consequently, an engineer who attempts to find all the sequences based solely upon his experience may overlook many potential candidates. To facilitate the process of seeking assembly sequences, several sequence generation methods can be applied. The object of all the methods is to avoid making
intuitive choices of what the feasible sequences are. Once all the sequences are found, other analytical tools can be used to evaluate them and select the best candidate. One possible method of reducing the number of sequences needed for consideration can be done by eliminating awkward or difficult assembly moves and/or assembly states under user-defined criteria. Kein [1986] had done an extensive study of generating and evaluating assembly sequence alternatives of two industrial products.

1.1 BACKGROUND

Among the many different methods of assembly analysis, the precedence diagram [Prenting and Battaglini, 1964] is the one most extensively used by manufacturing engineers. A precedence diagram is a directed graph in which a node represents an assembly task and a directed edge represents the precedence order between two nodes (A \(\rightarrow\) B means A before B). It is commonly used for assembly line balancing and after a base part has already been chosen. Despite the widespread use of the precedence diagram in the industry, it cannot represent all assembly sequences of a product [De Fazio and Whitney, 1986].

The recent interest in robotic assembly and automatic generation of robotic assembly plans has led to research on automatic generation of assembly sequences. There are two basic approaches — exhaustive and algorithmic. The first approach requires exhaustive trials of finding all possible ways of either assembling parts/subassemblies or disassembling assembly/disassemblies. Ko and Lee [1987] applied exhaustive assembling trials to generate assembly sequences based on the mating conditions among the parts. Homem de Mello and Sanderson explored all assembly sequences using exhaustive disassembling trials.

The algorithmic approach was employed by Bourjault [1984] to generate all sequences. The method that Bourjault developed, called Liaison Sequence Analysis,
constitutes a set of questions addressed to the conditions of matings in the assembly represented by a graph called a liaison diagram. In the liaison diagram, all the nodes and edges are defined by the user. In many cases, a node can represent either a part, a subassembly, or a non-assembly task such as inspection and test. A numbered edge represents the relationship between two nodes. A sample liaison diagram is shown in Figure 1-1. Bourjault's questions are of two general forms:

**Question 1:** Is it true that liaison \( L_i \) can be established if liaisons \( (L_j \ldots L_k) \) have been established?

**Question 2:** Is it true that liaison \( L_i \) can be established if liaisons \( (L_j \ldots L_k) \) have not been established?

The group \( (L_j \ldots L_k) \), called the body of questions, may consist of one or more liaisons. The final form of the questions, based on the two general forms, is modulated as the user answers the questions. The answers "yes" or "no" to questions generated by Bourjault's method ultimately yield the precedence constraints of the assembly's states. The generated assembly sequences are called liaison sequences, which are represented by stages of established liaisons in the assembly.

The method of De Fazio and Whitney is a variation on Bourjault's method. Like Bourjault's method, it requires a liaison diagram for the assembly but it is different in the form of questions asked. Two questions are asked to each liaison \( L_i \):

**Question 1:** What liaisons must be established prior to establishing \( L_i \)?

**Question 2:** What liaisons must not be established until after establishing \( L_i \)?

The answers are directly the set of precedence relations for the assembly system and are either in the form of "none" or a logical combination. A sample answer is:

\[ 3 \rightarrow 4, 6 \]
Figure 1-1. Liaison Diagram of a sample assembly that consists of 10 parts and 18 liaisons.

which indicates that the condition of establishing both liaisons 3 and 4 is required before the establishment of liaison 4. The simplified technique by De Fazio and Whitney reduces the number of questions to be answered from 2(L^2+L) (to a maximum of L^2L) required by Bourjault's method to 2L, where L is the number of liaisons. This reduction makes the sequence generation practical for a product with a large liaison count. The answers to the questions are presently found manually and are the generating logic of the assembly sequences. The details of the question and answer session using these two algorithmic methods were described by De Fazio and Whitney [1986]. The sequence graph, a representation of the assembly sequences, is a network of states and state transitions represented by boxes and lines, respectively (see Figure 1-2 for a sample sequence graph). Each box is divided into cells that represent liaisons. Darken cells indicates the establishment of liaisons. Every state represents an assembly state and each state in the graph is unique. A state transition represents the
path from one state to another and there are usually multiple state transitions leading to or originating from a state. A possible assembly sequence can be visualized as a path from the unassembled state (0th rank) to the final assembled state (last rank).

![Graph representation of assembly sequences](image)

**Figure 1-2.** A network of sequence graph that represents all valid assembly paths.

### 1.2 THE NEED FOR COMPUTERIZATION

The logical form of the precedence relations coupled with the liaison diagram of the assembly allows the operator to generate a complete set of all possible assembly sequences. When this task is done manually, it is tedious and subject to human error. Thus the coding of the assembly sequences generation process is desirable. Though answering the questions from the method of De Fazio and Whitney is difficult and may not be accurate, the process of answering the questions is still done manually. Once all
the sequence are generated, aids are needed to support editing of the sequence graph due to the large number of sequences.

This thesis shows how to use the precedence relations to enumerate automatically all the sequences from the unassembled state of a product to the final assembled state. The computer support for editing the sequence graph is also presented in this thesis. Due to the effort of partial automation of the Liaison Sequence Method, the sequences can now be generated and evaluated more easily and in a shorter time span. In Chapter 2, an overview of the generation, display and editing sequences is presented. The algorithms and data structures for the generation process will be discussed in Chapter 3. In Chapter 4, the sequences display and edit features are discussed. Future work on the computerization of Liaison Sequence Method is discussed in Chapter 5.
CHAPTER 2

CURRENT SYSTEM OVERVIEW

The functional aspects of the current system are depicted in Figure 2-1. The current system is comprised of two computer programs: 1) Program LSG (Liaison Sequence Generation) for generating sequence data from the liaison diagram of a mechanical product and the precedence relations of the assembly [Whitney et al., 1988], and 2) Program SED (Sequence Edit) for displaying and editing the sequence graph generated from the LSG's sequence data.

Figure 2-1. Current software system.
The current system requires a user to provide the liaison diagram and precedence relations of the assembly. The liaison diagram is represented in a input data file which can be read by LSG (see Appendix A for detailed format) and the precedence relations are incorporated into a computer subroutine written by the user (see Appendix B for detailed program syntax). After the object code of the precedence relations subroutine has been linked with the LSG object code to produce an executable LSG program, the program can then be executed with the input data file of the liaison diagram to produce two outputs. The first output is a sequence data file to be processed by SED and the second one is a comprehensible display of the generated sequence data on the computer screen, which can also be redirected to a printer. The structure and algorithms of LSG will be discussed in Chapter 3 in detail.

In order to display or print the sequence data in a graphical form, the generated LSG's data need to be processed by SED. SED is a program that employs a mouse interface with the computer in a user-friendly interactive environment. Not only can SED display the sequence graph, it also enables a user to edit the sequence graph by removing states or state transitions from it and to print the sequence graph at any stage of the editing session. The count of sequences in the graph will be updated and displayed automatically. The list of deleted states and/or state transitions is recorded and displayed so that the deleted items can be undeleted at any time. An additional feature of SED helps the user to visualize a state in the graph and the information associated, with the state such as the list of its parent and child states.

Both computer programs, LSG and SED, are written in the C computer language, chosen primarily for its portability across other computers — an important feature for future system development — and also for its rich set of expressions [Kernighan and Ritchie, 1978]. Program LSG runs and has been successfully verified on an IBM XT computer using the Microsoft C compiler (version 4.00). This code was used for an
assembly of 11 parts and 18 liaisons; 32,436 sequences with subassemblies, and 220 states including subassembly states (number of states in each rank varied from 1 to 39) were generated in about 40 seconds. Program SED was then used to process the generated data and to edit the sequence graph. All the graphical and printing subroutines that the program SED utilizes are from the Halo Graphics Kernel and Device Drivers [Media Cybernetics] software package. The detailed discussion of the structure and various features of SED is included in Chapter 4.

In Chapter 5, a technique of consolidating LSG and SED into a single program is presented. The technique allows the liaison diagram and the precedence relations to be edited interactively, without requiring a user to edit, compile, and link subroutine code as is currently required.
CHAPTER 3

COMPUTERIZED GENERATION OF ASSEMBLY SEQUENCES

In this chapter, the structure of the software program, Liaison Sequence Generation (LSG), and the algorithmic approach to the assembly sequences generation are presented in Section 3.1. In Section 3.2, various methods of coding the precedence relations into subroutines are discussed. The data structure of the liaison diagram is discussed in Section 3.3. A method is described in Section 3.4 to identify the set of simultaneous liaisons from the liaison diagram when there is a need for establishing more than one liaison during an assembly move. Lastly, Section 3.5 presents the detailed description of how new states are generated from the wholly-unassembled state to the final assembled state.

3.1 PROGRAM STRUCTURE OF LIAISON SEQUENCE GENERATION

LSG is comprised of three modules: 1) incorporation of the precedence relations and liaison diagram to the code, 2) generation of non-redundant cycles if there are loops in the liaison diagram, and 3) generation of assembly sequences. These modules are connected as shown in Figure 3-1. Two sets of input data are needed for this system — the precedence relations and liaison diagram. The formats of these two input data files and the program file are described in Appendices A and B. The output data of LSG (from module 3) has two forms. The first form is a data file for the Sequence Edit program, SED, and the second is a screen display of the generated sequences.
Figure 3-1. Schematic diagram of the program LSG.

The first module represents the translation of the precedence relations into a subroutine code. The subroutine tests for the execution potential of a liaison given the assembly state of established liaisons. Presently, this translation is done manually by an operator who needs to know some basic C language syntax. The written subroutine code is then consolidated with other existing codes to create an executable program for
generating assembly sequences. In addition, this module creates a data file for describing the liaison diagram. This information is not necessary if there is no "loop" in the diagram or when an unconnected subassembly state is precluded.

When the liaison-count is greater than or equal to the part-count, there is at least on loop in the liaison diagram. This loop is found by module 2 before generating all the assembly sequences in module 3. The geometric information on the liaison diagram is essential to efficient sequence generation.

The executable program with the precedence-relations subroutine and the non-redundant cycles (if any) are the input to the third module. Starting from the 0th rank — the wholly-unassembled state — the code generates new states in the next rank until the finished-assembly state is reached. Afterwards, the information of all states and state transitions of the generated sequences is ready to be interpreted by the SED program which processes the data and displays and edits the graph of all the complete sequences on the computer screen, or outputs it on a printer. The program SED is described in detail in Chapter 4.

### 3.2 Precedence Relations

The logical form of a precedence relation (PR) conveniently allows a user to translate a PR into a computer code format. This translated code can then be linked with other codes to create an executable program. All PR have a common characteristic: a logical expression on both the right hand side (RHS) and the left hand side (LHS) of the relation. These two expressions can be evaluated to be either TRUE or FALSE, based on the presence of the liaisons in an assembly state. If a liaison is present, it is established and a value of "1" or TRUE is assigned. Table 3-1 shows that the only condition when a PR is violated is that LHS is FALSE and RHS is TRUE, that is, the condition from LHS does not precede the one from RHS. This logic underlies the
determination of the execution potential of a non-established liaison of a state. A liaison is a next-executable one if its establishment produces a state that satisfies all PR.

Table 3-1. The underlying logic of evaluating precedence relations. PR is satisfied only if LHS precedes RHS, i.e., the logical values of LHS and RHS cannot have the FALSE-TRUE combination.

<table>
<thead>
<tr>
<th>LHS</th>
<th>RHS</th>
<th>Is The Relation Satisfied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>Yes</td>
</tr>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>Yes</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>No</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3.2.1 A Special Form of Precedence Relations

If a set of PR has singular dependences on one liaison appearing on the RHS of all PR such as the set of sample PR shown in Figure 3-2, a special algorithm can take the advantage of this special format for coding it into a subroutine. When a liaison is tested for execution potential, it is first assumed to be established in order to check for any PR violation; hence, the RHS will always evaluate to be TRUE. Therefore the evaluation of the corresponding LHS directly determines the liaison's execution potential (Table 3-1). Figure 3-3 shows the flow chart of this subroutine algorithm. For example a PR such as:

\[ 1 \text{ AND (2 OR 3)} \to 4 \]  \hspace{1cm} (1a)

can be translated to:

```cpp
    case (4) {
    }  \hspace{1cm} (1b)
```
where && and || stand for the logical operators AND and OR respectively. The array liaison[] is a table storing the presence of all established liaisons. A value of "1" or TRUE indicates a liaison's presence and "0" or FALSE for its absence. When liaison 4 is tested for its execution potential, the PR subroutine will branch to case (4) for liaison 4 and evaluate the corresponding expression. If the table, for example, contains a "1" for liaison[1], 0 for liaison[2], and 1 for liaison[3], the expression (1b) will be evaluated to be FALSE which will be returned to the calling routine — liaison 4 is not an executable one.

<table>
<thead>
<tr>
<th>Expression</th>
<th>→</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6 OR (3 AND 8))</td>
<td>→</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>→</td>
<td>2</td>
</tr>
<tr>
<td>((3 OR 14) AND (14 OR 17) AND (3 OR 17))</td>
<td>→</td>
<td>5</td>
</tr>
<tr>
<td>AND (15 OR 18) AND (4 OR (16 AND 17))</td>
<td>→</td>
<td>7</td>
</tr>
<tr>
<td>(1 OR 3)</td>
<td>→</td>
<td>8</td>
</tr>
<tr>
<td>(8 OR (3 AND 14))</td>
<td>→</td>
<td>9</td>
</tr>
<tr>
<td>(9 OR (8 AND 15) OR (3 AND 14 AND 15))</td>
<td>→</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>→</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>→</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>→</td>
<td>13</td>
</tr>
<tr>
<td>(17 OR (4 AND (3 OR 14))</td>
<td>→</td>
<td>16</td>
</tr>
<tr>
<td>(15 OR 18)</td>
<td>→</td>
<td>17</td>
</tr>
</tbody>
</table>

**Figure 3-2. Precedence Relations of an assembly of 18 liaisons.**

This PR-to-code translation method assumes that each liaison can have no more than one precedence condition. That is, no liaison appears more than once on the RHS of all PR. If a liaison does not have any precedence constraint, the test for its execution potential is always evaluated to be TRUE by default. The advantage of the
The existing algorithm is that the maximum efficiency is now achieved because there is at most one expression (LHS) needed to be evaluated for any liaison.

**Figure 3-3.** Flow chart of the present subroutine which tests potentially executable liaison using restricted format of PR.
3.2.2 Transformation of Precedence Relations

On the other hand, the assumption that the right hand side of a precedence relation depends on one liaison only is not always true as demonstrated in the following example:

\[
\text{EXP} \rightarrow 3 \text{ AND } 4 \tag{2}
\]

where EXP represents any logical expression constituted of evaluating a set of liaisons using logical operators. The valid and invalid combinations of all logical values (True or False) of EXP, 3, and 4 are summarized in Table 3-2. For the PR of (2), it can be transformed into two PR as follows.

\[
\text{NOT } 4 \text{ OR (EXP)} \rightarrow 3 \tag{3a}
\]

and

\[
\text{NOT } 3 \text{ OR (EXP)} \rightarrow 4 \tag{3b}
\]

Liaisons 3 and 4 now have their own precedence conditions — LHS of (3a) and (3b) respectively. Table 3-3 shows that the two new PR, (3a) and (3b), gives the same results as (2). This transformation is possible because (2) does not allow LHS to be FALSE and RHS to be TRUE (Table 3-1); hence, liaisons 3 and 4 must not be both TRUE at the same time when LHS is FALSE. The Boolean logic underlies this transformation of (2) to (3a) and (3b) for liaisons 3 and 4.

Another PR of similar pattern is:

\[
(\text{EXP}) \rightarrow 5 \text{ AND } 6 \text{ AND } 7 \tag{4}
\]

and (4) can be transformed to:

\[
\text{NOT } 6 \text{ OR NOT } 7 \text{ OR (EXP)} \rightarrow 5 \tag{5a}
\]

\[
\text{NOT } 7 \text{ OR NOT } 5 \text{ OR (EXP)} \rightarrow 6 \tag{5b}
\]

\[
\text{NOT } 5 \text{ OR NOT } 6 \text{ OR (EXP)} \rightarrow 7 \tag{5c}
\]
Table 3-2. A True-False Table of a PR, EXP → 3 AND 4, where EXP is a logical expression.

<table>
<thead>
<tr>
<th>EXP</th>
<th>3</th>
<th>4</th>
<th>3 AND 4</th>
<th>Is EXP → 3 AND 4 satisfied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>No ←</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3-3. A True-False Table of two PR transformed from the PR EXP → 3 AND 4.

<table>
<thead>
<tr>
<th>EXP</th>
<th>3</th>
<th>4</th>
<th>NOT 3</th>
<th>NOT 4</th>
<th>NOT 3 OR EXP → 4</th>
<th>NOT 4 OR EXP → 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>No</td>
<td>No ←</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
3.2.3 Simplification and Combination of Precedence Relations

If all the RHS of PR depend on one liaison and there are multiple PR which have the same RHS of PR, L4, some simplification and combination can be made. If EXP 1 and EXP 2 are two distinct logical expressions, then some examples are:

Example 1) 
\[(\text{EXP 1}) \rightarrow L_4\] 
\[(\text{EXP 2}) \rightarrow L_4\]
becomes 
\[(\text{EXP 1}) \text{ AND } (\text{EXP 2}) \rightarrow L_4\]

Example 2) 
\[(\text{EXP 1}) \rightarrow L_4\]
\[(\text{EXP 1}) \text{ OR } (\text{EXP 2}) \rightarrow L_4\]
becomes 
\[(\text{EXP 1}) \rightarrow L_4\]

Example 3) 
\[(\text{EXP 1}) \rightarrow L_4\]
\[(\text{EXP 1}) \text{ AND } (\text{EXP 2}) \rightarrow L_4\]
becomes 
\[(\text{EXP 1}) \text{ AND } (\text{EXP 2}) \rightarrow L_4\]

3.2.4 An Alternative Method of Coding Precedence Relations

Combining multiple PR into a single one can increase the efficiency of the subroutine because of the reduced number of expressions needed to be evaluated. However, the restriction of the appearance of a single liaison on the RHS of a PR leads to the transformation of some PR into multiple PR. To remove this restriction and hence the elimination of this transformation process, one can choose another approach to test for the execution potential of a liaison when more than one liaison appear on the RHS of a PR (see flow chart in Figure 3-4). The execution potential of the liaison i is determined by the answer from each PR which has the occurrence of liaison i on the RHS. If none of the PR is violated (RHS is not TRUE and LHS is not FALSE), the liaison i is executable in the next rank or else it is non-executable.
Figure 3-4. Flow chart of an alternative algorithm which tests potentially executable liaison using general form of PR.
This alternative algorithm is less efficient than the currently employed one, since two expressions (LHS and RHS) are needed to be evaluated. Nevertheless, a set of PR can be coded easily after waiving the requirement of combining, simplifying, or transforming any PR.

3.3 DATA STRUCTURE OF LIAISON DIAGRAM

A liaison diagram describes the relationship among parts in an assembly. It is represented by a list of liaison-numbers and the two corresponding part-numbers which are stored in a data file. This data file, written by an operator, can then be read by the computer. The present code assumes that: (i) all the nodes (parts) are connected to form a single connected graph, (ii) there is at most one unique link (liaison) between two distinct nodes (parts), and (iii) there is no link with the node itself (no self loop). For convenience, the liaison label and part label are numbered and are used directly as the indices of the following three data structures: 1) liaison table, 2) part table, and 3) part-to-part table. The numbered labels which are determined by the user are not necessarily arranged in a consecutive order.

The second and third tables are all derived from the liaison Table so all of them are related to each other. These tables are created to facilitate the computation speed for various parts of the non-redundant-cycles-generation module (Section 3.4) and assembly-sequences-generation module (Section 3.5).

Liaison Table

This Table has almost the same format as the input data of the liaison diagram from the operator.

<table>
<thead>
<tr>
<th>Index</th>
<th>Exist</th>
<th>Part x</th>
<th>Part y</th>
</tr>
</thead>
<tbody>
<tr>
<td>liaison # of lab</td>
<td>T or F</td>
<td>part # a</td>
<td>part # b</td>
</tr>
</tbody>
</table>
The index of the liaison table is the liaison-number of $l_{ab}$. A value of TRUE (T) or FALSE (F) is assigned to the field "exist" if the liaison-number appears in the liaison diagram. The fields "Part x" and "Part y" contain the two part numbers corresponding to $l_{ab}$.

**Part Table**

<table>
<thead>
<tr>
<th>Index</th>
<th>Exist</th>
<th>List Count</th>
<th>Adjacent Part List</th>
<th>Adjacent Liaison List</th>
</tr>
</thead>
<tbody>
<tr>
<td>part 1</td>
<td>T or F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Part table is a look up table using the part-number, $i$, as an index. The field "Exist" contains the flag to indicate if part $i$ appears in the liaison diagram. List count refers to the number of adjacent elements (parts and corresponding liaisons) connected to part $i$. The adjacent part list and adjacent liaison list contain the part-number and corresponding liaison-number connected to part $i$ respectively.

**Part-to-Part Table**

Part-to-Part table is a look up table for finding the liaison-number for any two part-numbers which are the indices of the table. Because the entries in the table are diagonally symmetric, only the upper half of the table is stored in the computer memory.

```
          j
         /     \
 i | | | | | | | |
    . . . . . . .
```

Therefore this two dimensional array with index $k$ can then be transformed to a one dimensional array using the following formula for locating the entry $(i, j)$:

34
\[ k = (l - 1)(\text{Maxparts} - 1/2) + j \]

where \text{Maxparts} is the number of columns in a row, and the entry \( k(l, j) \) contains the corresponding liaison \# \( l_j \) of part \( i \) and part \( j \).

3.4 LIAISON DIAGRAM WITH A LOOP(S)

Often, an assembly can be highly integrated, and it can have a liaison-count that is much greater than one less than its part-count (see Figure 3-5 — a liaison diagram from an assembly of rigid parts only. Its corresponding PR is from Figure 3-2). This means that more than one liaison may need to be established in one or more state transitions. Assuming that all liaisons between two subassemblies, or a part and a subassembly, have to be established simultaneously, the number of ranks of the generated sequences is equal to the total number of parts in an assembly. This assumption is necessary for an assembly consisted of rigid parts only and when the transition between two states only characterizes the complete mating of two parts, or two subassemblies, or a part and a subassembly. For example, in Figure 3-6, when part L is mated with the assembled part (nodes that are connected by solid lines), liaisons 5, 17, and 18 (dotted lines) are required to be established simultaneously. Establishing liaisons 5, 17, and 18 is the state transition.

Even though the mating of those simultaneously established liaisons may require some order, the respective state transition does not give any information about the order of the establishment of liaisons. On the other hand, if some of the parts of an assembly are flexible and do not require or allow simultaneous establishment, there will be more ranks in which flexible parts do not have all their liaisons with other parts established. Therefore the maximum number of ranks of an assembly sequence can be equal to the assembly's liaison-count. By default, the present system assumes...
that an assembly consists of all rigid parts only and must satisfy the simultaneous establishment requirement.

Figure 3-5. A sample liaison diagram. There are 11 parts and 18 liaisons in the assembly.

Figure 3-6. A state with no subassembly. When part L is mated with the assembled parts (nodes that are connected by solid lines), liaisons 5, 17, and 18 (dotted lines) are required to be made simultaneously.
If a product contains liaisons that must be done simultaneously, the liaison
diagram will contain loops or cycles. If there are m links (liaisons) which constitute a
cycle, there are also m nodes (parts) along that cycle. When m-1 of these liaisons have
been established along that cycle, all m parts are in fact mated. By the earlier
assumption of simultaneous establishment of liaisons, the last m\textsuperscript{th} liaison must be
established as well.

It is very time consuming for a computer to search for the m\textsuperscript{th} undone link
along a loop when the loop count is large. A more efficient method is now presented.
First, generate all possible loops (Korfhage, 1984) and a list of the liaison-numbers
along each loop. Now every time a liaison, i, is established, the liaison-count of each
loop that contains i is decreased by one. When one of the loops' liaison-counts is down
to one, the last liaison can then be looked up from the list of liaison-numbers for that
loop. There are two advantages of using this method. First, it eliminates the process of
searching for the m\textsuperscript{th} liaison along one of the loops of the liaison graph whenever an
executable liaison is added to a state to form a new state of the next rank. Second, the
liaison-counts of undone liaison of each loop can also be carried forward to the child
state fathered by the current state.

3.4.1 Non-Redundant Cycles

When the total number of possible cycles is very large, it is undesirable to find
the last m\textsuperscript{th} liaison, especially when the size of computer memory is limited. To
address this problem, the concept of non-redundant cycles (NRC) is employed to
minimize the number of cycles that need to be monitored. Each NRC has the minimum
number of nodes to form a cycle. When the last liaison is found in one NRC, its
consequent establishment can trigger more last liaisons in other NRC. It is possible
because a last liaison appears in multiple NRC. Each NRC is usually made of a set of
nodes that is also a subset of nodes of some other cycles which are called redundant cycles (RC). RC are eliminated by NRC. Figure 3-7 is a sample graph of 6 nodes and 8 links. It has 6 possible cycles listed in Table 3-4. Figure 3-8a shows how cycle 5 can be eliminated by either cycle 3 or 4 when cycles 3 and 4 are the two subsets of cycle 5. Similarly, cycle 6 can also be eliminated either by cycles 2 or 3 as shown in Figure 3-8b. On the other hand, if links are used as elements to describe a cycle, RC can also be identified by having one minus the count of common links with NRC. Cycle 5 has intersections, shown in Figure 3-9a, of links 1, 2, and 7 with cycle 4; links 3 and 8 with cycle 3. Figure 3-9b shows the similar relationships for cycle 6 with cycles 2 and 3.

![Graph of 6 nodes and 8 links](image)

**Figure 3-7.** A graph of 6 nodes and 8 links.

**Table 3-4.** All the six possible cycles of Figure 3-7.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Link List</th>
<th>Node List</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 4, 5</td>
<td>A, B, C, D</td>
<td>NRC</td>
</tr>
<tr>
<td>2</td>
<td>4, 5, 6, 7</td>
<td>B, C, D, E</td>
<td>NRC</td>
</tr>
<tr>
<td>3</td>
<td>3, 6, 8</td>
<td>B, E, F</td>
<td>NRC</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 6, 7</td>
<td>A, B, D, E</td>
<td>NRC</td>
</tr>
<tr>
<td>5</td>
<td>1, 2, 3, 7, 8</td>
<td>A, B, D, E, F</td>
<td>RC</td>
</tr>
<tr>
<td>6</td>
<td>3, 4, 5, 7, 8</td>
<td>B, C, D, E, F</td>
<td>RC</td>
</tr>
</tbody>
</table>
Figure 3-8. Using part names to represent a cycle, the relationship between RC and NRC is that NRC is a subset of RC.

Figure 3-9. Using link-number to represent a cycle, NRC can be identified by having a link-count of one plus the number of common links with a RC.

This use of NRC not only enhances the speed of finding out the last liaisons along a loop when a part is mated to a partial assembly, but also helps to efficiently locate all liaisons between two subassemblies without referring to the liaison diagram to find the loops. Figure 3-10 shows a state of two subassemblies, \( \{A, B, C, D, E, F\} \) and \( \{G, H, J, K, L\} \). The establishment of liaison 3 triggers the subsequent requirement of establishing liaisons 4, 5, 8, 9, and 10 (found from the NRC of Figure 3-5) without violating any PR.
The making of NRC consists of four linked processes: 1) Constructing a spanning tree, 2) Making basic cycles, 3) Generating all cycles, and 4) Selecting NRC. Figure 3-11 is the schematic diagram of this process.

![Diagram](image)

Figure 3-10. A state with 2 subassemblies. Six liaisons \(\{3, 4, 5, 8, 9, 10\}\) may need to be established in some preceding order to reach the final-assembled state.

![Flowchart](image)

Figure 3-11. Schematic diagram of the non-redundant-cycle generation module.
3.4.2 Constructing A Spanning Tree

A spanning tree is a connected graph with no loop. Two methods can be used to make a spanning tree from a connected graph of k links and n nodes: adding links to an empty graph, or deleting links from the existing graph. The former method adds links of increasing numerical order to an empty graph until the link-count of the building graph is equal to one minus the node-count of the existing graph. It is not necessary to maintain the connectivity of the building graph but it is required that the insertion of a link does not yield a loop. The latter method is not chosen because it requires the computer to check for its graph-connectivity every time a link is removed — a time consuming process. In contrast, the former method has a lesser number of these connectivity checks than the latter method does. The former method can check whether the insertion of a link, $l_{xy}$ between nodes $x$ and $y$, to the building graph may yield a loop by testing the following two possible cases first: 1) $l_{xy}$ is attached to an end of a graph, or 2) $l_{xy}$ is not attached to any other links. A loop cannot be yielded if the building graph is of case (1) or (2). A loop is possible only when there are links already attached to both nodes $x$ and $y$ as shown in Figure 3-12. In that case, a function, "loop_test", will then be used to confirm the formation of a loop; hence, a spanning tree can be built by avoiding the insertion of a link that can yield a loop. Figure 3-13 is one of the possible spanning trees of Figure 3-5.

The algorithm of "loop_test" is explained as follows: For the nodes $x$ and $y$ of link $i$ from the building graph, first take node $x$ as the root of the LHS tree and node $y$, of the RHS tree (See Figure 3-14). Second, find the respective leaves of each tree. Choose the tree that has a lesser number of roots as the working tree. The root’s node-number is entered in every found leaf’s path-array so that each leaf will know its root. The code chooses again either side of the trees which has a lesser number of leaves as the working tree. These new level leaves become the roots and then start searching for their leaves.
This recursive process and the process of switching to a tree with a lesser number of
leaves as the working one are repeated until either (i) a node that is visited by the other
tree is found, or (ii) when all the leaves are found to be dead ends. Case (i) indicates that
a loop is found but not for case (ii) even after link 1 is added to the building graph. This
algorithm is efficient when the graph is highly branched.

![Diagram](image)

(a) case 2.

(b) case 1.

(c) case 1.

(d) addition of $L_{xy}$ may yield a loop.

Figure 3-12. Test of two nodes, $x$ and $y$, for a possible loop formation (solid and dotted
tlines stand for the presence and absence of links, respectively). If $L_{xy}$ is
added to a building graph (not necessarily connected), only (d) may yield
a loop.
3.4.3 Making Basic Cycles

Once a spanning tree is created, the deleted k-n+1 links are added back to the spanning tree one at a time to create the k-n+1 basic cycles. The function "loop_test" is used again to find the list of liaison-numbers along each basic cycle. Figure 3-15 shows two of the eight basic cycles.
3.4.4 Generating All Cycles

There are at most $2^{k-n+1} - 1$ possible cycles that can be generated from all the combination of the basic cycles. The combination can be represented from the position of the bits of a binary number which are ON. For example, 7 has the 00000111 eight bits format which can represent the combination of the basic cycle sets 1, 2, and 3. To obtain all the combination, simply increase a number from 1 to $(2^{k-n+1} - 1)$ by one and
find each bit-field combination of the corresponding basic cycle sets. The set of link-numbers that constituted the resulting cycle is determined by applying the operation of "Exclusive OR" to all the link-numbers from each basic cycle of a given combination.

Exclusive OR is an operation that takes the combinations FALSE-FALSE to be FALSE, TRUE-FALSE to be TRUE, FALSE-TRUE to be TRUE, but TRUE-TRUE to be FALSE. Each resulting graph can have three different forms:

(i) one loop, that is, link-count is equal to node-count, or
(ii) multiple loops that share one or more common nodes, or
(iii) two or more separate loops.

To eliminate a graph of case (ii), count the number of nodes in the graph. If the node-count is not the same as the link-count, the resulting graph is of case (ii). A graph of case (iii) can only be identified by counting the number of links along a loop in the graph; hence, if it is less than the number of links in the resulting graph, it is of case (iii) form. There are 79 possible cycles in Figure 3-5.

3.4.5 Eliminating Redundant Cycles

To find NRC from all the possible cycles, "AND" two link-lists, loop A and loop B, and find the common links. If the common-link-count is equal to the link-count in loop A minus one, loop B is redundant and it will be eliminated from the comparison list. Loop A is not guaranteed to be a NRC until it is compared with the rest of the loops in the comparison list. Thus a loop may eliminate or be eliminated from the comparison list and the surviving loops constitute the set of NRC. Table 3-5 shows the 13 NRC of Figure 3-5.
Table 3-5. The thirteen non-redundant cycles from Figure 3-5.

<table>
<thead>
<tr>
<th>Non-Redundant Cycle</th>
<th>Link Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 5 6 10 18</td>
</tr>
<tr>
<td>2</td>
<td>1 5 6 9 17</td>
</tr>
<tr>
<td>3</td>
<td>1 4 6 9 16</td>
</tr>
<tr>
<td>4</td>
<td>4 5 16 17</td>
</tr>
<tr>
<td>5</td>
<td>9 10 15</td>
</tr>
<tr>
<td>6</td>
<td>15 17 18</td>
</tr>
<tr>
<td>7</td>
<td>3 5 14 17</td>
</tr>
<tr>
<td>8</td>
<td>3 4 14 16</td>
</tr>
<tr>
<td>9</td>
<td>6 7 12</td>
</tr>
<tr>
<td>10</td>
<td>1 2 11</td>
</tr>
<tr>
<td>11</td>
<td>1 3 6 8</td>
</tr>
<tr>
<td>12</td>
<td>3 5 8 10 18</td>
</tr>
<tr>
<td>13</td>
<td>8 9 14</td>
</tr>
</tbody>
</table>

3.5 ASSEMBLY SEQUENCES GENERATION

Figure 3-16 is the flow chart of the assembly-sequence-generation module. The minimum number of ranks equals the part-count minus one so that each state transition describes the complete mating of parts or subassemblies. An extra rank is needed for each liaison associated with a flexible part when its mating does not require or allow the simultaneous establishment of all liaisons with the rest of the partial assembly. As a result, the number of generated ranks equals the rigid-part-count plus the count of liaisons associated with flexible parts minus one. Starting from the wholly-unassembled state or the 0th rank, treated as the parent rank, find all executable liaisons using the PR subroutine by providing the current state of established liaisons. A returning value of TRUE for a liaison-number stands for its execution potential and FALSE for its non-execution potential. If no executable liaison
is found and the state is not a final-assembled state, then the product cannot be
assembled, or the PR are erroneous, and either way an error message will be displayed
to the user.

The NRC are used to find all the last liaisons when undone-liaison-count along a
NRC is down to one after establishing a liaison i. If some of them appear in the next-
executable liaison list, delete them from the list. Any last liaisons that do not appear in
the list will be cross checked with the PR for execution potential because the order of the
last liaisons establishment is assumed to be essential. At this point, if the liaison
involves a flexible part, this requirement can be suppressed in a future system.
Otherwise, the PR will have an error if all the last liaisons cannot be established
simultaneously.

Initially, all the next-executable liaisons are obtained by testing each
individual liaison using the PR subroutine. However, it is possible that some of the
generated simultaneous liaisons cannot be tested individually and therefore cannot be
obtained during the course of PR testing. In order to ensure that all the simultaneous
liaisons are truly executable, a subroutine, "rules_check", which has two levels of error
checking, is used to double check the execution potential of the entire set of
simultaneous liaisons. "Rules_check" first checks if all the liaisons can be executed in
some possible preceding orders. If some of the liaisons fail to do so in this first test, a
second test is done to test for the presence of a single PR that restrain the execution of
the entire set of simultaneous liaisons. However, if the second test fails but a third test
confirms that the generated state does not violate any PR, the generated state is a
permissible one. In this case, the generated state will be included in the sequence graph
but a warning message will be displayed to caution the user that the liaisons of the
generated state are not done in a preceding order. On the other hand, if the generated
state is not even a legitimate one, there is a possible PR error — some of the liaisons can
be executed independently while other liaisons of the same set of simultaneous liaisons
are not permitted to be executed. Therefore the generated state will be excluded from the
graph.

Figure 3-16. Flow chart of the assembly-sequence generation module.
When a state is created in the next rank, it is compared to all the existing states in the next rank to see if it is indeed a new state before adding it to the rank. Two basic pieces of information are recorded in the current state: the list of executable liaisons established to make the new state, and the location of the new states (column number). Three sets of information are recorded in each new state: 1) the table of liaisons, \( p_{\text{in\_tab}} \), established to arrive at the new state; 2) the state's established liaisons, and 3) undone-liaison-count of each NRC. Figure 3-17 shows how these two sets of information are related to the parent state and its child states.

![Diagram showing the process of making liaisons and calculating the new state's location and liaisons]

**Figure 3-17.** The content of the information recorded when a new state is made from the current rank.

Since it is possible to have more than a parent state or child state for any state, the table \( p_{\text{in\_tab}} \) may not be able to represent an exact state transition for a state. For
example, \texttt{p	ab{\textunderscore i	ab{\textunderscore n}} can have the value ON for both the liaisons 1 and 2 to represent the two state transitions of liaisons 1 and 2. The two state transitions share the same table. The purpose of \texttt{p	ab{\textunderscore i	ab{\textunderscore n}} is to facilitate the search of a particular state transition. For example, in order to search for a state transition of liaison 1 to one of the states in a particular rank, the operator "Inclusive OR" is applied to every state's \texttt{p	ab{\textunderscore i	ab{\textunderscore n}} with a mask of which liaison 1 is ON. If the resultant table with a state is the same as the mask itself, the state transition is found at that state. An alternative but slower search is to apply the operator "Exclusive OR" to each state's \texttt{ltai	ab{\textunderscore n}} in the rank with the \texttt{ltai	ab{\textunderscore n}} of every parent state of the state, and match the resultant table with the state transition. This search is particularly inefficient when there are multiple parent states for a state.

After all the states from the current rank (parent rank) have finished generating new states for the next rank (child rank), the child rank becomes the parent rank and starts creating its own child rank. This process is repeated until the final-assembled state is made.

3.6 CHAPTER SUMMARY

The program structure of LSG was presented in this chapter. Various methods of coding Precedence Relations, PR, were discussed. Each method has its own advantages and disadvantages and the decision of which method should be adopted depends on the form of the PR. If necessary, some PR needs to be combined into a single PR or a single PR needs to be transformed into several PR. The presence of loops in a liaison diagram leads to the need of efficient identification of the set of simultaneous liaisons at computer run time. The method of using Non-Redundant Cycles, NRC, to solve this problem was introduced. In addition, the process of generating NRC from 1) constructing a spanning tree, 2) making basic cycles, 3) generating all cycles, and 4)
eliminating redundant cycles were discussed. Lastly, by utilizing the PR checking subroutine and the set of NRC, the algorithm of the assembly sequences generation was presented.

In the next chapter, the structure of the program SED is discussed. SED can display and edit the sequence data generated from LSG described in this chapter. The features of SED and the mechanisms of its functions are also presented in detail.
CHAPTER 4

SEQUENCE GRAPH — DISPLAY AND EDIT

The sequence data generated by the program LSG (Liaison Sequence Generation) are processed by a program called SED (Sequence EDIt) so that the generated sequences can be displayed on the screen or printed through a printer. Another important purpose of SED is to allow a user to edit the liaison sequences by first selecting and then deleting states and/or state transitions. SED also provides an interactive graphics environment and mouse control for user-interface. The structure of SED and the interactions between windows on the screen is described in Section 4.1.

The input data from LSG and its placement in the data structures of the sequence graph is described in Section 4.2. Section 4.2 also describes how the data structures represent the relationships between a state with its neighboring states of the same rank, its parent states from the preceding rank and its child states in the next rank.

One of the important features of SED is the ability to display the sequence graph (the entire graph or a segment of it) within a prescribed area. This feature is described in Section 4.3. Another important feature is that a user can select a state or a state transition and delete it. This feature is discussed in Section 4.4 along with a discussion of the process of recovering a deletion. In Section 4.5, the mechanisms of the delete and undelete processes are presented.

4.1 PROGRAM STRUCTURE OF SED

SED is an interactive graphics program which is currently configured to use a mouse as its input device. The screen is divided into six basic zones: the Pull-Down
Menu, Icon-Menu, Zoom-In Sequence-Graph Window, Zoom-Out Sequence-Graph Window/View-State Text Window, View-State Window, and Delete-List Window (see Figure 4-1). By tracking the zone in which the mouse is located, control is passed to the module corresponding to the zone (see Figure 4-2). Figure 4-3 shows a schematic diagram of the SED start-up process. Since all the subsequent actions executed from the module may lead to some change of the information displayed in some other windows, updating the display of those windows is also part of the responsibility of each module. The purpose of each screen window is summarized below:

Sequence-Graph Windows have two forms—Zoom-In and Zoom-Out. Both windows display the sequence graph in a compact diamond form but the Zoom-In one provides a detail view of each state and possibly some state transitions. The Zoom-Out one, on the other hand, by fitting the entire graph inside the window, provides a view of the entire graph but may not provide a clear view of each state or state transition. Both windows, however, provide the means of selecting a state for deletion or further examination through the View-State Window/View-State Text Window.

The View-State Window allows an isolated view of a state selected from the Sequence-Graph Window. Furthermore, all the parent and child states of the selected state are displayed. If the View-State Text Window is currently on the screen, a text description of the liaisons and their associated part names are displayed. If a state transition is selected from the View-State Window and placed in the View-State Text Window, a description of the state transition is also shown.
Figure 4-1. Screen design of the program Sequence Edit (SED).

The Delete-List Window is a module which manages all the deleted items. After a state or state transition has been deleted from the sequence graph, it is appended as an item to the delete list and displayed in the Delete-List Window. The item can also be undeleted by returning the icon of the type of the deleted item to the Sequence-Graph Window.
Figure 4-2. User Interface of SED.

Figure 4-3. Initial screen setup of SED.
4.2 STRUCTURE OF SEQUENCE GRAPH

The format of the input data produced by LSG is shown in Figure 4-4. The first six numbers in the data file are:

1) the number of states,
2) number of state transitions between two states,
3) number of ranks,
4) maximum number of states in any rank of the graph,
5) the highest liaison #, and
6) the highest part #.

The first five numbers are mainly used for determining the number of blocks that are going to be read in the remaining data file. These numbers are also used for allocating the computer memory for the array of structures used in the program. The six numbers of the beginning of the data file are followed by two blocks of data. The first block is subdivided into sub-blocks that are determined by the number of ranks. Each sub-block of data is headed by the rank number and total number of states in that rank and followed by groups of data for each state in that rank as illustrated in Figure 4-4. Starting from the column at the far left in the rank, for each state, ST, each group consists of the following:

1) \( p_{in\_tab} \) – a table of liaisons needed to be established to create ST,
2) \( liai\_tab \) – a table of established liaisons of ST,
3) \( p_{out\_tab} \) – a table of next-executable liaisons of ST in the next rank, and
4) \( nchild \) – the number of ST's child states in the next rank.

The last block of data in the data file is a list of column numbers of all the child states of every state in the graph. To locate all the parent states for every state except the one and only unassembled state in the 0th rank, SED checks if a state, ST, finds itself the child state of another state, \( ST_p \), in the preceding rank. If a match is found, the parent state \( ST_p \) is then stored in data structure of the state ST.
\textit{nstate} - total number of states
\textit{nstatran} - total number of state transitions
\textit{nrank} - total number of ranks
\textit{maxcol} - maximum number of states in any rank
\textit{liaimax} - highest liaison #
\textit{partmax} - highest part #

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4-4.png}
\caption{Format of the input data of SED.}
\end{figure}
The structure of the sequence graph is designed in a state domain. Every state in the graph is represented by a structure that links with the state's neighboring states in the same rank, parent states and child states. The representation is therefore explicit. A state transition, on the other hand, is implicit because it is represented by the context inside the structures of the two involved states. In Section 4.1.1, the data structure of states and the technique of accessing state transitions will be described. The method of counting the number of sequences will be presented in Section 4.1.2.

4.2.1 Data Structure of States and State Transitions

The network of the sequence graph is arranged systematically in two forms. First, there is a structure called RANK that links all the states that belong to the same rank. All the states belong to either one of the two categories, Select or Delete, and they are linked according to the category that they belong to. When the input sequence data are read, all the states are placed in the selected-state-list. Second, within the structure of each state in the graph, there are two indirect pointers pointing to the lists of the state's parent and child states respectively. These two arrangements efficiently describe the left-right (states of the same rank) and up-down (parent and child states) relationships of the sequence graph.

The structure RANK (illustrated in Figure 4-5) has the following members-list:

1) a head pointer pointing to the first selected state, ptr, and if there is none in the selected-state-list, the pointer has a value "NULL".

2) a head pointer pointing to the first deleted state, dptr, and if there is none in the selected-state-list, the pointer has a value "NULL".

3) the number of states in the selected-state-list, ncol.

An array, rank, of structure RANK is then used to link all the states of each rank into either the selected-state-list or deleted-state-list according to the categories that the
states belong to. Both lists have similar structures — every state in a list has a pointer pointing to the next state in the same list. The last state of the list will have a value "NULL" assigned to its next-state-pointer.

(a)

(b)

Figure 4-5. Structures of selected-state-list and deleted-state-list of an arbitrary rank. (a) Before any movement of states between the two lists. (b) After some movements of states between the two lists.

This linked-list structure is chosen because of its dynamic property of ordering states in a list. A state can be placed anywhere in the list or even switched to a list of
different category simply by redirecting all the involved states' next-state-pointer during the move. For example, four states A, B, C, and D are placed in the order of A→B→C→D — state A points to B, B points to C, and C points to D. Now if the desired positions of the list is A→C→B→D, one way of accomplishing this order change is to change the state B's next-state-pointer from C to D; the state A's next-state-pointer from B to C; and finally the state C's next-state-pointer from D to B. The advantage of this linked-list method is that it eliminates the process of transporting a state's information around if the state needs to be reordered or switched to another list. The order of states in the selected-state-list also reflects the order in which they are displayed on the screen — states are displayed from the left to the right.

Each state is actually a structure called SELECT (see Figure 4-6). SELECT monitors the relationship of a state with its neighboring state (the one immediate to the right of the same rank) , its parent and child states. The state structure is described as the following:

1) **status**: designates the status of the state structure. It can have value either ON, HILITE or OFF. ON means that the state is presently in the select-state-list and displayable. HILITE denotes the absence of the state in the graph but for special display purpose it can still be displayed on the screen but in a highlighted color. OFF represents both the absence of the state in the graph and on the display.

2) **col**: the column number at which the state is displayed (the lowest value of col is 0). However, when the state's status is OFF, the column number is changed to a value of -1 to indicate that the state is in the deleted-state-list so that the display of the state can be suppressed during the editing of the sequence graph (see Section 4.5 for detail).
iii) \( \text{st} \): the pointer that points to the block of tables that describes the state: \( p_{\text{in}\_\text{tab}}, l_{\text{i}at\_\text{tab}}, \) and \( p_{\text{out}\_\text{tab}} \), the representations of these three tables are already described early in this chapter.

iv) \( \text{parentp, childp} \): the pointers which point to the list of parent-states pointers and the list of child-states pointers respectively. Each list is divided into a Selected segment and a Deleted segment which are used to distinguish the selected and deleted state transitions respectively. The list is occupied by first the Selected segment and followed by the Deleted segment.

v) \( \text{nparent, ndparent} \): the numbers of pointers in the Selected segment and Deleted segment of the Parent-state-list respectively.

vi) \( \text{nchild, ndchild} \): the numbers of pointers in the Selected segment and Deleted segment of the Child-state-list respectively.

vii) \( \text{next} \): pointer to the next state.

4.2.2 Number of Sequences

This program can also count the total number of sequences (see Figure 4-7 for the flowchart). It is done by first assigning each state in the sequence graph as a node and initializing its content to be zero. Second, assign a value of one to the content of the unassembled state node from the 0th rank and call it the parent node. Third, add all the contents of the parent nodes of the current rank to the contents of their respective child nodes of the next rank (Note that a child node can have more than one parent node). Fourth, if the child node is the final-assembled state, stop and the total number of sequences will be equal to the content of the child node. Otherwise, the child nodes are changed to parent nodes and go back to step 3 through 4 until the final-assembled node is reached.
Figure 4-6. Structure of a state in the state diagram.
From Calling Routine

\[ rnk \leftarrow 1 \]

\[ \text{parent}[0] \leftarrow 1 \]

\[ i \leftarrow 0 \]

\[ i < \text{ncol of rank}[rnk] \]

- YES: \[ \text{child}[i] \leftarrow 0 \]
  \[ i \leftarrow i + 1 \]

- NO: \[ i \leftarrow 0 \]

\[ i < \text{ncol of rank}[rnk] \]

- YES: For each child, \( j \), of parent[\( i \)]:
  - \( \text{child}[j] \leftarrow \text{child}[j] + \text{parent}[i] \)
  - \[ i \leftarrow i + 1 \]

- NO: Change array \( \text{child}[i] \) to \( \text{parent}[i] \)

\[ rnk \leftarrow rnk + 1 \]

\[ rnk < nrank -1 \]

- YES: \[ \text{Return child}[0] \]
- NO: \[ \text{Return child}[0] \]

Figure 4-7. Flow chart of the function "count_seq".
4.3 DISPLAY OF THE ASSEMBLY SEQUENCE GRAPH

In this section, the algorithm of displaying the sequence graph within a prescribed screen boundary is discussed. In addition, the criteria of the color used to represent a state or a state transition are presented. One important feature of this graph-display segment of SED is that when a state or state transition is deleted or recovered from the deletion, the entire process of the deletion or recovery can be seen on the screen at real time (instead of the end of the process). This feature provides a sense of continuity on the screen display during a deletion or recovery process.

4.3.1 Color Selection

There are three basic colors for displaying a state or state transition. The color definition is also consistent in some other windows such as the Delete-List Window where Black is the color for identifying an active deleted-item and Blue is the color for identifying an inactive deleted-item.

The selection of color for displaying a state depends on the state’s status (see Figure 4-8). The color Black is used for ON status; Blue for HILITE status; but no display for OFF status unless the state is displayed inside the View-State Window and in that case, the color Red is used. In addition, There is a special case when the state’s status is ON and the state has a column number of -1. The state is not displayed immediately on the screen because it is still in the deleted-state-list (see Section 4.4.2 for detail).

The selection of the color for displaying a state transition depends on: (i) the status of the two states represented by the state transition, (ii) the column numbers of the two states, (iii) whether the placement of the pointer pointing to the relative child state is in the Selected segment or Deleted segment of the parent state, and (iv) whether the state transition is a recently deleted item (see Figure 4-9). Similar to the color definition for the display of a state, the color Black denotes the presence of the state
transition in the graph and the color Blue denotes the absence of the state transition in the graph but visible on the Sequence Graph Window (zoom-in or zoom-out) through this highlighted color. State transitions that can not be displayed inside the Sequence Graph Window are displayed in the color Red inside the View-State Window.

![Flow chart of selecting the color for displaying a state.](image)

**Figure 4-8.** Flow chart of selecting the color for displaying a state.
Figure 4-9. Flow chart of selecting the color of displaying a state transition. If the returned value is FALSE and the state transition is displayed in the View-State-Window, the selected color is Red.
4.3.2 The Sequence Graph

The graphical output of the sequences is in a compact diamond form. Each state is unique in each rank and there are usually multiple state transitions to reach the state and to leave the state. This format is very important because the number of sequences are usually in the order of hundreds and in the order of thousands when states with subassemblies are included. The 11 parts and 18 liaisons assembly shown in Figure 3-5 has about 42,000 sequences when subassemblies are included.

Figure 4-10 is the flow chart of the sequences-display subroutine. This subroutine can display any portion of the graph given the starting-rank and end-rank. All the states in the selected-state-list of each rank are displayed and centered with respect to the unassembled state of the 0th rank. Due to the limitation of the screen size, the program can only show a small but clear portion of the big graph. To print the complete graph, the program divides up the graph into segments which can be fit into one screen size. Each segment is then shown using horizontal and vertical offset (Figure 4-11) so that picture outside the screen will not be displayed and the only desired segment can be viewed.
Figure 4-10. Flow chart of the graphics display module which draws states and transitions from a given starting-rank to an end-rank.
4.3.3 View A Selected State or State Transition

Sometimes state transitions between states of two ranks can be so numerous that overlapping may be inevitable. This may create some confusion of what the two states that a state transition connected to are. To solve this problem, SED allows a user to select a state from the Sequence Graph Window (Zoom-In or Zoom-Out) and place the state inside the View-State Window. The state is then displayed along with all its parent and child states alone (see Figure 4-12). This feature allows a close examination of just the selected state with its parent and child states. If the user has selected the "switch wdw" option from the View Menu, the Zoom-Out Sequence Graph Window will turn into
a View-State-Text Window through which a display of a text description of the selected state from the View-State Window is shown. The text description consists of the list of all the established liaisons and all the corresponding part names for each liaison. During the display of the part names, if a part name has not been displayed before, it is displayed in color Black, and Blue if it has already been displayed. This feature allows the group of all mated parts' names to stand out from the rest of all the parts' names. In addition, a user can also click and drag a state transition from the View-State Window and place it inside the View-State Text Window to view the text description of the selected state transition.

![Flow chart of displaying the View-State Window.](image)

**Figure 4-12. Flow chart of displaying the View-State Window.**

71
4.4 EDIT THE SEQUENCE GRAPH

One important feature of SED is to allow a user to edit the sequence graph by removing a state or state transition from the graph. In this section, some of the criteria of this state or state transition removal are discussed. The scheme of managing the deleted states and state transitions, and the algorithms of moving states between the selected-state-list and deleted-state-list are also discussed.

4.4.1 Criteria for Deleting States or State Transitions

When a state with more than one subassembly is permitted for an assembly in the generation of assembly sequences, the total number of all possible sequences can be very large (in the order of thousands). However, if an operator can identify an undesirable: (i) assembly state, (ii) particular state transition between two states, or (iii) pattern of state transition, many sequences can be eliminated by deleting the states involved. Furthermore, if the eliminated state is the only child of its parent, then the parent is also eliminated. Similarly, if the eliminated state is the only parent of its child or children, these children are also eliminated. These consequences are pursued all up and down the state diagram.

An additional constraint can be the preclusion of all or some subassembly states. This is proven to be very significant in reducing the resulting number of sequences. For the assembly from Figure 3-5, the number of sequences is reduced from about 42,000 to 440 when states which have more than one subassembly are eliminated. Details of selecting and applying criteria for screening the sequences were discussed by Klein[1986], and De Fazio and Whitney [1986].

Another possible method is to apply the assembly time, and assembly costs of the state and state transition analysis to the sequence structure. By treating the
sequence graph as a directed graph, this quantitative approach can select the most ideal sequence using the shortest path method.

4.4.2 Managing Deleted Items

During the editing process, SED keeps track of all the deleted items (states or state transitions) and displays the information of the deleted items through the Delete-List Window. Figure 4-13 is the schematic diagram of the Delete-List Window managing module which receives command from the Screen Monitoring module. There are two types of command — DELETE and UNDELETE. Every deletion made to the sequence graph is recorded so that a user is allowed to recover not only the most recently deleted item but any one of the deleted items. In this section, the approach to this delete-and-undelete managing scheme is discussed.

![Diagram](image)

Figure 4-13. Schematic diagram of the Delete-List Window Managing Module.
After an item has been received from the screen-control module which monitors the interactions among windows, the delete-list-management module examines the type of the received item and takes an appropriate action accordingly.

The deletion record is an array of a structure, `DEL`, of the following format:

1) Status: ON — if the item is activated (actively deleted); HILITE — if the item is inactivated and can be deleted again.

2) Type: STT — if the item is a state; STST — a state transition between two specific states; TRAN — a pattern of state transition in the graph.

3) Stp: a pointer pointing to the state for type STT and to the parent state if type is STST. Not applicable for type TRAN.

4) Rnk: the rank number of a state for type STT and the rank number of a parent state for type STST. Not applicable for type TRAN.

5) Table: a table that contains the liaison table for type STT and state transition for both types STST and TRAN.

Depending on the type of the to-be-deleted item, the computer first checks if the item already exists in the delete-list and if a match is found and (i) if the item's status is HILITE, the item is deleted again, or (ii) if the item's status is ON, no action will be taken. Otherwise, the item is new and it is appended to the delete-list. Second, the computer branches to the corresponding deleting function according to the type of the item.

Undeleting an item, however, requires some careful planning. The removal of a state or state transition may lead to the removal of other states due to a chain of single dependence relationships. Recording that chain removal of states for undeleting purposes is undesirable because of two major reasons. First, the requirement for storing the chain can be as large as the sequence graph itself. Second and more
Important, since it is required to recover any one of the deleted items in any order and it is possible that two or more items have consequential removal of states and/or state transitions that are not independent, the chain of the removed states may be recorded in a condition different from the condition when the deleted item is recovered. Thus, recovering the chain of removed states alone may not be adequate and not even necessarily correct. For example, a segment of the state diagram shown in Figure 4-14 has two chains of consequential removal of states due to the deletion of states 4 and 5. These two chains are not necessarily the two chains of recovered states when state 4 is first undeleted and, later, state 5 is undeleted. The second example shown in Figure 4-15 also has two chains of states removed due to the deletion of states 4 and 7. However, when state 4 is undeleted, neither state 4 nor state 1 can be recovered and placed in the state diagram due to the fact that state 7 remains deleted. If state 7 is undeleted later on, a chain of states different from the chain of states that is recorded when state 7 is deleted is recovered.

In order to recover all the states or state transitions correctly, all the undeleting processes assume the full recovery of all the states or state transitions associated with the type of the undeleted item. A detailed description of the undeleting processes is presented in Section 4.5.2. Yet the consequence of the recovery of a deleted item may lead to the recovery of some states or state transitions that are still chosen to be deleted. Therefore after undeleting a deleted item, all the activated items in the deleted-list are executed again to ensure the set of deleted items remains deleted. Furthermore, the deletion subroutines are intelligent enough that if a to-be-deleted item is already deleted from the graph, the deletion process will be skipped immediately.
Figure 4-14. Undeleting a state from the state diagram (case 1).
Figure 4.15. Undeleting a state from the state diagram (case 2).
As discussed in Section 4.3, during the deletion of an item, all the deleted states and state transitions are highlighted in Blue in real time. Similarly, during the process of undeleting a deleted item, all the recovered states and state transitions are displayed in Black in real time. However, it is possible that some of the recovered states are not currently displayed on the screen because they are linked in the deleted-state-list and not the selected-state-list. Therefore at the end of every undelete process, the computer checks if there is any state in the selected-state-list which has its status ON in each rank. If such a state is found, a subroutine called "move_off_to_on" will move all the ON status states from the selected-state-list to the selected-state-list and all the sequence graph windows will be refreshed to accommodate the recovered states.

When a state is deleted, it is still linked in the select-state-list of the respective rank with the state's status changed to HILITE. Yet unlike a state which has a variable status in its structure to designate itself to be highlighted, the state structure SELECT cannot designate a highlighted state transition (the present structure only has Selected segment and Deleted segment inside both the Parent-state-list and Child-state-list). Therefore, the only way to highlight the deletion of a state transition is to append it to a list called Tranlist which keeps track of all the highlighted state transitions due to the removal of state transitions alone.

If a user decides to refresh the screen such that all the highlighted (deleted) items are removed from the Sequence-Graph Window, the user can click on the RD (ReDraw) icon to accomplish it. This process is done by a subroutine called "move_hilite_to_off" which moves all the HILITE status states from the selected-state-list to the selected-state-list.

The two subroutines "move off to on" and "move hilite to off" are described in the following.
**Move_off_to_on**

This subroutine recovers a deleted item which is not currently displayed on the Sequence-Graph Windows and the approach is as follows: the delete-state-list is searched from the beginning and if a state's status is found to be ON, it is cut from the delete-state-list and pasted at the end of the select-state-list. Therefore, the recovered item is always displayed at the far right of the rank and not at the position where it used to be. These searching, cutting, and pasting processes are continued until the end of the delete-state-list is reached.

**Move_hilite_to_off**

The goal of this subroutine is to have all the highlighted states or state transitions removed from the Sequence-Graph Window screen. Starting from the beginning of the selected-state-list, every state whose status is HILITE is changed to OFF with a column number -1. The new OFF state is then appended to the end of the selected-state-list of the respective rank. After each removal of the state from the selected-state-list, the list is relinked and the column numbers updated accordingly. Furthermore, the Tranlist which records all the highlighted state transitions deleted by the removal of state transitions alone is also cleared.

### 4.5 DELETE AND APPEND (UNDELETE)

In this section, the algorithms of deleting and recovering a state or state transition are discussed.

There are two fundamental subroutines "edparent" and "edchild" for editing a state and state transition. Both subroutines "edparent" and "edchild" take three parameters: i) the process mode (either in DELETE or APPEND), ii) the rank of the processing state, and iii) the pointer to the processing state. At the end of either an item
deletion or recovery, the number of sequences count is immediately updated. Both subroutines are similar to the fact that most data processing are done to the Selected segment and Deleted segment of the Child-state-list for "edparent" and Parent-state-list for "edchild". Both lists have the same similar structures — the Selected segment is followed by the Deleted segment. Figure 4-16 illustrates the movement of a pointer from the Selected segment to the Deleted segment (Figure 4-16-a), and from the Deleted segment to the Selected segment (Figure 4-16-b).

(a) From selected segment to deleted segment.

(b) From deleted segment to selected segment

Figure 4-16. Illustrations of pointers movement between the selected segment and deleted segment. Both parent-state-list and child-state-list have the same segment-structure. (a) From the selected segment to deleted segment, and (b) from the deleted segment to selected segment.
Edparent

In DELETE mode (flow chart shown in Figure 4-17), for each parent state of the processing state, the pointer to the processing state is moved from the parent state's Selected segment to the Deleted segment and the processing state's status is changed to HILITE but its column number is kept unchanged. If after moving the pointer, there is no pointer in the Selected segment left, the parent state becomes a processing state and it is passed recursively to the subroutine "edparent" in DELETE mode.

In APPEND mode (flow chart shown in Figure 4-18), for each parent state of the processing state, the pointer to the processing state is moved from the parent state's Deleted segment to the Selected segment and the processing state's status is changed to ON. If after moving the pointer, the pointer is the first one in the Selected segment, the parent state becomes a processing state and is passed to the subroutine "edparent" in APPEND mode recursively.

Edchild

This subroutine is similar to the subroutine "edparent" both in the DELETE and APPEND modes. All the operations of the subroutine "edchild", however, are done to each child state, instead of each parent state, of the processing state.

4.5.1 A state

To delete a state, pass the state as the processing state to the subroutines "edparent" and "edchild" to eliminate all single dependent states up and down the state diagram. Similarly, to append a state back to the graph, pass the state as the processing state to the subroutines "edparent" and "edchild" to append all, if any, single dependent states.
Figure 4-17. Flow chart of the subroutine "edparent" in DELETE mode.
Figure 4-18. Flow chart of the subroutine "edparent" in APPEND mode.
4.5.2 A state transition between two specific states

To delete a state transition between a parent state PST and a relative child state CST, first move the CST pointer from the PST's Child-state-list Selected segment to the Deleted segment. If after moving the pointer, there is no pointer in the PST's Selected segment left, PST becomes a processing state and it is passed to the subroutine "edparent" in DELETE mode. Second, move the PST pointer from the CST's Parent-state-list Selected segment to the Deleted segment. If after moving the pointer, there is no pointer in the CST's Selected segment left, PST becomes a processing state and it is passed to the subroutine "edchild" in DELETE mode. As a result, all single dependent states, if any, in the graph are eliminated.

To append the state transition, the CST pointer is moved from the PST's Child-state-list Deleted segment to the Selected segment and if CST is the first pointer in the Selected segment, PST is passed to the subroutine "edparent" in APPEND mode. Similarly, if the PST pointer is moved from the CST's Parent-state-list Deleted segment to the Selected segment and if PST is the first pointer in theSelected segment, CST is passed to the subroutine "edchild" in APPEND mode. Consequently, all single dependent states, if any, in the graph are appended.

4.6 Chapter Summary

In this chapter, the program structure of SED is presented. The design of the screen and the utilization of windows for different purposes are described. The approach to processing information and displaying it through various windows is discussed. In particular, the interactions among windows during the editing of the sequence graph are discussed. Using a mouse as an input device, a user can select a state or state transition from the sequence graph and examine its context, or remove it from the sequence graph. If the deletion of a state or state transition can lead to a chain of
deletions of other states or state transitions in the state diagram, the chain removal will be done automatically and displayed on the screen in real time. Another important feature of SED is the ability to allow the user to undelete not only the most recently deleted item but also any previously deleted item by using the computer to manage and control all the deleted items. The algorithms of the processes of delete and undelete are discussed in detail.
CHAPTER 5

RECOMMENDATIONS FOR FUTURE WORK

In this chapter, future work is discussed and potential solutions, if any, are suggested. The work can be divided into three different areas: 1) Providing more features for SED, 2) Interfacing future software with other analytical programs, and 3) Automating the generation of precedence relations. The goals of this future work are to make Liaison Sequence Analysis more automated, complete, easier to use, and applicable to other analyses that rely on selected assembly sequences.

5.1 PROVIDING MORE FEATURES FOR SED

The following topics are some of the features that should be considered to enhance the current software:

Consolidating LSG and SED

The current software system requires the user to write a subroutine containing the precedence relations, and performs a subsequent manual process of compiling and linking the subroutine. The user needs understand the 'C' language syntax and the operating system of the computer. The goals of the two proposed methods presented here are to integrate the two programs, LSG and SED into a single program, and to waive the requirement of the user's knowledge of the language syntax and the computer's operating system. Both methods require the ability to edit text within the program SED so that a user can write the precedence relations in the general form of (...) \rightarrow (...) shown in Figure 3-2.
The first method requires the modification of SED so that it can automate the current manual operations (see Figure 5-1). First, the written form is translated into a C subroutine code by SED. Second, by issuing commands from SED to the computer operating system, the translated subroutine code is compiled and linked with the existing LSG object code to form an executable LSG program. Third, once the executable LSG program is linked successfully, SED sends a command to run LSG to generate the sequence data. The only disadvantage of this method is that it requires a C compiler in the computer system along with the program SED in order to produce an executable LSG program. Moreover, the processes of translating, compiling, and linking must be repeated whenever a new set of precedence relations is used to generate the new sequence data. This may pose a problem if the computer has a limited run-time memory or file storage space.

Figure 5-1. Consolidating LSG and SED through the computer operating system.
The second method can resolve the problems of the previous method. It employs a precedence-relations interpreter, which makes the installation of a C compiler unnecessary. The logical expressions of the precedence relations are no longer in a machine code. Using the second method, the precedence-relation expressions are converted, stored, and evaluated by the interpreter (see Figure 5-2). Even though the speed of evaluating the expressions by an interpreter is slower than the one of executing the expressions in a machine code form, the author believes that the speed may not be a disadvantage when it is compared with the dependence on a C compiler and the intermediate process of compiling and linking required by the first method. In addition, a precedence-relation interpreter gives the computer the flexibility to perform other analyses other than sequence generation. For example, a possible feature of SED in the future is to generate a truth-table from one or more precedence relations similar to the one shown in Table 3-2. This feature allows the user to visualize the sets of permitted and/or forbidden assembly states for a given set of precedence relations. It may be a valuable feature during the editing of the precedence relations when a change in any precedence relation can cause many changes in the corresponding truth-table. The most efficient means of reflecting the changes to the user promptly is through a precedence-relation interpreter. In order to facilitate the evaluation of the precedence relations, the following method of preprocessing the precedence relations into expressions ready to be interpreted is suggested.

From Figure 3-2, the expression on the left hand side of the precedence relation that precedes liaison 16 has the form:

\[(17 \text{ OR } (4 \text{ AND } (3 \text{ OR } 14)))\]  (6)
Figure 5-2. Incorporating a precedence-relation interpreter into the future software system.

It is an expression that has a Boolean form and is written in an infix notation. The evaluation of (6) proceeds from left to right. By using the logic convention, the binary operator AND has a higher precedence than the binary operator OR but the hierarchy of operations can be overruled by means of parentheses. The expression (6) can be represented by a tree shown in Figure 5-3. The label of each node represents the operation, and the arrows originating from a node lead to the operands. The infix form of expression (6) can be converted to a postfix form, commonly called reverse Polish notation (RPN)* [Berry, 1982]. The expression of (6) in RPN is:

* Polish notation was developed by the Polish logician Jan Lukasiewicz.
The postorder representation of (7) can be illustrated in Figure 5-4. RPN is a parenthesis-free notation which utilizes stack manipulation during the evaluation process. The advantage of converting the expressions of precedence relations into RPN is that in scanning a RPN expression from left to right, all the operands are encountered before their operator. Thus, when the interpreter encounter the operator, the fixed number of operands associated with the operator can be retrieved from the stack and executed immediately [Korfhage, 1984].

Figure 5-3. Tree representation of a Boolean Expression.

Figure 5-4. Postorder representation of the expression depicted in Figure 5-1.
Saving and retrieving the delete list

The usage of the delete list can extend beyond the primary purpose of deleting and undeleting states and state transitions during the editing of the sequence graph. First, instead of saving the data of the entire sequence graph after a sequence graph editing, the delete list, which usually occupies less storage space, can be saved as a data file along with the original sequence data. Multiple versions of the edited sequence graph can then be saved or retrieved efficiently. Second, if a new set of sequence data is produced due to changes in the precedence relations of the assembly, the user can retrieve a previous delete-list file and apply it to the new sequence graph. This feature also enables a user to eliminate awkward or difficult states or state transitions of the assembly of a redesigned product given the delete-list file of the original product.

Ordering problem

In the display of the sequence graph, the positions of the states sometimes give a large number of crossings among the state transitions. This makes it difficult for a user to identify the two corresponding states that a state transition originates from and leads to. To minimize the number of crossings for the entire graph, a feature is needed to rearrange the states so that the number of crossings can be minimized. Algorithms of this rearrangement process are widely used in the design of the layout of electrical components on a circuit board. May [1985] has suggested one of these algorithms.

Assembly parts in graphical forms

It will be useful to have a feature that allows a user to define every part of an assembly in graphical form so that the graphical objects can be used to display an assembly state or to animate the assembly step of a state transition. A user can then better comprehend the symbols (boxes and lines) of the sequence graph. In addition, during the interactive editing of the objects, the user can also create the liaison diagram
of the assembly within the same program. The information associated with the parts in the assembly may include detail parts' dimensions, tolerances, clearances, and mating conditions among parts. A more detailed discussion on this data structure is presented in Section 5.3.

**Dialog box**

A dialog box enables a user to send commands to the computer by typing on a keyboard and allows a user to read the computer's output in text form on the computer screen.

**Modification of the structure SELECT**

The current process of determining the color for displaying a state transition is inefficient (see Section 4.3.1). It can be improved by modifying the state structure SELECT. Two new segments are added to both the parent-state-list and child-state-list. The first segment, highlighted-selected-segment, represents a list of the recently undeleted state-transitions and the second one, highlighted-deleted-segment represents a list of the recently deleted state-transitions. This new state structure makes it possible to have an efficient process of highlighting the recently deleted and undeleted state transitions.

**Identifying a state with subassemblies**

Eliminating states that have subassemblies can significantly reduce the number of states in the state diagram. To identify a state with subassemblies, first treat the state as a graph and all mated parts and liaisons as nodes and edges respectively. Second, pick any node from the graph and visit all the nodes that are linked to the first node. Third, check if the number of visited nodes is equal to the number of nodes in the graph. The state has subassemblies if the two numbers are not the same. This
identification feature can be used for the interactive process of searching for and deleting states with subassemblies.

**Searching for a state or state transition**

This feature allows a user to locate a state or state transition in a sequence graph which is usually huge in size. Since a state transition may have multiple occurrences in the graph, the searching process may have to continue for the rest of the graph.

**Deleting a pattern of a state transition**

This is the feature TRAN discussed in Section 4.4.2. It is accomplished by searching for every occurrence of the state transition to be deleted, and deleting it by passing it to the present subroutine which deletes a state transition between two specific states.

**Constraining assembly moves**

This is a feature that enables a user to preserve (or eliminate) a specified set of state transitions originated from a specified state so that the rest of the state transitions can be eliminated (or preserved). This feature allows a convenient grouping of preserved (or eliminated) state transitions originated from the same state.

**Passing through a set of specified states**

This is a feature that enables a user to constrain assembly sequences by passing through some specified states for purposes such as testing, measurement and so on [De Fazio and Whitney, 1986]. The process of this feature can be executed by first marking all the ascendant and descendant states of the specified states to be passed in the entire state diagram. Next, all the states that are not marked in the state diagram are deleted.
5.2 INTERFACING FUTURE SOFTWARE WITH OTHER ANALYTICAL PROGRAMS

After the number of sequences is reduced by editing the sequence graph, other assembly analyses can be used to aid a production engineer to select the most effective assembly sequence. One of the available economic analyses is Assembly System Design Program (ASDP) which can be used to evaluate the overall performance of each candidate sequence for a given set of resources such as manual labor, fixed and programmable automation [Gustavson, 1986]. To interface with programs such as ASDP, future software will need to communicate with them in the form of assembly data.

Another possible approach is to use an expert system to determine the cost and time spent of each assembly state transition, and the penalty cost (e.g. an unstable assembly state that requires an extra jig), if any, of each state. The evaluated data can then be stored in a computer data base and utilized in other analyses. For example, the state diagram can be treated as a network with the unassembled state and final assembled state as the source and sink respectively. Costs can be assigned to each node (state) and arc (state transition) in the network. The shortest path (sequence with the minimum cost) can be found by using the technique developed by Dijkstra [Korfhage, 1984].

5.3 AUTOMATING THE GENERATION OF THE PRECEDENCE RELATIONS

Automating the generation of the precedence relations remains the most challenging part of the future research. Although it may be feasible for a production engineer to answer the type of questions suggested by De Fazio and Whitney [1986], the qualitative form of logical combinations of the liaisons as immediate answers poses a major obstacle to the answering of the questions by a computer. On the contrary, the
form of Bourjault's questions are more easily answered by a computer since the answer are always either 'yes' or 'no' and the answers are solely determined by the geometric constraints of the assembly [De Fazio, 1988]. Thus the first decision one has to make in this research is to determine which computer-modelling method should be used for describing the product design. The second step is to construct a data structure for describing the assembly so that information other than the geometric data of the parts can be included. The next problem is to find an approach to automate the answering of Bourjault's questions based on the data structures of the product's parts. In this section, some of the potential solutions is presented.

Marks [1987] observed that most of the next generation CAD systems will be based on solid modeling which defines parts by Boolean operations of primitive objects such as cylinders, rectangular boxes and so on. The Part and Assembly Description Language-2 (PADL-2 [Brown, 1982]) is an example of software that is based on the technique of constructive solid geometry (CSG). Some of the advantages of solid modeling are that the representation of a part is unambiguous when compared to wire frame representation, and that machining verification and parts interference testing can be done more easily than other modeling method [Requicha and Voelcker, 1983].

Wesley et al. [1980] described a geometric modeling system, based on an assembly directed programming system, AUTOPASS (AUTOmated Parts ASsembly System [Lieberman and Wesley, 1977]). Subsequently, Lee and Gossard [1985] developed another variation of an assembly data structure which eliminates the explicit definition of a $4 \times 4$ transformation matrix for each component required by AUTOPASS. The assembly data structure encompasses the mating condition of each component such as 'against' and 'fit'. The information of the mating conditions may be found useful for inferring the hierarchy structure of the possible assembly paths for each component such as translation, rotation, or a combination of both.
Interference detection during assembly moves has two types — static and dynamic [Mortenson, 1985]. Static interference refers to the interference between two objects when they are properly mated and it can be found by examining the union of the two solid objects. Dynamic interference refers to the collision of a part or a subassembly with other mated parts along an assembly path. It is done by static interference checking between the stationary object(s) and the volume swept by the moving object along a particular path. Works by Boyse [1979], Esterling and Rosendale [1983], and Wang and Wong [1986] may provide some solutions to an efficient process of dynamic interference checking. The effects of tolerances, which are discussed by Requicha [1983], can be represented by first modifying the nominal sizes and shapes of objects by using Monte Carlo methods. Afterward, the static and dynamic interference checkings can be applied to the modified objects.

Since the number of paths can range from none to infinitely many, dynamic interference checking is a computation intensive process. There is need for a research in finding the assembly path of an object (part or subassembly) based on the type of the object (e.g. bolt, screw, shaft) and/or its mating condition with other objects. Yet a complete generalization of mating paths based on non-geometric data requires extensive study of many assembly cases. Worse, an incomplete paths tree that tries to represent the hierarchy trials of assembly paths can give a negative answer when there is indeed a possible but intricate path. For example, the mating path may involve some changes of a part's orientation along the assembly path. Even if the paths tree is a complete one, the length of computation process may be staggering. If many tested cases have similar geometric configuration, an artificial intelligence system could help shorten the amount of time required to find a possible path by utilizing past experience. It is analogous to the mouse-in-a-maze problem where finding an exit path is just as
important as remembering past experiences. The path finding problem is further complicated when there are flexible parts in the assembly.

Solid modeling can provide a means to automate the answering of Bourjault's questions. On the other hand, some non-assembly tasks may not be encompassed by the interference checking scheme; therefore, the final form of the generated relations may need some revision so that additional constraints can be appended.
CHAPTER 6

CONCLUSIONS

The successful implementation of the Liaison Sequence Method has been described. The major features of the system are: 1) the automatic generation of all valid assembly sequences, 2) incorporation of precedence relations into the existing software with relative ease, 3) generation of assembly sequences in a short period of time (less than a minute in many cases), 4) no limitation on the size of an assembly in terms of the liaison-count, and 5) support for editing of the generated sequence graph.

Some of the issues encountered in the development of the system were: determination of the format of the precedence-relations subroutine code, identification of an efficient method of finding a set of last liaisons during simultaneous liaisons mating, generation of the assembly states that constitute the liaison sequence graph, and specification of the features for the sequence graph editor.

The work from this research helps an engineer to examine all valid assembly sequences and to reduce the large number of sequences to a size useful for further analysis. As a result, an engineer can find a good assembly sequence with the confidence that no other sequence has skipped their attention.
APPENDIX A:

LIAISON DIAGRAM INPUT DATA

The following is the format of sample input data for LSG. The data describe the liaison diagram of an assembly in Figure 1-1. In the first block, a liaison number is followed by two part names (up to 8 non-space characters). Liaison 0 designates the end of entries. The next block is a detailed description of each part name. Space can be used but the name must be terminated with a semicolon.

```
1   A   B
2   A   C
3   A   E
4   A   F
5   B   C
6   B   F
7   C   E
8   C   D
9   D   G
10  E   F
11  F   G
12  E   H
13  D   H
14  H   J
15  D   H
16  H   K
17  J   K
18  G   H
0    0    0

A   Belt1;
B   Pulley;
C   Impeller;
D   Rear Case;
E   Input Shaft;
F   Gear 1;
G   Gear 2;
H   Belt 2;
J   Front Case;
```
APPENDIX B:

PRECEDENCE RELATIONS SUBROUTINE CODE

The following code is taken from Figure 3-2 using the approach shown in Figure 3-3. The second example code is taken from another set of precedence relations using the approach shown in Figure 3-4.

```c
/*
 * Program EXAMPLE1.C updated on 2/18/88
 */

#define TRUE 1
#define FALSE 0

/*
 * Return TRUE or FALSE based on the restricted format of precedence rules.
 *
 * Assume that the tested liaison i is established. See if the generated state
 * satisfies the precedence rules, and a TRUE or FALSE answer will be returned
 * accordingly.
 */

int rules(li, i)
int .., li[];
{
    switch (i) { /* Branch to the PR that contains i on the RHS */
    case 1: /* liaison 1 is always evaluated true, now evaulate the LHS */
    case 2:
        return (li[1]);
    case 5:
    case 7:
        return (li[2]);
    case 8:
        return (li[1] || li[3]);
    case 9:
    case 10:
    case 11:
        return (li[1]);
    case 12:
        return (li[2]);
    case 13:
```
return (li[7]);
case 16:
case 17:
    return (li[15] || li[18]);
default:
    return(TRUE);
}

/***************************************************************************/
/*
/* Program EXAMPLE2.C updated on 2/23/88 */
/*
/***************************************************************************/

#include "common.h"
#include "extern.h"

define TRUE 1
#define FALSE 0
#define ON 1
#define OFF 0
#define N(array) (sizeof(array) / sizeof(int))

typedef struct lpr_info {
    int *array;
    int n;
} LIPR;

int li[] = { 8};  /* liaison 1 is on the RHS of the 8th precedence relation */
int li2[] = { 1, 5, 8};  /* first, fifth, and eighth PR for liaison 2 */
int li3[] = { 0};  /* liaison 3 is not found in the RHS of any PR */
int li4[] = { 2, 8};
int li5[] = { 0};
int li6[] = { 3, 8};
int li7[] = { 4};
int li8[] = { 5, 6, 7, 13};
int li9[] = { 6, 7};
int li10[] = { 0};
int li11[] = { 0};
int li12[] = { 9};
int li13[] = { 7, 10};
int li14[] = { 11};
int li15[] = { 12, 13, 14, 15, 16, 17};
int li16[] = { 0};
int li17[] = { 0};
int li18[] = { 0};

/* assign the number of PR needs to evaluate for every liaison */
LIPR lpr[] = {N(li1), N(li2), N(li3), N(li4), N(li5), N(li6), N(li7), N(li8), N(li9), N(li10), N(li11), N(li12), N(li13), N(li14), N(li15), N(li16), N(li17), N(li18)};

/*****************************************************************************
* Assume that the tested liaison i is established. See if the generated state
* satisfies the precedence rules. If the RHS and LHS of any PR is evaluated
* to be TRUE and FALSE, respectively, PR are not satisfied.
*****************************************************************************

104
int rules(li, i)
int li[], i;
{
    int j, flag = OFF;
    int part_table[MAXPARTS];

    for (j = part_max; j; j--) /* percussion of subassembly state */
        part_table[j] = OFF;
    for (j = liai_max; j; j--)
        if (li[j]) {
            part_table[ liai_tab[j].part_x ] = ON;
            part_table[ liai_tab[j].part_y ] = ON;
            flag = ON;
        }
        if (flag == ON && part_table[ liai_tab[i].part_x ] == OFF
            && part_table[ liai_tab[i].part_y ] == OFF)
            return (FALSE);
    li[i] = ON;

    for (j = 0; j < lipr[i].n; j++) /* check every n PR */
        if (check_pr(li, lipr[i].array[j])) {
            li[i] = OFF;
            return (FALSE);
        }
    li[i] = OFF;
    return (TRUE);
}

/*************************************************************************
/* check_pr() /*
/* Branch to the requested PR to check for PR validity. */
**************************************************************************/

int check_pr(l, pr)
int l[], pr;
{
    switch (pr) { /* branch to the numbered precedence relation */
    case 0:
        return (TRUE);
    case 1: /* the first precedence relation */
        if (l[2]) /* RHS */
            return (l[1] || l[5]); /* LHS */
        else return (TRUE);
    case 2: /* the second precedence relation */
        if (l[4])
            return (l[3] || l[10]);
        else return (TRUE);
    case 3:
        if (l[6])
            return ((l[1] && l[3]) || (l[1] && l[10])
        else return (TRUE);
    case 4:
        if (l[7])
        else return (TRUE);
    case 5:
        if (l[2] && l[8])
            return (l[5] || l[1]);
        else return (TRUE);
case 6:
  if (l[8] & l[9])
else return (TRUE);

case 7:
else return (TRUE);

case 8:
  return (l[8] || l[9]);
else return (TRUE);

case 9:
  if (l[12])
else return (TRUE);

case 10:
  if (l[13])
  return (l[9] || l[10]);
else return (TRUE);

case 11:
  if (l[14])
  return (l[16] || l[17]);
else return (TRUE);

case 12:
  if (l[15])
           || (l[14] & l[18])));
else return (TRUE);

case 13:
  if (l[8] & l[15])
else return (TRUE);

case 14:
  if (l[15])
else return (TRUE);

case 15:
  if (l[15])
  return (((l[13] & l[16]) || l[17]));
else return (TRUE);

case 16:
  if (l[15])
  return (l[16]);
else return (TRUE);

case 17:
  if (l[15])
else return (TRUE);
APPENDIX C:

PARTIAL LISTING OF SOFTWARE CODE

The following code is used for generating the assembly sequences data (Files LCOMMON.H, LEXTERN.H, MOD2.C. and MOD3.C) and editing the sequence graph (Files SCOMMON.H, SEXTERN.H, EDIT.C, and UTILITY.C). Any request for the complete software code should be sent to The Charles Stark Draper Laboratory, Inc.

```c
/* ***********************************************************************/
/* File LCOMMON.H updated on 2/22/88 */
/* ***********************************************************************/

#define MAXLAIAS 24        /* maximum liaison # */
#define MAXPARTS 24        /* maximum part # */
#define MAXBACYS 24        /* maximum # of basic cycles */
#define MAXLOOPS 400       /* maximum # of generated loops in */
                       /* a liaison graph */
#define MAXNRCYS 24        /* maximum # of non-redundant cycles */
#define MAXSTATES 1000     /* maximum # of states */
#define MAXCOLUMNS 250     /* maximum # of columns in a rank */
#define MAXSTATRANS 6000   /* maximum # of state transitions */

#define NRCYSIZE (MAXNRCYS+15)/16   /* size for short */
#define LSIZE (MAXLAIAS+15)/16      /* int array which is */
#define PSIZE (MAXPARTS+15)/16      /* in bit field form */

#define ON 1
#define OFF 0
#define TRUE 1
#define FALSE 0

typedef struct lliai_info {  /* check if liaison # exists */
    int exist;
    int part_x;       /* part_x # */
    int part_y;       /* part_y # */
} LIAI, *LIAIPTR;

typedef struct part_info {  /* check if part # exists */
    int exist;
    int list_count;     /* # of entries in the list */
    int part_list[MAXPARTS];  /* a list of connected parts */
    int lliai_list[MAXLAIAS]; /* a list of connected liaisons */
```
typedef struct liainnrcy_info {
    int nrcyps;  /* start pointer to a list of non-redundant */
    /* cycles that include the liaison */
    int nrcype;  /* end pointer to the last element of */
    /* a nr-cycle list */
} LIAINRCY, *LIAINRCYTPTR;

typedef struct nrcylist_info {
    int nlist;  /* # of counts in a list */
    unsigned short int list[LSIZE];  /* bit-field format array */
} NRCYLIST, *NRCYLISTPTR;

typedef struct nrcy_info {
    int count;  /* non-redundant cycles count */
    int size;  /* used size of bit-field array */
    LIAINRCY licy[MAXLIAIS];  /* table lookup of liaison to */
    /* respective non-redundant cycles */
    NRCYLIST list[MAXNRCS];  /* non-redundant cycles' liaisons list */
    int li_tab[MAXNRCS*MAXLIAIS];  /* table where licy[] points to */
} NRCY, *NRCYTPTR;

/***************************************************************************/
/*                                                                       */
/*  File LEXERN.H updated on 10/22/87                                  */
/*                                                                       */
/***************************************************************************/

extern LIAI lial_tab[MAXLIAIS+1];
extern PART part_tab[MAXPARTS+1];
extern int papa_tab[(MAXPARTS+1)*MAXPARTS/2+1];
extern int lial_count, part_count;
/* # of liaisons and parts in the assembly */
extern int lial_max, part_max;
/* respective highest index of liai and part */
extern int lial_size, part_size;
/* used size of short int array */

/***************************************************************************/
/*                                                                       */
/*  File MOD2.C    updated on 2/26/88                                  */
/*                                                                       */
/***************************************************************************/

#include <stdio.h>
#include "lcommon.h"
#include "lexern.h"

typedef struct tree_node_info {
    int list[MAXPARTS];  /* list of node # */
    int lp;  /* pointer to the list */
} TNODE, *TNODEPTR;

typedef struct side_info {
    TNODEPTR rt;  /* pointer to list of current level roots */
    TNODEPTR lv;  /* pointer to list of leaves from */
}
typedef struct mark_info {
    int path[MAXPARTS]; /* node # that the node directed from */
    int meet[2]; /* the node that the two trees meet at */
    /* the other node which leads to the meet- */
    /* node beside the node stored in path[] */
} MARK, *MARKPTR;

typedef struct loop_info {
    int lcount; /* contains # of links along the loop */
    unsigned short int liai_tab[LSIZE]; /* table of liaison that constituted the loop */
} LOOP, *LOOPPTR;

/**************************************************************************/
/* Make non-redundant cycles. */
/**************************************************************************/

int mk_nrcycles(ncp)
NRCYTPTR ncp; /* structure non-redundant cycle */
{
    int bacy_count; /* basic cycles count */
    int comb_count; /* combinations of all possible cycles */
    int sptree[MAXLIAIS+1]; /* list of links of the spanning tree */
    int node[MAXPARTS+1]; /* table of nodes of the tree */
    int mslink[MAXLIAIS]; /* missing-links list */
    int mslink_count; /* # of missing-links */
    unsigned short int bacy[MAXBACYS][LSIZE]; /* table of links for basic cycles */
    LOOP loop[MAXLOOPS]; /* all possible generated loops */

    /* Create a spanning tree from the liaison graph */
    mslink_count = mk_spantree(sptree, mslink);

    /* Make basic cycles by adding missing link one a time */
    bacy_count = mk_bacycles(sptree, mslink, mslink_count, bacy);

    /* Generate all cycles from all possible combinations of the basic cycles */
    comb_count = mk_all_cycles(bacy, bacy_count, loop);

    /* Eliminate any redundant cycle from all the generated cycles */
    screen_cycles(comb_count, loop, ncp);
}

/**************************************************************************/
/* Make a spanning tree by adding links, of increasing numerical order, */
/* to an empty tree. The addition of a link to a graph does not */
/* yield a loop when: */
/* 1) the link of degree one is attached to the end of a tree, or */
/* 2) it is not attached to any other link, or */
/* 3) using function, loop_test, as the last resort. */

int mk_spantree(sptree, mmlink)
int sptree[]; /* table of links of a spanning tree */
int mmlink[]; /* table of missing links in sptree[] */
{
    int i;
    int link_i; /* edge of the liaison graph */
    int x, y; /* part # */
    int spl_count = 0; /* # of links in the spanning tree */
    int mmlink_count = 0; /* # of missing-links */
    int list[MAXLIAIS]; /* list of links in a loop */
    int nlinks; /* # of links in a loop */
    int node[MAXPARTS]; /* table of nodes of the tree */

    for (i = part_max; i; i--)
        node[i] = OFF;
    for (i = liai_max; i; i--)
        sptree[i] = OFF;

    for (link_i = 1; link_i <= liai_max; link_i++)
        if (liai_tab[link_i].exist) {
            x = liai_tab[link_i].part_x;
            y = liai_tab[link_i].part_y;

            if (! (node[x] && node[y]) ||
                !loop_test(link_i, sptree, list, &nlinks)) {
                node[x] = node[y] = ON;
                sptree[link_i] = ON;
                spl_count++;
            }
            else
                mmlink[mmlink_count++] = link_i;
        }
    return(mmlink_count);
}

/****************************************************************************/
/* Test if the tree has a loop in it. */
/****************************************************************************/

int loop_test(link_i, org_sptree, lp_list, plinks)
int link_i; /* link # that may yield a loop */
int org_sptree[]; /* table of links of the spanning tree */
int lp_list[]; /* list of links along the loop */
int *plinks; /* pointer to # of links in loop_list[] */
{
    SIDE left; /* left hand side (L.H.S.) tree */
    SIDE right; /* right hand side (R.H.S.) tree */
    SIDEPTR act; /* pointer to the active tree */
    SIDEPTR inact; /* pointer to the inactive tree */
    SIDEPTR temp; /* temporary variables for */
    TNODEPTR tran; /* swapping two pointers */
    TNODE left_1, left_2; /* lists for L.H.S. tree */
    TNODE right_1, right_2; /* lists for R.H.S. tree */
    MARK mark; /* information on the two */
        /* growing trees' path */


110
int i;
int node_i;
int sptree[MAXLIAIS]; /* copy of org_sptree[] */

left.rt = &left_1; /* assign two lists to each */
left.lv = &left_2; /* side of the tree */
right.rt = &right_1;
right.lv = &right_2;

left.rt->list[0] = liai_tab[link_i].part_x;
right.rt->list[0] = liai_tab[link_i].part_y;
left_1.lp = right_1.lp = 1;

for (i = liai_max; i >= 0; i--) {
    mark.path[i] = OFF;
sptree[i] = org_sptree[i];
}

act = &left;
inact = &right;

/*
* Function search_lv() check if leaves are found from the current
* root level. If it does, the leave nodes are stored in the leave list.
*/

search_lv(sptree, &mark, inact);

while (TRUE) {
    switch(search_lv(sptree, &mark, act)) {
    case -1:
        return(FALSE); /* a dead end is found in a tree, so */
    /* the two trees are not connected */
    case 0:
        if (act->lv->lp > inact->lv->lp) {
            temp = act; /* swap the active and inactive */
            act = inact; /* pointers so that the active tree */
            inact = temp; /* has less leaves to work with */
        }
    tran = act->rt; /* swap the roots and leaves list */
    act->rt = act->lv; /* pointer to generate leaves for */
    act->lv = tran; /* the next level */
    break;
    case 1:
        lp_list[0] = link_i;
        lp_list[1] = papa_tab[papa_index(mark.meet[0], mark.meet[1])];
        *pnlinks = 2; /* the list of links are not in */
        /* serial order */
        for (i = 1; i >= 0; i--) {
            node_i = mark.meet[i];
            while (mark.path[node_i]) {
                lp_list[(*pnlinks)++] =
                    papa_tab[papa_index(node_i, mark.path[node_i])];
                node_i = mark.path[node_i];
            }
        }
        return(TRUE);
    }
}

/*******************************************************************************/
/*
* search_lv()
*/
/*****************************/

111
int search_lv(sptree, pmark, pside)
int sptree[]; /* table of links of the original tree */
MARKPTR pmark; /* nodes pointer */
SIDEPTR pside; /* pointer to the working tree */
{
  int i;
  int ptr; /* pointer to the connected node list */
  int root; /* root of the current level */
  int leaf; /* leaf of the next level */
  PARTPTR p; /* pointer to the part_tab */

  pside->lv->lp = 0;

  for (i = pside->rt->lp; i; i++) {
    root = pside->rt->list[--i];
    p = &part_tab[root];
    for (ptr = p->list_count; ptr;)
      if (sptree[p->lai_list[--ptr]]) {
        sptree[p->lai_list[ptr]] = OFF;
        leaf = p->part_list[ptr];

        if (pmark->path[leaf]) {
          pmark->meet[0] = leaf;
          pmark->meet[1] = root;
          return(1); /* a connected node with the other */
          /* other side of the tree is found */
        } else {
          pmark->path[leaf] = root;
          /* every entry in mark->path[] contains */
          /* the node # it comes from */
          pside->lv->list[pside->lv->lp++] = leaf;
        }
      }
  }

  if (pside->lv->lp)
    return(0); /* some roots are found */
  else return(-1); /* no root is found */
}

void mk_bacycles()
/* Make basic cycles from adding missing links to the spanning tree */
/* one at a time. */

int mk_bacycles(sptree, mslink, mslink_count, bacy)
int sptree[]; /* list of links of the spanning tree */
int mslink[]; /* table of missing links in sptree[] */
int mslink_count; /* # of missing-links in mslink[] */
unsigned short int bacy[MAXBACYS][LSIZE];
/* table of links of basic cycles */
/* in bit-field format */
{
  int i, j;
  int bacy_count = 0; /* # of bacycles */
  int list[MAXLIAIS]; /* list of links along a loop */
  int nlinks; /* # of links in the loop */
  int onbit;

  for (i = mslink_count; i; i++) {
    loop_test(mslink[--i], sptree, list, &nlinks);
    for (j = liai_size; j >= 0; j--)
      112
bacy[bacy_count][j] = OFF;
for (j = nlinks; j;)
    onbit(bacy[bacy_count], list[--;j]);
if (+bacy_count > MAXBACYS) {
    printf("tERROR: # of basic cycles is too big\n");
    return;
}
}
printf("t# of basic cycles = %d\n", bacy_count);
return(bacy_count);
}

/* *************************************************************/
/* */
/* Find all cycles from all the combinations of basic cycles. */
/* */
/* The combination generates 3 types of results: */
/* */
/* 1) one independent loop, */
/* */
/* 2) two or more independent loops, */
/* */
/* 3) loops that have common edges (link-count != part-count). */
/* */
 المتحول، */

int mk_all_cycles(bacy, bacy_count, loop)
int bacy_count;
unsigned short int bacy[MAXBACYS][LSIZE];
LOOPPTR loop;
{
    int i, j;
    int lcount; /* links count */
    int pcount; /* parts count */
    int comb_count = 0; /* count of possible loops */
    int ploop_count; /* part count in a loop */
    int node[MAXPARTS+1]; /* table of nodes */
    int pl_tab[MAXPARTS+1][MAXLIAIS+1]; /* table of part # vs. link # */
    unsigned long int maxcomb; /* maximum combination # in bit format */
    unsigned long int comb; /* combination # */
    unsigned long int copy;
    unsigned short int lop[LSIZE]; /* list of links in a loop */
    /* in bit field format */

    for (i = part_max; i; i--) {
        for (j = llia_max; j; pl_tab[i][--;j] = OFF);
            for (j = part_tab[i].list_count; j)
                pl_tab[i][part_tab[i].llia_list[--j]] = ON;
    }

    if (bacy_count == 32)
        maxcomb = -0;
    else maxcomb = (1 << bacy_count) - 1;

    for (comb = maxcomb; comb; comb--)
    {
        copy = comb; /* comb contains padded bits representing */
        /* the set of basic cycles need to be */
        /* combined with */
        lcount = pcount = 0;
        for (i = llia_size; i >= 0; i--) lop[i] = OFF;
        for (i = part_max; i; i--) node[i] = OFF;

        for (i = 0; copy; i++)
            if (copy & 1)
                for (j = llia_size; j >= 0; j--)
                    lop[j] ^= bacy[i][j];
            copy >>= 1; /* shift copy 1 bit to the right to */
            /* test which basic-cycle-list to */

for (i = liai_max; i; i--)
    if (is_bit_on(lop, i)) {
        lcount++;
        if (++node[liai_tab[i].part_x] == 2) pcount++;
        if (++node[liai_tab[i].part_y] == 2) pcount++;
        /* count how many parts there are in lop */
    }

if (lcount == pcount && one_loop(lcount, pcount, lop, pl_tab)) {
    for (i = liai_size; i >= 0; i--)
        loop[comb_count].liai_tab[i] = lop[i];
    loop[comb_count].lcount = lcount;
    if (++comb_count > MAXLOOPS)
        printf("\tERROR: # of loops is too big\n");
}

printf("\t# of total possible loops = %d\n", comb_count);
return(comb_count);

int one_loop(lcount, pcount, lop, pl_tab)
int lcount;    /* links count */
int pcount;    /* parts count */
unsigned short int lop[]; /* list of loop-links in bit field format */
int pl_tab[MAXPARTS+1][MAXLIAIS+1]; /* table of part # vs. link # */
{
    int i;
    int x;    /* current part # */
    int target_y;    /* target part # */
    int link_i;    /* previous link # */
    int l_j;    /* current link # */
    int ploop_count = 0;    /* count of parts in the loop */
    int pointer;    /* pointer to the current entry in paths[] */
    int pathp = 0;    /* end pointer to last+1 entry in paths[] */
    int flag = ON;    /* flag for finding the target part # */
    int paths[MAXLIAIS]; /* list of links in the testing loop */

    for (i = liai_max; i; i--)
        if (is_bit_on(lop, i))
            paths[pathp++] = i;

    link_i = paths[--pathp];
    x = liai_tab[link_i].part_x;
    target_y = liai_tab[link_i].part_y;

    while (flag) {
        pointer = pathp;
        while (TRUE) {
            l_j = paths[--pointer];
            if (pl_tab[x][l_j] & link_i != l_j) {
                ploop_count++;
                if (liai_tab[l_j].part_x == x) /* get a new x */
                    x = liai_tab[l_j].part_y;
                else x = liai_tab[l_j].part_x;
            }
        }
    }
}
if (x == target_y) {
    flag = OFF;
ploop_count++;
    break;
}

link_i = l_j;
paths[pointer] = paths[--pathp];
    /* replace the entry that has the found l_j with */
    /* the last link in the paths[] list */
break;
}

if (ploop_count == pcount)
    return (TRUE);
else return (FALSE);

/*************************************************************
/* screen_cycles() */
在广州 eliminate all the redundant cycles from all the generated loops. */
/*************************************************************

int screen_cycles(comb_count, loop, ncp)
int comb_count;    /* # of possible cycles */
LOOPPTR loop;     /* pointer to loop that contains all possible cycles */
NRCPTR ncp;       /* structure pointer non-redundant cycle */
{
    int i, j, k;
    int pointer;     /* pointer to the list[] */
    int index;      /* # of common links of 2 loops */
    unsigned short int test; /* test bytes for bit field variable */
    int budp = comb_count; /* boundary pointer to the list */
    int list[MAXLOOPS]; /* list of pointers to all cycles */

    ncp->count = 0;
    for (i = comb_count; i;) list[--i] = i;
        /* each entry in list[] contains */
        /* the corresponding index */

    while (budp) {
        i = list[--budp];
        pointer = budp;

        while (pointer) {
            j = list[--pointer];

            if (loop[i].lcount != loop[j].lcount) {
                index = 0;

                for (k = liai_size; k >= 0; k--) {
                    test = loop[i].liai_tab[k] & loop[j].liai_tab[k];
                    while (test) { /* count # of common links */
                        if (test & 1)
                            index++;
                        test >>= 1;
                    }
                }

                if (loop[i].lcount == index + 1)
                    list[pointer] = list[--budp];
                    /* shrink the list of cycles */
                    /* need to be checked */
            }
        }
    }

    return (TRUE);
}

else if (loop[j].lcount == index + 1)
    break; /* the cycle i is eliminated */

} 
if (pointer == 0) { /* the cycle i survive the tests with others */
    ncp->list[ncp->count].nlist = 0;
    for (j = 0; j <= liai_max; j++)
        if (is_bit_on(loop[i].liai_tab, j)) {
            ncp->list[ncp->count].nlist++;
            printf("%d ", j);
        }
    printf("\n");
    for (j = liai_size; j >= 0; j--)
        ncp->list[ncp->count].list[j] = loop[i].liai_tab[j];
    if (++ncp->count > MAXNRCYS) {
        printf("\tERROR: # of non-redundant cycles is too big\n");
        return;
    }
}

ncp->size = ncp->count/16;
pointer = 0;

for (i = liai_max; i; i--)
    if (liai_tab[i].exist) {
        ncp->licy[i].ncrctype = ncp->licy[i].ncryps = pointer;
        for (j = ncp->count; j;)
            if (is_bit_on(ncp->list[--j], liai_tab[i])) {
                ncp->li_tab[pointer++] = j;
                ncp->licy[i].ncrctype++;
            }
        ncp->licy[i].ncrctype--; /* point to the last element */
    }
printf("\t# of non-redundant loops = %d\n", ncp->count);
typedef unsigned short int TABLE;

typedef struct rank_info {
    TABLE p_in_tab[LSIZE]; /* paths in, just established liaisons */
    TABLE liai_tab[LSIZE]; /* established liaisons of current state */
    int nrcy_list[MAXNRCYS]; /* each element contains a count of undone */
        /* liaison in the respective NRC */
} RANK, *RANKPTR;

struct state_info {
    int ltab[MAXLIAIS];  /* established liaisons of current state */
    int p_out[MAXLIAIS]; /* paths out, next prospective liaisons */
    int lp;            /* p_out[]'s list pointer */
    TABLE p_out_tab[LSIZE]; /* compact table of paths out */
    int childp;        /* pointer to the start of child state list */
    int nchild;        /* # of child state */
} st;

int child_listp; /* pointer to the child_list */
int child_list[MAXSTATTRANS];

int thisrk_ncol; /* n columns in the current rank */
int nextrk_ncol; /* n columns in the next rank */

RANK rank_i[MAXCOLUMNS]; /* alternating parent rank and */
RANK rank_j[MAXCOLUMNS]; /* child rank */

void mk_seq()
/* Make mechanical assembly sequences and send them to the output stream. */
/* Assume 1) all liaisons between two parts are mated simultaneously. */
/* 2) # of ranks = total # of parts. */
***********************************************************************

int mk_seq(ncp, fp)
    NRCY PTR ncp; /* structure non-redundant cycles */
    FILE *fp; /* pointer to the output stream */
{
    int i;
    int rank, column; /* # of ranks & columns of current rank */
    int max_col = 0; /* # of maximum columns */
    int nstate = 0; /* # of generated states */
    RANKPTR this_rkp; /* pointer to this rank */
    RANKPTR next_rkp; /* pointer to next rank */
    RANKPTR temp; /* temporary pointer */
    RANKPTR this_step; /* pointer to the current state */
    this_rkp = &rank_i[0]; /* set pointers to this rank's and */
next_rkp = &rank_j[0]; /* the next rank's states */
this rk ncol = 1; next rk ncol = 0;
child_listp = 0;

array_init(this rk->p_in_tab, OFF, liai_size+1);
array_init(this rk->liai_tab, OFF, liai_size+1);
for (i = ncp->count; i;)
    this rk->nrcy_list[i] = ncp->list[--i].nlist;

fprintf(fp, "%4x%4x%4x%4x%4x", nstate, child_listp, rank, max_col,
        liai_max, part_max);
/* reserved space for nstate, child_listp, nrank, and max_col */

for (rank = 0; rank < part_count; rank++) {

    nstate += this rk ncol;
    fprintf(fp, " %x %x", rank, this rk ncol);

    for (column = 0, this_step = this rk p;
            column < this rk ncol; column++, this_step++) {
            printf("\n\n rank: %d, column: %d", rank, column);

        /*
** Search for prospective liaisons and put them in st.p_out[] list.
*/
        expand(st.itab, this_step->liai_tab, liai_max);
        st.lp = 0;
        array_init(st.p_out_tab, OFF, liai_size+1);
        st.childp = child_listp;
        st.nchild = 0;

        for (i = liai_max; i; i--)
            if (liai_tab[i].exist & & !st.itab[i] & & rules(st.itab, i))
                st.p_out[st.lp++] = i;

        printf("\njust mated liaisons:");
        record(fp, this_step->p_in_tab);

        printf("\n established liaisons:");
        record(fp, this_step->liai_tab);

        printf("\nchild-state column #: ");
        mk_child_state(this_step, next rk p, ncp);
            /* create states in the next rank from */
            /* the present state */

        printf("\nnext liaisons:");
        record(fp, st.p_out_tab);

        fprintf(fp, " %x %x", st.childp, st.nchild);
    }

    temp = this rk p; /* swap this rk p and next rk p for */
    this rk p = next rk p; /* preparing the next rank */
    next rk p = temp;

    if (this rk ncol > max_col)
        max_col = this rk ncol;

    this rk ncol = next rk ncol;
    next rk ncol = 0;
printf("\n\nMaximum # of columns = %d\n# of state transitions = %d\n", max_col, child_listp);

for (i = 0; i < child_listp; i++)
   fprintf(fp, " %x", child_list[i]);

fseek(fp, 0L, SEEK_SET); /* point to the beginning of the file */
fprintf(fp, "%x%4x%4x%4x", nstate, child_listp, rank, max_col);

fseek(fp, 0L, SEEK_END); /* point to the end of the file */
}

*******************************************************************************/
/* mk_child_state() */
/* Generate all the states in the next rank from the present state. */
/* 1. Assume simultaneous liaisons establishment. */
/* 2. Check if the last liaison of a non-redundant cycle is permitted by */
/* the precedence relationships. If it is not permitted, a deadend */
/* is found. An error message will be displayed and that newly */
/* generated state will be discarded because of assumption (1). */
*******************************************************************************/

int mk_child_state(this_step, next_rkp, ncp)
RANKPTR this_step; /* pointer to the current state */
RANKPTR next_rkp; /* pointer to the next rank */
RANKPTR ncp; /* structure non-redundant cycles */
{
   int i, j;
   int done_list[MAXLIAIS], dlp;
   /* list of established liaisons of current state */
   RANKPTR next_step; /* pointer to the next state */

   while (st.lp) {
      dlp = 0;
      next_step = next_rkp + next_rk_ncol;
      if (next_rk_ncol == MAXCOLUMNS) {
         printf("\n\nFATAL ERROR: # of columns exceed the upper limit
%d\n", MAXCOLUMNS);
         printf("\nPlease readjust the limit.\n");
         exit(0);
      }

      array_init(next_step->p_in_tab, OFF, liai_size+1);
      array_copy(next_step->liai_tab, this_step->liai_tab, liai_size+1);
      array_copy(next_step->nrcty_list, this_step->nrcty_list, ncp->count);

      /* Take an available prospective liaison from st.p_out[]. */
      i = done_list[dlp++ = st.p_out[--st.lp]];
      onbit(next_step->p_in_tab, i);
      onbit(next_step->liai_tab, i);

      /* Find all the liaisons that need to be established simultaneously when */
      /* liaison i is done. */
      find_liai(done_list, &dlp, this_step, next_step, ncp);

      /* Test if those that are found from above are legitimate prospective */
      /* next liaisons. If yes, a state will be generated in the next rank */
      /* or else check them off from st.p_out[] list. */
   }

119
if (rules_check(done_list, &dlp, this_step))
    update(done_list, dlp, this_step, next_step, next_rkp);
else {
    /* inconsistent PR to liaisons from p_out[] list */
    printf ("\nWARNING: Error in simultaneous liaisons establishment.
State is aborted.\n\"\n    printf("\n\tliaison \#:"");
    for (i = dlp-1; i >= 0; i--) {
        for (j = st.lp-1; j >= 0; j--)
            if (st.p_out[j] == done_list[i]) {
                st.p_out[j] = st.p_out[--st.lp];
                break; /* check the OK's one off */
            }
    }
    printf(" \d", done_list[i]);
    }
    printf("\n\n");
}

**************************************************************************
/* find_lastliais() */
/* When a liaison is picked from p_out_tab, another liaisons may be */
/* need to be established simultaneously. This last liaison can be */
/* searched from all the non-redundant cycles. */
**************************************************************************

int find_lastliais(list, plp, this_step, next_step, ncp)
int list[], *plp; /* list and list pointer of done liaisons */
RANKPTR this_step; /* pointer to the state of the current rank */
RANKPTR next_step; /* pointer to the state of the next rank */
NRCYPTR ncp; /* pointer to non-redundant cycle structure */
{
    int i, flag;
    int point; /* pointer to the non-redundant cycle list */
    int pointer; /* pointer to list[] */
    int cy; /* NRCY # */
    int lastl; /* last liaison # in the loop */

    /*
    * List contains all the liaisons need to be established in order to
    * make a state in the next rank. If a last-liaison is found, it is
    * add to the list and see if it can generate more last liaisons
    * from other non-redundant cycles.
    */
    for (pointer = 0; pointer < *plp; pointer++)
        for (point = ncp->liy[list[pointer]].nrcyps;
            point <= ncp->liy[list[pointer]].nrcype; point++) {
            /* check each NRC that contains liaison */
            /* from stack[pointer] */
            cy = ncp->li_tab[point];
            if (next_step->nrcy_list[cy] & & next_step->nrcy_list[cy] == 1) {
                for (lastl = liai_max; (!is_bit_on(ncp->list[cy].list, lastl) ||
                    is_bit_on(next_step->liai_tab, lastl)) & & lastl; lastl--)
                    /* find last liaison */
                    if (lastl) {
                        flag = TRUE; /* assume last1 is not in the list */
                        for (i = 0; i < *plp; i++)
                            ...
if (list[i] == last1) {
    flag = FALSE;  /* skip it if last1 is */
    break;  /* already in the list */
}

if (flag) {  /* a new one is found, put it in list */
    list[(*plp)++] = last1;
    onbit(next_step->liai_tab, last1);
    next_step->nrcy_list[cy] = 0;
    /* if it is also is st.p_out, write it off */
    for (i = st.lp-1; i >= 0; i--)
        if (st.p_out[i] == last1) {
            st.p_out[i] = st.p_out[--st.lp];
            break;
        }
}
}

/*******************************************************************************

   rules_check()

   See if any of the simultaneous liaisons violates precedence
   relationships.

*******************************************************************************

int rules_check(list, plp, this_step)
int list[], *plp;  /* list and list pointer */
RANKPTR this_step;  /* pointer to the state in the current rank */
{
    int i, count;
    int flag;  /* flag indicates if any liaison is satisfied by PR */
    int pointer, lp = 0;  /* pointer to the list */
    int liai[MAXLIAIS], tmp;

    if (*plp > 1) {
        lp = *plp;
        flag = ON;
        expand(liai, this_step->liai_tab, liai_max);
        /*
         ** It is possible that some of the simultaneous liaisons are established in
         ** some required order so they need to be rearranged
         ** and checked them again using rules().
         */
        while (lp & flag) {
            flag = OFF;
            for (pointer = lp-1; pointer >= 0; pointer--)
                if (rules(liai, list[pointer])) {
                    liai[tmp] = list[pointer] = ON;
                    list[pointer] = list[lp-1];
                    list[--lp] = tmp;
                    /* swap the current one with the one */
                    /* at the end of list[] */
                    flag = ON;
                }
        }
    }
    if (lp > 0) {
        /* some liaisons are invalid unless there is a PR */
        /* that constrains all the liaisons from the list */
        count = 0;
        for (i = 0; i < lp; i++) /* turn on the not OK's first */
            liai[li[i]] = ON;

    121
for (i = lp; i < *plp; i++) {  /* re-check all the OK's */
    liai[list[i]] = OFF;  /* ones, one at a time */
    if (rules(liai, list[i]))
        count++;
    liai[list[i]] = ON;
}

if (count == 0)  /* all liaisons are not permitted by PR */
    *plp = 0;
else {
    for (i = 0; i < lp; i++) {  /* check if the new */
        liai[list[i]] = OFF;  /* state is permitted */
        if (rules(liai, list[i]))
            count++;
        liai[list[i]] = ON;
    }
    if (count != *plp)  /* a definite error in PR */
        return (FALSE);  /* a possible error */
    printf("\nWARNING: Possible error in simultaneous liaisons establishment.\n")
}
}

return (TRUE);  /* liaisons are satisfied by all PR consistently */

/******************************************************************************
/                        update()                                    */
/  Check if any of the next state is already in the list and update     */
/  information.                                                      */
*******************************************************************************/

int update(done_list, dlp, this_step, next_step, next_rkp)
int done_list[], dlp;  /* list of required mating liaisons */
RANKPTR this_step;  /* pointer to the state in the current rank */
RANKPTR next_step;  /* pointer to the state in the next rank */
RANKPTR next_rkp;  /* pointer to the next rank's first state */
{
    int i, j;
    int flag = TRUE;  /* flag for finding a non-identical state */
    int col;  /* column */
    if (dlp == 0) return;

    for (col = 0; flag && col < next_rk_ncol; col++) {
        flag = FALSE;
        for (j = liai_size+1; j >= 0; j--)
            if (((next_rkp + col)->liai_tab[j] != next_step->liai_tab[j])) {
                flag = TRUE;
                break;  /* it is not an identical state */
            }
    }

    if (!flag) col--;

    next_step = next_rkp + col;  /* col has now either an old or */
    /* new location of the state */

    if (col == next_rk_ncol)  /* check if a new state is generated */
        next_rk_ncol++;
    /* in the next rank */

    return (TRUE);
}
for (i = dip-1; i >= 0; i--) {
    onbit(st.p_out_tab, done_list[i]);
    onbit(next_step->p_in_tab, done_list[i]);
}

printf("%d", col);

child_list[child_listp++] = col;
st.nchild++;

if (child_listp > MAXSTATTRANS) {
    printf("\tERROR: # of state transitions is too big\n");
    exit(0);
}

//****************************************************************************
/*
/* Print an array to an output stream in hexadecimal form.
/*
****************************************************************************/

int record(fp, array)
FILE *fp;
int array[];
{
    int i;

    for (i = liai_max; i--; i--)
        if (is_bit_on(array, i)) {
            printf("%d", i);
        }

    for (i = 0; i <= liai_size; i++)
        fprintf (fp, " %x", array[i]); /* the character field is 2 bytes */
}

array_init(to, const, n)
TABLE to[];
int const, n;
{
    while (n--)
        to[n] = const;
}

array_copy(to, from, n)
TABLE to[], from[];
int n;
{
    while (n--)
        to[n] = from[n];
}

expand(to, from, n)
TABLE from[];
int to[], n;
{
    while (n >= 0)
        to[n--] = is_bit_on(from, n);
}
#define ESCAPE 27
#define TRUE 1
#define FALSE 0
#define ON 1
#define OFF 0
#define HILITE 2
#define DELETE 0
#define APPEND 1
#define STT 1
#define STST 2
#define TRAN 3
#define NTRL 0
#define UP 1
#define DOWN 2
#define LEFT 3
#define RIGHT 4
#define LSIZE 2
#define N(object, type) (sizeof(object) / sizeof(type))

#define BTWN(lowbdry, point, upbdry) ((lowbdry) <= (point) && (point) <= (upbdry))
#define WITHIN(bdry) (BTWN(bdry.nwx, mx, bdry.sex) && BTWN(bdry.sey, my, bdry.nwy))

#define MSCHGE (msw > 127)
#define RB (msw & 1)
#define MB (msw & 2)
#define LB (msw & 4)

#define OPTN_END NULL, NULL

typedef unsigned short int TABLE;
typedef int BOOLEAN;

typedef struct optn_info {
    char *title;           /* option-title string */
    int (*fctn)();         /* pointer to the respective function */
} OPTN_DEF, *OPTN_DEF_PTR;

typedef struct bdry_pts {
    int fore, back;       /* foreground and background colors */
    float nwx, nwy;       /* North West coordinate of boundary */
    float sex, sey;       /* South East coordinate of boundary */
} BDRY, *BDRY_PTR;

typedef struct frame_info {
    BDRY title;           /* window title */
    BDRY l;               /* scroll left button */
    BDRY r;               /* scroll right button */
    BDRY u;               /* scroll up button */
typedef struct gph_info {  
    int fore, back; /* foreground and background colors */  
    int hilite;   /* highlight color */  
    float mwx, mwy; /* North West BDHYinate of display section */  
    float sx, sy; /* South East BDHYinate of display section */  
    float h_offset, v_offset; /* display offsets from the top */  
    /* middle of the unassembled state */  
    float box_w, box_h; /* width and height of the state box */  
    float h_space, v_space; /* spacing between states and ranks */  
    float x_inc, y_inc; /* size of each cell in a state-box */  
} GRAPH, *GRAPHPTR;

typedef struct state_info {  
    TABLE p_in_tab[LSIZE]; /* paths in, just established liaisons */  
    TABLE lia1_tab[LSIZE]; /* established liaisons of this state */  
    TABLE p_out_tab[LSIZE]; /* paths out, next prospective liaisons */  
} STATE, *STATEPTR;

typedef struct select {  
    STATEPTR st; /* pointer to the state structure */  
    struct select **parentp; /* pointer to the start of the parent state */  
    struct select **childp; /* pointer to the start of the child state */  
    struct select **next; /* pointer to the next selected state */  
    int status; /* state is selected or highlighted */  
    int col; /* column # in the rank */  
    int nparent; /* number of parents */  
    int ndparent; /* number of deleted parents */  
    int nchild; /* number of children */  
    int ndchild; /* number of deleted children */  
} SELECT, *SELECTPTR;

typedef struct rank_info {  
    SELECTPTR ptr; /* pointer to the first selected state */  
    SELECTPTR dptr; /* pointer to the first deleted state */  
    int ncol; /* # of selected states */  
} RANK, *RANKPTR;
extern BDRY menu;
extern BDRY scrn;
extern BDRY mbxbrdr, zmibdr, zmobdr, viewbrdr, delbrdr;
extern BDRY mbx_dstate, mbx_dstst, mbx_dtran, mbx_redraw, mbx_trash;
extern GRAPH zmi;
extern FRAME zmif;
extern GRAPH zmo;
extern GRAPH view;
extern FRAME viewf;
extern GRAPH del;
extern FRAME delf;
extern int dev_max_x, dev_max_y;
extern int image_buffer;
extern int image_offset;
extern float one_x, one_y;
extern float two_x, two_y;
extern float mx, my;
extern int msw;
extern int nrank, max_col;
extern int lsi1_max, part_max;
extern int lsi1_size;
extern int x_num, y_num;
extern SELECTPTR view_stp;
extern int view_rnk;
extern BOOLEAN zmowdw_flag;
extern int argc;
extern char **argv;
extern BOOLEAN file_status;
extern char* printfile;
int ed_stt(flag, rnk, stp)
int flag;  /* Delete or Append */
int rnk;   /* the rank of the state */
SELECTPTR stp; /* pointer to the state */
{
    mouse(OFF);

    if (flag == DELETE) {
        /*
         ** Highlighted the state itself and any single dependent parent or child
         */
        edparent(DELETE, rnk, stp);
        edchild(DELETE, rnk, stp);
    }

    else if (flag == APPEND) {
        /*
         ** Append the state itself and all deleted parent or child
         */
        edparent(APPEND, rnk, stp);
        edchild(APPEND, rnk, stp);
    }

    setclip(&scrn.nwx, &scrn.nwy, &scrn.sx, &scrn.sey);
    mouse(ON);
}

int ed_stt()  /*
/* Edit (DELETE or APPEND) a given state and its dependents. */
/* The delete state will first be given a highlighted status so that the */
/* display subroutine can highlight it with other selected states. The */
/* highlighted states will disappear from the screen only if the screen */
/* is redrawn by clicking the ReDrawn menubox. */
*/*
***************************************************************************
int ed_stst(flag, rnk, stp, tran_table)
int flag;    /* Delete or Append */
int rnk;     /* the rank of the parent state */
SELECTPTR stp;  /* pointer to the involved parent state */
TABLE tran_table[]; /* state transition table */
{
    SELECTPTR find_c_sp(), c_sp;

    mouse(OFF);

    if (flag == DELETE) {
    /*
    ** Highlighted the state itself and any single dependent parent or child
    */
        if ((c_sp = find_c_sp(stp, 0, stp->nchild, tran_table)) == NULL)
            return (FALSE);

        if (delete(c_sp, stp->childp, &stp->nchild, &stp->ndchild))
            edparent(DELETE, rnk, stp);

        if (delete(stp, c_sp->parentp, &c_sp->nparent, &c_sp->ndparent))
            edchild(DELETE, rnk+1, c_sp);
    }
    else if (flag == APPEND) {
    /*
    ** Append the state itself and all deleted parent or child
    */
        if ((c_sp = find_c_sp(stp, stp->nchild,
                        stp->nchild+stp->ndchild, tran_table)) == NULL)
            return (FALSE);

        if (append(c_sp, stp->childp, &stp->nchild, &stp->ndchild))
            edparent(APPEND, rnk, stp);

        if (append(stp, c_sp->parentp, &c_sp->nparent, &c_sp->ndparent))
            edchild(APPEND, rnk+1, c_sp);
    }

    show_statran(stp, rnk, c_sp, rnk+1);

    setclip(&scrn.nwx, &scrn.nwy, &scrn.sex, &scrn.sey);
    mouse(ON);
}

/******************************************************************************
/* edparent() */
/* Edit (DELETE or APPEND) all the parents of a given state. */
/******************************************************************************

int edparent(flag, rnk, stp)
int flag;    /* Delete or Append */
int rnk;     /* the rank of the state */
SELECTPTR stp;  /* pointer to the state */
{
    SELECTPTR *p_sp;

    if (flag == DELETE) {
    /*
    ** Delete all its parents
    */
        for (p_sp = stp->parentp; p_sp < stp->parentp + stp->nparent; p_sp++)
            if (((*p_sp)->status == ON && delete(stp, (*p_sp)->childp,
                        (*p_sp)->nchild, (*p_sp)->ndchild)) (}
(*p_sp)->status = HILITE;
edparent(DELETE, rnk-1, *p_sp);
}

stp->ndparent += stp->nparent;
stp->nparent = 0;
stp->status = HILITE; /* highlighted the just deleted state */
show_state(rnk, stp);

} else if (flag == APPEND) {

/**
** Append all its parents
*/

for (p_sp = stp->parentp + stp->nparent; 
p_sp < stp->parentp + stp->nparent + stp->ndparent; p_sp++)

if (append(stp, (*p_sp)->childp,
&(*p_sp)->nchild, &(*p_sp)->ndchild)) {

(*p_sp)->status = ON;
edparent(APPEND, rnk-1, *p_sp);
}

stp->nparent += stp->ndparent;
stp->ndparent = 0;
stp->status = ON;
show_state(rnk, stp);
}

/****************************************************************************
/*
** Edchild()
*/
/****************************************************************************

int edchild(flag, rnk, stp)
int flag; /* Delete or Append */
int rnk; /* the rank of the state */
SELECTPTR stp; /* pointer to the state */
{

SELECTPTR *c_sp;

if (flag == DELETE) {

/**
** Delete all its children
*/

for (c_sp = stp->childp; c_sp < stp->childp + stp->nchild; c_sp++)
if (((*c_sp))->status == ON && delete(stp, (*c_sp)->parentp,
&(*c_sp)->nparent, &(*c_sp)->nchild)) {

(*c_sp)->status = HILITE;
edchild(DELETE, rnk+1, *c_sp);
}

stp->nchild += stp->nchild;
stp->nchild = 0;
stp->status = HILITE; /* highlight the just deleted state */
show_state(rnk, stp);

} else if (flag == APPEND) {

/**
** Append all its children
*/

for (c_sp = stp->childp + stp->nchild;
    c_sp < stp->childp + stp->nchild + stp->ndchild; c_sp++)
if (append(stp, (*c_sp)->parentp,
&(*c_sp)->nparent, &(*c_sp)->nchild)) {

(*c_sp)->status = ON;
edchild(APPEND, rnk+1, *c_sp);

}
} 
stp->nchild += stp->ndchild; 
stp->ndchild = 0; 
stp->status = ON; 
show_state(rnk, stp); 
}

/*********************************************************
/* delete() */
/* Move the state pointer from the list1 given by ptr and numlp to list2 */
/* denoted by num2p. It is used for parentp and childp. */
/*********************************************************

BOOLEAN delete(stp, ptr, numlp, num2p)
SELECTPTR stp, *ptr; 
int *numlp, *num2p; 
{
  int pointer;
  SELECTPTR temp;

  for (pointer = 0; *(ptr+pointer) != stp; pointer++);
    /* find where stp is */
  temp = *(ptr+pointer); 
  *(ptr+pointer) = *(ptr + --(*numlp)); 
  *(ptr + *numlp) = temp; /* move stp from list1 to list2 */
  (*num2p)++;

  if (*numlp == 0)
    return (TRUE); /* single dependence relationship */
  else return (FALSE);
}

/*********************************************************
/* append() */
/* Move the state pointer from the deleted-list given by ptr and num2p to list2 */
/* to kept-list denoted by numlp. It is used for parentp and childp. */
/* Retrun TRUE if the state ,stp, is the first appended parent or child. */
/*********************************************************

BOOLEAN append(stp, ptr, numlp, num2p)
SELECTPTR stp, *ptr;
int *numlp, *num2p;
{
  int pointer;
  SELECTPTR temp;

  for (pointer = *numlp; *(ptr+pointer) != stp; pointer++);
    /* find where stp is */
  temp = *(ptr+pointer); /* move stp from deleted-list to kept-list */
  *(ptr+pointer) = *(ptr + *numlp);
  *(ptr + (*numlp)++) = temp;
  (*num2p)++;

  if (*numlp == 1)
    return (TRUE); /* single dependence relationship */
  else return (FALSE);
}

/*********************************************************
/* show_state() */
130
/* Update the picture of the state both in zoom-in and zoom-out window */
/* when the state has positive column #. It is used to screen all */
/* the states just recovered from OFF to ON (i.e., not on the screen */
/* currently) during the edit session. */

int show_state(rnk, stp)
int rnk;    /* the rank of the state */
SELECTPTR stp; /* pointer to the state */
{
    if (stp->col != 0) {
        setclip(&zmi.nwx, &zmi.nwy, &zmi.sex, &zmi.sey);
        disp_state(&zmi, stp, rnk);
        if (zmowdw_flag) {
            setclip(&zmo.nwx, &zmo.nwy, &zmo.sex, &zmo.sey);
            disp_state(&zmo, stp, rnk);
        }
    }
}

int show_statran(stp1, rnk1, stp2, rnk2)
{
    setclip(&zmi.nwx, &zmi.nwy, &zmi.sex, &zmi.sey);
    disp_statran(&zmi, stp1, rnk1, stp2, rnk2);
    if (zmowdw_flag) {
        setclip(&zmo.nwx, &zmo.nwy, &zmo.sex, &zmo.sey);
        disp_statran(&zmo, stp1, rnk1, stp2, rnk2);
    }
}
```c
#include <stdio.h>
#include <math.h>
#include "clrcode.h"
#include "scommon.h"
#include "sexttern.h"

extern RANKPTR rank;

long int count_seq()
{
    int rank, i;
    static long int *sre_p, *tgt_p;
    long int *p;
    SELECTPTR p_sp, *c_sp;

    if ((sre_p = (long int *) alloca (max_col * sizeof(long int))) == NULL)
        memory_err();
    if ((tgt_p = (long int *) alloca (max_col * sizeof(long int))) == NULL)
        memory_err();

    tgt_p[0] = 1; /* start from the unassembled state */

    for (rank = 0; rank < nrank - 1; rank++) {
        p = sre_p; /* swap the source and target pointer */
        sre_p = tgt_p;
        tgt_p = p;
        for (p = tgt_p; p < tgt_p + rank[nrank+1].ncol; *(p++) = 0);

        if ((p_sp = rank[rank].p) == NULL) return(0);
        /* clear the target array */
        /* in case there is no column in the rank */
        do
            if (p_sp->status == ON)
                for (i = 0, c_sp = p_sp->childp; i < p_sp->nchild; i++, c_sp++)
                    if (((c_sp)->status == ON)
                        *(tgt_p + (*c_sp)->col) += *(sre_p + p_sp->col);
                while ((p_sp = p_sp->next) != NULL);

        return(*tgt_p);
    }

/* move_hilite_to_off() */
/* Move all the HILITE states (if any) from the keep list to delete list. */

BOOLEAN move_hilite_to_off()
{
    BOOLEAN flag = FALSE; /* flag for any movement between the keep */
    /* ...
```
int rnk, col;
SELECTPTR stp, kp, dp;

for (rnk = 0; rnk < nrank; rnk++) {

col = 0; /* starting column # for the keep list state */
kp = dp = NULL; /* end ptr to the end of the keep and delete list */
stp = rank[rnk].ptr;
rank[rnk].ptr = NULL; /* clear the ptr for later assignment */

for (; stp != NULL; stp = stp->next)
    if (stp->status == HILITE) {
        flag = TRUE; /* a movement change */
        stp->status = OFF; /* the state is now deleted */
        stp->col = -1; /* for future display purpose */
        rank[rnk].ncol--; /* update total count */
        append_stp(&rank[rnk].dptr, &dp, stp);
/* move stp to the delete list */
    } else {
        stp->col = col++;
        append_stp(&rank[rnk].ptr, &kp, stp);
/* keep stp in the keep list */
    }

if (dp != NULL) /* mark the state at the end of the delete list */
    dp->next = NULL;

if (kp != NULL) /* mark the state at the end of the keep list */
    kp->next = NULL;

return (flag);
}

******************************************************************************
/* move_off_to_on() */
******************************************************************************

BOOLEAN move_off_to_on()
{
    BOOLEAN flag = FALSE; /* flag for any movement between the keep */
    int rnk; /* list and delete list */
    SELECTPTR stp, kp, dp;

    for (rnk = 0; rnk < nrank; rnk++) {

        kp = dp = NULL;
        stp = rank[rnk].dptr;
        rank[rnk].dptr = NULL; /* clear the dptr for later assignment */

        for (; stp != NULL; stp = stp->next)
            if (stp->status == ON) {
                flag = TRUE; /* a movement change */
                stp->col = rank[rnk].ncol++; /* its appended at the end of kept list */
                /* and update total count */
                append_stp(&rank[rnk].ptr, &kp, stp);
/* move stp to the keep list */
            } else {

            }

            if (dp != NULL) /* mark the state at the end of the delete list */
                dp->next = NULL;

            if (kp != NULL) /* mark the state at the end of the keep list */
                kp->next = NULL;

            return (flag);
}
append_stp(&rank[rnk].dptr, &dp, stp);
    /* keep stp in the delete list */
}

if (dp != NULL)    /* mark the state at the end of the delete list */
    dp->next = NULL;

if (kp != NULL)    /* mark the state at the end of the keep list */
    kp->next = NULL;

return (flag);

*******************************************************************************/
/* append_stp() */
/* Append a state pointer to the end of a given list and update the end */
/* pointer to the list. */
*******************************************************************************/

int append_stp(ptr, endp, stp)
SELECTPTR *ptr;    /* the pointer to first item in the list */
SELECTPTR *endp;    /* endp is the pointer to the state at the */
                    /* end of the list */
SELECTPTR stp;    /* pointer to the appended state */
{
    if (*ptr == NULL)    /* the first one in the delete list */
        *ptr = stp;
    else {
        if (*endp == NULL)    /* first find the end of the list */
            for (*endp = *ptr; (*endp)->next != NULL;
                *endp = (*endp)->next);

        (*endp)->next = stp;
        /* append stp to the end of the list */
    }

    *endp = stp;    /* update the end pointer of the delete list */
}
REFERENCES


