DEVELOPING A TECHNIQUE TO SUPPORT DESIGN CONCURRENT COST
ESTIMATION USING FEATURE RECOGNITION

A Thesis Presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University
In Partial Fulfillment
of the Requirement for the Degree
Master of Science
by
Ashutosh Trasi
March, 2001
Acknowledgements

First and foremost, I would like to express my sincere gratitude to Dr. Robert P. Judd. The successful completion of this thesis is largely attributed to his vision and guidance. To him, I will always remain grateful for everything that I have learned through his courses at Ohio University as well as this thesis.

I also wish to thank my advisor, Dr. David A. Koonce, for providing valuable suggestions and inputs during the course of writing my thesis. His co-operation in making this thesis a success has been invaluable. I am also thankful to Dr. Robert Lipset for showing great interest in this work and being co-operative at all times.

This work has been possible due to the admirable work done by Prahlad Athreya, the developer of CQL, which has been the foundation to this thesis. I would also like to thank my colleagues at CASSI, Vijay Ramachandran and Sachin D’souza, for helping me with their valuable suggestions during various stages of my thesis.

Finally, I would like to dedicate this thesis to my parents, my brother Abhijit, and to the love of my life, Ruta, for their constant love and encouragement.
Table of Contents

INTRODUCTION ...................................................................................................................... 1

1.1 INTRODUCTION ........................................................................................................ 1

1.2 PROBLEM STATEMENT .............................................................................................. 5

1.3 THE BACKWARD FILLING SOLUTION – AN ADVANCED TECHNIQUE ..................... 7

BACKGROUND .................................................................................................................... 9

2.1 FEATURE RECOGNITION OF 2.5D COMPONENTS ..................................................... 10

2.2 GRAPH BASED APPROACH ...................................................................................... 13

2.3 CSG BASED APPROACH ........................................................................................... 16

2.4 NEURAL NETWORK BASED APPROACH ................................................................. 16

2.5 DESIGN BY FEATURES ............................................................................................. 18

2.5.1 Destructive Solid Geometry (DSG) ...................................................................... 18

2.5.2 Compositional Feature Models ............................................................................ 19

2.5.3 Procedural Feature Languages ........................................................................... 22

2.6 OTHER IMPORTANT RESEARCH APPROACHES ...................................................... 22

2.7 CONCLUSION ............................................................................................................ 29

BACKWARD FILLING – APPROACH AND ALGORITHM .................................................... 30

3.1 THE APPROACH ........................................................................................................ 30

3.2 ASSUMPTIONS ........................................................................................................... 31

3.3 RECOGNITION OF FEATURES AND ASSOCIATED TERMINOLOGY ..................... 31

3.3.1 Inside and Outside Corners (Up-edges and Down-edges) ....................................... 32
3.3.2  Inside and Outside Arcs ................................................................. 34
3.3.3  Next Logical level ................................................................. 36
3.3.4  Loops or FaceLoops ................................................................. 38
3.3.5  Features Recognized ................................................................. 39
3.4  SUMMARY OF THE ALGORITHM ....................................................... 41
3.5  A SIMPLE EXAMPLE ................................................................. 44

IMPLEMENTATION ........................................................................ 49

4.1  THE UNIGRAPHICS API (APPLICATION PROGRAMMER’S INTERFACE) .... 49
4.2  EXTENSION TO CQL ................................................................. 50
  4.2.1  Extrusion ................................................................................. 50
  4.2.2  Boolean Operations – Intersection and Union ......................... 54
  4.2.3  Creation of a line ..................................................................... 57
  4.2.4  Exporting an Object ................................................................. 59
4.3  ROUTINES DEVELOPED IN TNR .................................................... 61
  4.3.1  Preliminary tasks and Output Routines – main.tnr .................... 61
  4.3.2  Feature Classification Routines – classify.tnr ......................... 62
  4.3.3  Backward Filling Operation Routines – fill.tnr ....................... 62
  4.3.4  Support Routines – support.tnr ................................................. 64
4.4  INPUT AND USER INTERACTION ..................................................... 68
4.5  OUTPUT ......................................................................................... 70
4.6  DATA STRUCTURES CREATED FOR OUTPUT ................................... 71
  4.6.1  Features[] ............................................................................... 71
  4.6.2  setUps [] ............................................................................... 72
## DEMONSTRATIONS

5.1 PART 1 ........................................................................................................74

5.1.1 Phase 1 of processing .............................................................................75
5.1.2 Phase 2 of processing .............................................................................76
5.1.3 Phase 3 of processing .............................................................................78
5.1.4 Phase 4 of Processing ............................................................................79
5.1.5 Phase 5 of processing ............................................................................80

5.2 PART 2 ........................................................................................................82

5.2.1 Phase 1 of processing ............................................................................83
5.2.2 Phase 2 of processing ............................................................................84
5.2.3 Phase 3 of processing ............................................................................85

5.3 PART 3 ........................................................................................................87

5.4 PART 4 ........................................................................................................91

5.5 PART 5 ........................................................................................................95

5.6 PART 6 ........................................................................................................99

5.7 PART 7 .......................................................................................................103

5.8 PART 8 .......................................................................................................107

5.9 PART 9 .......................................................................................................111

5.9.1 Phase 1 of Processing ............................................................................112
5.9.2 Phase 2 of Processing ............................................................................113
5.9.3 Phase 3 of Processing ............................................................................114
5.9.4 Phase 4 of Processing ............................................................................115
B.2  Find Direction of Machining and a Machinable Plane........................................... 138
B.3  Process the Machinable Plane................................................................................. 141
B.4  Classify a Face Loop.............................................................................................. 142
B.5  Classify an Edge .................................................................................................... 143
B.6  Fill all the Inside Loops on a Plane....................................................................... 145
B.7  Fill the inside loop representing a through hole ................................................... 146
B.8  Fill all the round corners/ blends existing on the plane ....................................... 147
B.9  Fill a round corner or blend ................................................................................... 149
B.10 Fill the slots defined by the outside loop of a plane ............................................. 153
B.11 Fill a slot formed on the outer loop of a plane .................................................... 156
B.12 Finding the edges that are tangent to the arc edge defining a slot.... 157
B.13 Constructing Lines to Square off a Slot ................................................................. 159
B.14 Extruding the plane after machining all features existing on the plane ................. 163
B.15 The Extrusion, Intersection and Union operations............................................... 165
B.16 Determining the “Next Logical Level” to find the limit of Extrusion for a feature ................................................................. 166
## List Of Figures

| Figure 2.1 | Example part for Convex Hull-Based Feature Recognition Method ..........12 |
| Figure 2.2 | AAG of a part ..................................................................................14 |
| Figure 2.3 | Recognizing a pocket feature using the AAG ......................................15 |
| Figure 2.4 | Example part for neural network based approach ..............................17 |
| Figure 2.5 | Example part to illustrate the use of a features database in process planning and manufacturability evaluation ......................................................20 |
| Figure 2.6 | Object hierarchy for part shown in Figure 2.5 (a) ............................21 |
| Figure 2.7 | Flow chart of cost analysis system proposed by Yang and Lin ..........28 |
| Figure 3.1 | Example of inside corner formed at an edge .......................................33 |
| Figure 3.2 | Example of an Outside corner formed at an edge ...............................33 |
| Figure 3.3 | Algorithm to find the inside and outside corners formed by edges .........34 |
| Figure 3.4 | Example of an outside arc formed by a round corner .........................35 |
| Figure 3.5 | Algorithm to distinguish between inside and outside arcs ...................36 |
| Figure 3.6 | Example demonstrating the concept of ‘Next Logical Level’ ...............37 |
| Figure 3.7 | Algorithm to determine the ‘Next Logical Level’ ................................38 |
| Figure 3.8 | Example of a simple part to be processed .........................................45 |
| Figure 3.9 | Solid objects corresponding to the various machined features in part  shown in figure 3.8 .................................................................46 |
| Figure 3.10 | Example of the Features file (all features not included) ......................47 |
| Figure 3.11 | Example of the setups file ................................................................48 |
Figure 5.23  Part 4 after processing slot 2 and slot 3 ........................................... 94
Figure 5.24  Part 4 transformed into raw stock ...................................................... 94
Figure 5.25  Part 5 .................................................................................................. 95
Figure 5.26  Part 5 after processing pocket with through hole and protrusion .......... 96
Figure 5.27  Part 5 after processing pocket 2 ......................................................... 96
Figure 5.28  Part 5 after processing pocket 4 ......................................................... 97
Figure 5.29  Part 5 after processing rectangular pad 1 and simple hole ................. 97
Figure 5.30  Part 5 after processing rectangular pad 2 ........................................... 98
Figure 5.31  Part 5 transformed into raw stock ..................................................... 98
Figure 5.32  Part 6 .................................................................................................. 99
Figure 5.33  Part 6 after processing simple hole .................................................... 100
Figure 5.34  Part 6 after processing the step feature .............................................. 100
Figure 5.35  Part 6 after processing round pocket .................................................. 101
Figure 5.36  Part 6 after processing rectangular pocket 1 ..................................... 101
Figure 5.37  Part 6 after processing rectangular pocket 2 ..................................... 102
Figure 5.38  Part 6 transformed into raw stock ..................................................... 102
Figure 5.39  Part 7 .................................................................................................. 103
Figure 5.40  Part 7 after processing step 1 .............................................................. 104
Figure 5.41  Part 7 after processing through hole 1 and step 1 ................................ 104
Figure 5.42  Part 7 after processing through hole 2 and step 2 ............................. 105
Figure 5.43  Part 7 after processing through hole 3 .............................................. 105
Figure 5.44  Part 7 transformed into raw part ....................................................... 106
| Figure 5.45 | Part 8 ........................................................................................................ 107 |
| Figure 5.46 | Part 8 after processing through hole ....................................................... 108 |
| Figure 5.47 | Part 8 after processing pocket 1 ................................................................. 108 |
| Figure 5.48 | Part 8 after processing slots 1 and 2 .......................................................... 109 |
| Figure 5.49 | Part 8 after processing slots 3 and 4 .......................................................... 109 |
| Figure 5.50 | Part 8 after processing the step feature with through hole ...................... 110 |
| Figure 5.51 | Part 8 transformed into raw stock .............................................................. 110 |
| Figure 5.52 | Part 9 ........................................................................................................... 111 |
| Figure 5.53 | Phase 1 of filling operation for part 9 ......................................................... 112 |
| Figure 5.54 | Phase 2 of filling operation for part 9 ......................................................... 113 |
| Figure 5.55 | Phase 3 of filling operation for part 9 ......................................................... 114 |
| Figure 5.56 | Failure phase of filling operation on part 9 ............................................... 115 |
Chapter 1

Introduction

1.1 Introduction

In today's world, as we enter the millennium, overwhelming accomplishments and breakthroughs in areas like communication and information technology have opened up vast opportunities for an enterprise to enhance its growth and make its place in the world market. These technological endeavors and achievements have made an enterprise susceptible to immense competition. This is evident from the development of various techniques and strategies, such as TQM, re-engineering, etc., to improve operating efficiencies and enhance product quality [1]. Cost estimation is one of the most important issues that determine the survival of any manufacturing product that competes in today's marketplace. Cost estimating is the prediction of expenses and investments that will be made to design, manufacture, and market a product [2]. Cost estimation finds a wide use in various areas of manufacturing and is utilized as a valuable tool by a wide range of people for diverse purposes. It is now widely accepted that one of the most powerful
methods for reducing manufacturing cost and increasing productivity is through the application of techniques of product design for efficient manufacturability [3]. According to Winchell [2], product cost estimates may be used for:

- Evaluating design proposals.
- Establishing a bid price of a product for a quotation or contract.
- Verifying the quotations submitted by suppliers.
- Ascertaining whether a proposed product can be manufactured and marketed profitably.
- Providing data for make-buy decisions.
- Determining the most economical method, process, or material for manufacturing a product.
- Providing a temporary standard for production efficiency and guide operating costs at the beginning of a project.

Concurrent engineering seeks to save on the overall product development time in the product design stage itself by analyzing the factors that affect the subsequent production processes. Several methods, such as DFM (Design for Manufacture), DFA (Design for Assembly) etc., have been developed to implement the ideas of concurrent engineering. Generically, these methods are referred to as DFX methods. Among these DFX methods, DFC (Design for Cost) is one area that has been considered by researchers recently. The basic concept of DFC is to estimate the manufacturing cost in the early design stage with a view to achieve the following objectives:
• To verify the feasibility of the design and modify it accordingly.

• To provide an environment to estimate alternative cost for comparative design models.

• To find the segment of a part model that might cause a high manufacturing cost.

After identifying the importance of cost estimation in manufacturing, it is also imperative to know about the major factors that affect the cost of a product. This thesis deals with the issue of machining prismatic or milled parts. Some of the main constituents of the cost of a part are listed below:

• The machining or removal volume i.e. the amount of material to be removed from the raw stock to produce the finished product.

• The type of tools to be utilized while machining.

• The type of processes to be employed for machining.

• The type of material to be used.

• The set-up costs of the part at each stage of the entire machining cycle.

This thesis presents a system that:

• Recognizes the features existing on the part,

• Provides accurate information about the volume to be removed from the raw material to produce the finished product,

• Provides accurate information about the area to be machined to produce each feature,

• Suggests the size of the milling cutter to be used in the machining process,
• Determines the number of set-ups required for machining the entire part.

All this information can be used as valuable and accurate input to a cost estimation procedure to finally arrive at the cost of manufacturing the part.

While the major objective of concurrent engineering is to consider the related downstream manufacturing during the design stage, estimating manufacturing cost is one area that has received little attention from researchers. The importance of estimating the product manufacturing cost at design stage lies in its ability to provide the designers with valuable information that can help them modify the design to achieve proper performance as well as a reasonable cost at an early stage of the product development process. Some of the major factors that affect the manufacturing cost of a part are the volume that needs to be removed from the raw material to produce the finished part, the number of set-ups required to machine the part, and the cutting tool used to perform the machining operation. This thesis presents a technique to estimate the manufacturing cost of prismatic parts. The technique involves recognizing the various features embedded in the part, calculating the volume to be removed to machine each of these features, estimating the number of set-ups required to machine the part, and suggesting the size of the cutter to be used in each machining operation. The approach, used in achieving the objectives of this research, is of a constructive nature and consists of transforming the finished part into the raw stock. The solution presented in this thesis, called the backward filling solution, builds the finished part into the raw part by filling up each feature existing on the finished part. Thus, the volume required to fill a feature represents the material to be
removed from the raw part to machine that particular feature. Tools such as CQL\(^1\) (Cad Query Language) and TNR\(^2\) (Templates and Rules) have been employed to extract data from the CAD model and utilize it, so that accurate removal volume can be calculated.

With the help of mechanisms developed in CQL such as ‘extrude’, ‘intersect’, ‘union’ etc., the solution developed in this thesis extracts the accurate machining volume of prismatic parts and predicts some of the important aspects of a process plan to manufacture the part. This data can be used further to provide an accurate cost estimate for machining the part. A significant part of the data provided by the developed system can also assist an automated cost estimation system in achieving the goals of design concurrent cost estimation.

1.2 Problem Statement

Current methods available for feature recognition suffer from:

- The computational inefficiencies caused within the pattern matching procedure during the process of recognizing interacting features,
- The inability to handle features formed by the combination of cylindrical and planar faces.

\(^1\) CQL is a query language developed at the Center for Software Systems Integration (CASSI), Industrial and Manufacturing Systems department, Russ College of Engineering and technology, Ohio University.

\(^2\) TNR is a scripting language developed at the Center for Software Systems Integration (CASSI), Industrial and Manufacturing Systems department, Russ College of Engineering and technology, Ohio University.
Some of these methods cannot be extended to non-planar faces and are restricted to planar faces only.

In design by features, the features can be primitives meaningful to either design or manufacturing. If manufacturing features are chosen as design primitives, it becomes easier to extract the volume which, to produce the finished part, is in need of being removed from the raw stock. However, such being the case, the freedom and creativity of the design will be reduced since manufacturing features constrain the product designer’s idea in terms of current manufacturing capabilities. On the other hand, if design features are used as design primitives, the manufacturing features would have to be recognized and extracted from the design model in order to automate the generation of a feasible process plan. The backward filling approaches that have been suggested in the past to find the machinable features and their volumes work only for specific features like protrusions and depressions and are incapable of handling features such as blends and through holes.

Some other methods, devised to automate the generation of a process plan and eventually the cost estimate, use pre-developed databases to obtain information about the machining operations and the tools that are required to machine the part. These methods fail to give correct information if the appropriate manufacturing process is not included in the database.
1.3 The Backward Filling Solution – An Advanced Technique

The method developed in this thesis overcomes some of the inefficiencies inherent in the previous feature recognition methods and also puts forward a more efficient backward filling solution for building the finished part into the raw stock. Moreover, it gathers information about machining features which can serve as valuable input to a cost estimation system. This technique/method involves utilizing the data provided by the CAD system by querying the CAD database, recognizing various features depending on the geometry of the boundary representation (B-rep) elements, utilizing mechanisms offered by the CAD system to build the finished part into the raw stock, and eventually, determining significant aspects of a process plan that will lead to a more accurate cost estimate to machine the part.

The backward filling method developed here requires a solid model of the finished part and a raw stock input. It utilizes the following tools:

- CQL queries the CAD database and gets information about the B-rep elements of the solid model.
- TNR manipulates the data retrieved by CQL, recognizes the various features existing in the part and finally suggests the appropriate process plan and a cost estimate for machining each feature.
The mechanisms, provided by the CAD system, which have been used in the implementation of this technique, are 'extrude', 'intersect' and 'union'. These operators have been developed in CQL and have been utilized in an attempt to construct the finished part into the raw stock. Based on the nature of each feature, its underlying geometry, and its interaction with other features, the cost estimate for machining the feature will be suggested.

This thesis provides a new method, to achieve the broader objective of design concurrent cost estimation, to assist a designer, who has little knowledge about the manufacturing process, to estimate the cost of a design during its conceptual stage and reduce unnecessary costs in the downstream process.
Chapter 2

Background

Features can be defined as ‘a subset of geometry on an engineering part which has a special design or manufacturing characteristic’ [5]. Feature recognition converts a CAD model into a feature model that is specific to the application. It requires extraction of embedded manufacturing features, along with their parameters and properties, from the CAD model. In this chapter, the approaches proposed by researchers, for feature recognition have been summarized. Also, an introduction to CQL and TNR has been provided. CQL or Cad Query Language was developed to access data from CAD databases. TNR (Templates and Rules) is a scripting language developed specifically for the purpose of enabling software integration and supports interfaces to several data resources. These tools were developed at the Center for Advanced Software Systems Integration (CASSI), Ohio University, and have been used to implement the backward filling method in this thesis.
2.1 Feature Recognition of 2.5D Components

In 1990, Ferreira and Hinduja [6] devised a combination of methods for recognizing features in 2.5D components. These methods were:

- The Convex hull-Based Method: This method constitutes recognizing features that originate from the convex hull of each face. This method takes each planar face of a component and detects the edges that are in the convex hull of the face. A convex hull can be defined as the minimum convex polygon that encloses a face. A convex polygon is, in turn, the one in which a straight line from one point to another lies wholly within the face. Once the edges are detected depending upon whether they are convex or concave a decision can be made about the feature to which they belong. From the nature of the edges, information about the direction of approach of cutter and machining depth can also be determined.

- The Inner Loop Method: In this method, features originating in the inner loop of edges are extracted. Similar to the previous method, the other faces common to these edges are marked. The inner loop of the edges is traversed in the clockwise direction, and the feature number is incremented every time a convex edge is encountered.

- The Analysis of Concave Edges: In some situations, the convex hull and inner loop methods fail to detect certain faces that form a part of the feature. These faces contain concave edges. The logic applied in this case is that, the faces forming a concave edge must have the same feature number.
• The Gluing Method: The methods described above analyze the faces and edges of the component and extract the features. However, two or more features, when joined together, can be more economical to machine than individual features. Two or more features are glued together if they satisfy the following conditions:
  • They have the same top surface
  • They have the same base surface
  • They are adjacent to each other

An example\textsuperscript{1} illustrating the mechanism of this technique is described below. There are two features shown in the example part. In Figure 2.1 (a), the first feature is recognized as a slot because it has three entry faces and it’s cross section has only one possible entry face. The three faces of the slot are marked as the ‘restriction faces’. The restriction faces are marked by taking one of the through faces of the slot as a reference face, selecting the common edges between the reference face and the slot and the then marking the faces belonging to these edges, other than the reference face. The second feature, highlighted in Figure 2.1(b), is recognized as a closed pocket because it has only one entry surface and it contains at least one side restriction face that is planar.

Since features 1 and 2 satisfy the conditions for gluing, they are merged into a single feature which is referred to as '1 pocket and 1 slot', as shown in Figure 2.1(c).

Figure 2.1  Example part for Convex Hull-Based Feature Recognition Method [6]
2.2 Graph Based Approach

In 1988, Joshi and Chang [7] suggested a graph-based heuristic for the recognition of machined features from a 3-D solid model. They developed a concept of attributed adjacency graph (AAG) for the recognition of machined features from a 3-D B-rep of a solid. They used a B-rep scheme and all the feature recognition algorithms were based on it. According to their theory, the topology of a part is described by a boundary model in which the primitive topological entities such as faces, edges and vertices are represented. Associated with these faces, edges and vertices are the corresponding geometrical definitions. The b-rep in its raw form provides low-level information that is not directly usable for feature recognition. Additional information regarding the type of face adjacencies and relationships between the sets of faces needs to be explicitly available to assist in feature recognition. To facilitate the recognition process, the concept of a attributed adjacency graph built on the underlying B-rep was proposed. The AAG converts the low-level B-rep into high-level structural entity whose primitives are the elements of a B-rep. Although this method was effective, its implementation was limited to polyhedral features such as pockets, slots, steps, blind steps, etc.

The AAG graph can be defined as a graph \( G = (N, A, T) \) where \( N \) is the set of nodes, \( A \) is the set of arcs, and \( T \) is the set of attributes to arcs in \( A \) such that,

- For every face \( f \) in \( F \), there exists a unique node \( n \) in \( N \).
- For every edge $e$ in $E$, there exists a unique arc $a$ in $A$, connecting the nodes $n_i$ and $n_j$, corresponding to face $f_i$ and $f_j$ which share the common edge $e$.

- Every arc $a$ in $A$ is assigned an attribute $t$, where $t = 0$, if the faces sharing the edge form concave angle, and $t = 1$, if the faces sharing the edge form a convex angle.

Figure 2.2 shows the example of the AAG for a part. The AAG is represented in the form of a matrix. The various features are recognized on the basis of certain specialized rules. These recognition rules are based on the properties of the AAG unique to each feature.
For example\textsuperscript{1}, refer to figure 2.3, which shows an AAG graph representing a pocket.

The properties of the AAG used to define a pocket are as follows:

- The graph is cyclic
- It has exactly one node $n$ with the number of incident ‘0’ arcs equal to (number of nodes $-1$).
- The number of ‘0’ arcs is greater than the number of ‘1’ arcs, after deleting the node $n$.

These properties of the pocket are general enough to identify a wide range of pockets.

![Diagram of AAG graph and 3D model](image)

**Figure 2.3** Recognizing a pocket feature using the AAG \[7\]

2.3 CSG Based Approach

In 1990, Perng et al. [8] proposed a method for automatically extracting the machining features from 3-D CSG (constructive solid geometry) solid input. In this method, the CSG tree representation of a part is first converted into an equivalent DSG (destructive solid geometry) tree representation. During the extraction of machinable features, the object surface information in terms of boundary representation is needed and is generated from the DSG data. From this information, the machinable volumes can then be extracted. The part domain for this method includes prismatic as well as cylindrical faces.

2.4 Neural Network Based Approach

In 1992, Prabhakar and Henderson [9] devised a technique for performing form-feature recognition using the principles of neural networks. Neural nets require parallel input of data, which in this case are B-rep solid models of parts. An input format was developed which includes face descriptions and face-face relationships. The approach of this technique can be summarized as follows. The solid model was coded in terms of certain essential parameters and characteristics followed by the construction of an adjacency matrix of the part. Then, the networks were constructed one for each feature type in the feature library and programmed using the feature definition language that was developed in this research. Each net was then fed with the adjacency matrix, one row at a time, and,
after the iteration is complete for one row, and the output nodes of each net were examined. If the output node was in the active condition, the feature faces were extracted. An example\(^1\) illustrating this mechanism is described below (refer to Figure 2.4).

The example block consists of two rectangular through slots intersecting each other at right angles. The solid modeler can construct this part in two ways merged and unmerged. When the faces are unmerged, the principal slot face (shaped as a plus) is represented as five faces, separated by the edges shown by the four lines in Figure 2.4(a).

---

The *merge* command unifies the five faces into one face, as seen in Figure 2.4(b). The recognizer gives two different results for the two cases. In the unmerged case, the algorithm recognizes four instances of a through slot with the principal faces being F14, F9, F6, and F3. In the merged case, only one slot is recognized and all the faces adjacent to the principal face are identified.

### 2.5 Design by Features

Design by features, or feature-based design, uses a library of 2D or 3D features as design primitives on the product modeling level [4]. Thus, the designer can work directly with a feature for example, a pocket rather than associating low-level primitives like the vertices or edges of the pocket. Features provide additional information that can be useful for cost estimation systems. Some of the feature based design approaches have been summarized in the following sections.

#### 2.5.1 Destructive Solid Geometry (DSG)

This approach uses a solid as the starting point. The user then instances and positions the features on the stock corresponding to the material to be removed from the stock. Thus, the cumulative volume of all these features represents the total material to be machined, and this data can be supplied to a cost estimating system.
2.5.2 Compositional Feature Models

In this approach, developed by Luby et al. [10], no base solid is required. The design procedure is carried out by adding, subtracting, and manipulating features. This method concentrates on creating a high-level representation of part-geometry, allowing manufacturability evaluation at the same time. To allow evaluation of manufacturability, the geometric database is designed to express the features in terms corresponding to available manufacturability knowledge and to represent the relationships or connectivity of these features.

For example, the part shown in figure 2.5 is broken up into an object hierarchy as shown in figure 2.6 and then the program for manufacturability evaluation is invoked. The software locates and outputs several problems along with recommendations of how each feature can be machined. The advantage of a features database is that it yields a symbolic representation of the design that can be used for manufacturability evaluation, process design and process planning.

---

Figure 2.5  Example part to illustrate the use of a features database in process planning and manufacturability evaluation [10]
Figure 2.6  Object hierarchy for part shown in Figure 2.5 (a) [10]
2.5.3 Procedural Feature Languages

As explained by Lin and Lin [4], this approach represents geometric objects in a hierarchy, as a specific procedure. The topmost object procedure invokes other object procedures until it gets to a primitive object. At this point there are no more procedures to be called. This method develops a sequence of operations that can be combined, stored and retrieved. Execution of these sequences produces a list of features, attributes, and variables. These features are created by CSG operation sequences.

2.6 Other Important Research Approaches

In 1993, Chamberlain et al. [11] did research work on taking a feature-based model as input and then reasoning on the model to discover interactive machinable-features. Their approach was an attempt to make the design environment closer to the design process and extract manufacturing features from the CAD model without applying inefficient feature recognition algorithms developed earlier. A prototype system was developed to test the concept of the framework of prismatic part data generation. The system consists of three modules:

- A CSG to DSG transformation module
- A 2-D CAD system interface module
- A unit-machined loop construction module.
The module transforms the CSG input provided by the user to DSG and derives the prisms and sectional cylinders. The second module reads 2-D CAD data from the IGES data file and performs the volume-removal identification and decomposition. The third module derives the unit-machined loop for each of the prisms and sectional cylinders based on defined rules. These rules are:

- Rule of the cutter: Here the corner, bottom, and the width of the decomposed prisms and sectional cylinders determine the size and type of cutter.
- Rule of machining cost: An attempt is made to find an optimization among the minimum number of unit-machined loops, set-up time, and cutters used.
- Rule of process experience: This rule depends on the process-planning experience of the engineer.

Dewhurst and Boothroyd [3] described product-costing procedures, which intend to form the basis for a design analysis method for product design for efficient manufacture. The suggested approach consists of two steps:

- Identification of the appropriate materials and manufacturing processes for the component part of a new product design.
- Detailed design of the individual components consistent with the capabilities and limitations of the material-process combinations.

This approach is primitive and aims at estimating the product cost based on certain assumptions such as ideal processing conditions and optimum investment in tools, dies and efficient operating conditions.
In 1994, Tseng and Joshi [12] researched on recognizing multiple interpretations in 2.5D machining of pockets. To integrate CAD and CAM of 2.5D prismatic machining parts, part descriptions in low-level format, such as B-rep, are interpreted into forms of high-level features, such as pockets and virtual pockets, to facilitate process planning and tool path generation. They developed a recognition procedure to transform low-level data into machinable pocket definition. The algorithm suggested in their research uses volume decomposition to decompose the total removal volume into small blocks of machinable volume. Then, based on the machining strategies of 2.5D pockets, the small blocks are reconnected in a systematic way to reconstruct pocket volumes and generate multiple interpretations. The algorithms developed in this research accomplish the following tasks:

- Recognition of high level pockets from low-level B-rep part descriptions.
- Transformation of complex depression volumes into individual machinable pockets for existing milling approaches.

A method for automatically extracting machinable volumes was proposed by Li and Tzieng [13] in 1991. This approach considered a non-blank type of raw material input. The method takes in CSG input (both raw material and finished part) and determines the relative shifting origins of the finished part to ensure its machinability. CSG to DSG transformation is then performed to extract the DSG primitives of the raw material and the finished part. Both DSG primitives are mapped to find the removal volumes based on geometrical definition. The geometrical definition removing volumes are further
reorganized into a set of machinable volumes by considering the influence of process information.

For example, consider a non-blank type of raw material from which the finished part is to be machined. In this method, the input CSG trees for both the raw part and the finished part are determined. The system first computes the minimized cover block dimensions of the raw material and the finished part. Next, the system determines the new relative original coordinate of the finished part by positioning it on the raw material. Then the system performs the mapping and combination process to extract the removing volumes based on their geometric definition. Finally, the geometric definition removal volumes are merged and split into machinable volumes by considering the influence of process information. The final result is expressed as a DSG tree.

In 1990, Li and Yu [15] mapped machinable volumes into four semantic groups: base features, pre-defined features, combined features and irregular features. Their research indicates that the removal volumes may require reorganization into machinable volumes to be mapped to one of the four types of features. Also, in the process of feature extraction, the influence of process information should be considered. They propose a framework for prismatic part-data generation based on the concept of the unit-machined loop. The unit machine loop is a removal volume that can be machined with a type of cutter within a type of operation. In this method, the CSG solid input is first transformed into a DSG tree. The primitives of the DSG tree are the removal volumes and are
classified as prisms and sectional cylinders. Thus, the removal volumes of the part are identified from the CAD database and decomposed into prisms and sectional cylinders. Using these primitives, the unit-machined loop is constructed along certain guidelines and rules that can provide valuable information necessary to compute the cost of manufacturing the part.

In 1997, Yang and Lin [1] proposed a framework for estimating the manufacturing cost in terms of a feature-based approach. This system tends to estimate the manufacturing cost of a design according to the shapes and precision of its features. The objective of their research is to provide a tool to assist a designer, who has little knowledge about the manufacturing processes, to estimate the fabrication cost of a design during the conceptual stage, to reduce unnecessary costs in the downstream process. Their approach integrates a feature-based CAD model and a database, storing product and process cost. This approach analyzes a part model designed in a feature-based CAD system according to a certain criteria to roughly estimate its required machining cost. In this method there are three modules: the CAD module, the reference library module and the analyzing module. The CAD module supports the feature-based part construction and modification function for the user to perform the design task. The part database not only stores the B-rep data of a part and it's features but also saves other data such as values of the parameters used in defining individual features and the precision requirements of each feature. The reference module comprises of reference data such as:

- available machines in the shop,
• kind of operation that can be performed by each machine,
  operating cost of the equipment,

• information about the required manufacturing processes to produce certain features,

• sequence of operation,

• data about individual features of a part, and,

• estimated manufacturing costs for each part and it’s features.

The most important module in the framework is the analysis module, which consists of two sub-modules. The feature geometry analysis module retrieves the feature-based part data from the CAD/CAM system. The cost analysis module uses this data to perform three functions:

• analysis of the manufacturability of the part,

• estimation of the required manufacturing time for each feature,

• computation of the manufacturing cost of the part.

The flow chart of the cost analysis process is as shown below in Figure 2.7.
In 1998, Alan Lin and Shou-Yee Lin [4] proposed a volume decomposition approach to process planning for prismatic parts with depression and protrusion design features. The volume decomposition approach partitions the whole machining volume, to be removed from the raw material into some machinable features. Once the machinable features are found, the machining of the part can be viewed as material removal for a series of machinable features. The tasks involved in their method were:

- Extraction of the machining volume from the part model.
- Decomposition of the machining volume into machinable features.
- Determination of procedures required for cutting out the machinable features.
2.7 Conclusion

The various methods and approaches discussed above do not extend beyond feature recognition and volume removal. From a manufacturing perspective, these methods cannot be used to gather information about other important aspects such as set-ups required to machine the part, diameter of the cutter to be used etc. Moreover, the feature recognition methods suggested in the past suffer from:

- The computational inefficiencies caused within the pattern matching procedure during the process of recognizing interacting features,
- The inability to handle features formed by the combination of cylindrical and planar faces.

Some of these methods cannot be extended to non-planar faces and are restricted to planar faces only. Also, the backward filling approaches that have been suggested in the past to find the machinable features and their volumes work only for specific features like protrusions and depressions and are incapable of handling features, such as blends and through holes. The backward filling method developed in this thesis not only encompasses more features in its feature recognition algorithm but also strives to collect information about manufacturing the part. The data gathered by implementing this approach, can be used as valuable input information to a cost estimation system.
Chapter 3

Backward Filling – Approach and Algorithm

3.1 The Approach

A backward filling approach has been adopted in this thesis and the method primarily comprises of developing the finished part into the raw stock. It requires a raw part in addition to the finished part in the CAD file. The algorithm operates on one face of the finished part at a time, until all the faces have been processed. Processing of a face consists of identifying the features existing on the face and filling the volume occupied by these features through recursive extrusion, intersection, and union operations. This eventually, builds the finished part into the raw part. A detailed explanation of the algorithm is provided further in this chapter.
3.2 Assumptions

The assumptions made prior to implementing the algorithm are listed here in this section.

- All faces are either planes or cylinders i.e. curved surfaces cannot be processed by the developed algorithm.
- A plane is considered machinable only if the faces adjacent to the edges of the plane are perpendicular to the plane.
- The set-ups assume a 3-axis machine.
- Problems occur when there are intersecting cylinders and non-prismatic geometry (geometry at oblique angles).
- All normal, tangent, and defining direction vectors are of unit length.

3.3 Recognition of Features and Associated Terminology

In the process of building the finished part into the raw stock, a number of features are recognized. The machining parameters for these features, such as tooling, feed, speed, material removal rate, etc., are different from one another and, therefore, the method used to fill each of these features is different from the other. This section, briefs on the various features that can be recognized by the algorithm developed in this thesis, the methodology utilized in filling up the volume occupied by these features, and the associated concepts and terminology. Using analytical geometry and vector arithmetic, the features recognized are classified into seven categories, as described further in this
chapter. Before understanding the feature recognition algorithm, it is imperative to gain knowledge about the various concepts that support the algorithm and the various terms that have been used in the implementation.

3.3.1 Inside and Outside Corners (Up-edges and Down-edges)

The feature identification algorithm is based on the nature of the edges that define a feature existing on a face and the relationship between the faces that share the edges. The corner formed at an edge between two faces are distinguished here as inside and outside corners. If an inside corner is formed at an edge then the edge is an up-edge and if an outside corner is formed at an edge the edge is a down-edge. When recognizing a feature, the corners formed by all the edges that define the feature are analyzed and depending on the type of corner found at each edge, a decision is made as to which feature they belong. Figures 3.1 and 3.2 show examples of an inside and outside corner respectively. Figure 3.3 describes the algorithm to identify inside and outside corners.
Defining direction of face A

Vector running from defining point of face A to an extreme point on face B

Edge in consideration

Figure 3.1 Example of inside corner formed at an edge

Vector running from defining point of face A to an extreme point on face C

Edge in consideration

Figure 3.2 Example of an Outside corner formed at an edge
The algorithm explained in Figure 3.3 is with reference to Figure 3.1.

Note: The term “extreme point” used in the algorithms is defined as the farthest point lying on an object in a particular direction.

- Determine the defining direction of face A
- CQL query – Find the extreme point on face B, in a reverse direction to the defining direction of face A
- Find the vector from the defining point of face A to the extreme point
- Determine the dot product of the newly found vector and the defining direction of face A
- If (dot product >= 0)
  Edge in consideration forms an inside corner
  Else
  Edge in consideration forms an outside corner.

Figure 3.3 Algorithm to find the inside and outside corners formed by edges

3.3.2 Inside and Outside Arcs

Round corners or blends are formed by the arc edges existing on the outside loop of a face. However, some of the arc edges present on the loop may form slots, for example, a U-slot. Therefore, while machining round corners, care should be taken to distinguish between the arc edges that form the round corners and those that form slots. To
accomplish this task, the concept of inside arcs and outside arcs has been introduced. A round corner always contains an outside arc while a slot always contains an inside arc. Figure 3.4 shows an example of an outside arc formed by the arc edge in consideration.

Figure 3.4 Example of an outside arc formed by a round corner
Figure 3.5 shows the algorithm to distinguish an inside arc from an outside arc.

- CQL query - Determine the normal to the arc edge at one of its vertex.
- Determine the vector running between this vertex and the center of the arc.
- Determine the dot product of this vector and the normal.
- If (dot product < 0)
  
  Arc edge forms an outside arc.

  Else

  Arc edge forms an inside arc.

Figure 3.5 Algorithm to distinguish between inside and outside arcs

3.3.3 Next Logical level

The 'next logical level' is defined as the logical limit to which an object can be extruded. It represents the upper limit or the end point of the extrusion operation. Figure 3.6 demonstrates the concept of next logical level and Figure 3.7 shows the algorithm to determine the next logical level.
Figure 3.6  Example demonstrating the concept of ‘Next Logical Level’

The Algorithm shown in Figure 3.7 is with reference to Figure 3.6
The algorithm described by Figure 3.7 determines the ‘Next logical level’ for extruding face A.

- Determine all the edges of the face A.
- Determine all the faces that are adjacent to these edges other than the face in consideration i.e. face B, face C, face D and face E
- Determine all the edges of these adjacent faces
- Find the extreme points on the edges in direction of the defining direction of face A greater than the start point of extrusion.
- Find the dot product of the defining direction of face A and each vertex.
- Lowest dot product = ‘Next logical level’.

Figure 3.7 Algorithm to determine the ‘Next Logical Level’

3.3.4 Loops or FaceLoops

Face Loops are user-defined objects (UDO). UDOs are a powerful way to define new kinds of entities with user specified behavior in Unigraphics. Face loop entities have been implemented in CQL. All the data in a Face Loop is an array of integers. The first element in the array is the object ID of the parent face. The next entry indicates whether the loop is an internal or external loop. This is followed by the object IDs of the edges that comprise the loop in order.
3.3.5 Features Recognized

The various features recognized are explained in brief below.

- **Depressions** are formed by faces where each edge on the outside loop of the face forms an inside corner. These kinds of features are filled up by extruding the entire face to the next logical level. A depression can be imagined as a blind hole and is different from a through hole.

- **Through Holes** are formed by the inside loop on a face where each edge of the inside loop forms an outside corner. Through holes are filled up by extruding the outside loop of the hole, in the direction opposite to the defining direction of the face, up to the 'next logical level' in the direction of extrusion.

- **Round Corners/Blends** are formed by faces that are cylindrical, with each edge of the face forming outside corners. These features are found only on the outside loop of a face. Round corners/Blends are squared off by drawing tangent lines to one of the arc edges of the face and then extruding these lines along with the arc edge till the other arc edge of the face.

- **Slots** After the round corners/blends are squared off, any arc edge present on the outside loop of a face indicates the present of a slot. When filling up the slots, the start and the end points of the slot are determined. These points are basically
vertices of the edges of the slot. These two points are then connected with one/two lines and the edges that represent the slot are extruded along with these lines in a direction opposite to that of the face containing that slot up to the next logical level in that direction.

- **Bosses/Protrusions** are formed by the inside loops of faces where each edge of the inside loop forms an inside corner. In case of machining bosses/protrusions, the area of the face around the protrusion is of more importance than the area covered by the protrusion itself. Thus, protrusions are machined by extruding the face on which the protrusion exists along with the inside loop that defines that protrusion. In case of multiple protrusions on a face, the face is extruded up to the top of that protrusion which has the lowest height.

- **Steps** are formed by faces that have a combination of edges forming inside and outside corners. These are one of the simplest features and are filled up by extruding the face in the defining direction of the face up to the next logical level.

- **Final Planes** are the faces that are devoid of all features mentioned above, except protrusions, with each edge of the outside loop of the face forming outside corners. These planes are machined by extruding them to the height of the raw stock in the direction of the face.
A summary of the algorithm for the backward filling method has been discussed in the following section.

3.4 Summary of the Algorithm

This section summarizes the algorithm for the backward filling method.

Note: Tasks 2 to 6 are performed recursively till the finished part is transformed into the raw part.

Task 1: Initialize

- Initialize a defining direction. This initialization is done arbitrarily and the initialized defining direction is called the base defining direction.

Task 2: Identify a machinable plane

- Find a machinable plane that is most closely aligned with the base direction and the update the base direction with the direction of this new plane.
- Find all the machinable planes in this new direction and select the lowest machinable plane.
- While iterating to find the machinable planes, mark all the planes that are not machinable.
Task 3:  **Process all through holes on the plane**

- Identify all the internal loops on the plane that represent through holes.
- Find the direction opposite to the defining direction of the face. This is the direction of extrusion.
- For each loop identified:
  - Find the limit of extrusion by calculating the next logical level.
  - Extrude the loop, in the direction of extrusion, till the next logical level
  - Intersect the extruded solid with the raw part
  - Unite the intersected solid with the finished part.

End For

**Task 4:  Process all the round corners/blends on the plane**

- Find the arc edges existing on the outer loop of the face
- Find the direction opposite to the defining direction of the face. This is the direction of extrusion.
- For each arc edge:
  - If arc edge forms an outside arc:
    - Construct lines from the each vertex of the arc, tangent to the arc
    - Find the limit of extrusion by calculating the next logical level.
    - Extrude the newly constructed lines along with the arc edge till the next logical level
    - Intersect extruded solid with the raw part
    - Unite the intersected solid with the finished part
End If

End For

Task 5: Process all slots on the plane

- Find all the arc edges on the plane. Since all the round corners have been processed at this stage, all the arc edges existing on the outer loop of the lane represent slots
- For each arc edge:
  - Find the boundary points of the slot. These are two points on the outer loop of the face, and all the edges that lie between these two points represent the slot.
  - Find the direction opposite to the defining direction of the face. This is the direction of extrusion
  - Connect the two boundary points by constructing line(s).
  - Determine the next logical level for extrusion.
  - Extrude the newly constructed lines along with the edges lying between the two boundary points till the next logical level in the direction of extrusion.
  - Intersect the extruded solid with the raw part
  - Unite the intersected solid with the finished part

End For
Task 6: Extrude the plane

- Identify any inside loops existing on the plane. All inside loops represent protrusions at this stage.
- Find the next logical level for extruding the plane, considering the height of the protrusions also.
- Extrude the plane up to the next logical level.
- Intersect the extruded solid with the raw part
- Unite the intersected solid with the finished part

To demonstrate the back ward filling mechanism, a simple example has been provided in the next section.

3.5 A Simple Example

The part used in this example is a rectangular block consisting of at least one of each feature recognized by the algorithm. Figure 3.8 shows the finished part along with the raw part designed in Unigraphics. The raw part exists on a different layer than the finished part in the part file. The raw part has been represented by dotted lines while solid lines represent the finished part, in the figure. Figure 3.9 shows all the solid objects corresponding to the features machined from the raw part to produce the finished part. Figures 3.10 and 3.11 display the two output files, Features and setUps created for machining the part.
Figure 3.8   Example of a simple part to be processed
Figure 3.9  Solid objects corresponding to the various machined features in part shown in figure 3.8
The labels provided on each solid correspond to the features they represent. Although, all the features machined are not shown in Figure 3.9, an elaborate idea about the backward filling technique is provided by the example. The solid objects shown in figure 3.8, when united with the finished part transform the finished part into the raw part. Figure 3.10 shows the Features file while Figure 3.11 shows the setUps created after the machining is complete.

Notes:

1. Solid Object tag corresponds to the object id of the solid obtained after the intersection with the raw part.

2. Key Diameter on the feature represents the diameter of the cutter used to machine the particular feature.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Volume removed</th>
<th>Solid Object tag</th>
<th>Depth Machined</th>
<th>Area of Surface Machined</th>
<th>Perimeter of Surface Machined</th>
<th>Complexity of Machining</th>
<th>Key Diameter on the Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>POCKET</td>
<td>0.94634954 cu.inch</td>
<td>1354</td>
<td>1.000000 inch</td>
<td>0.946350 sq.inch</td>
<td>3.570796 inch</td>
<td>1</td>
<td>0.500000 inch</td>
</tr>
<tr>
<td>STEP</td>
<td>0.32400000 cu.inch</td>
<td>1866</td>
<td>0.400000 inch</td>
<td>0.810000 sq.inch</td>
<td>3.600000 inch</td>
<td>1</td>
<td>0.360000 inch</td>
</tr>
</tbody>
</table>

Figure 3.10  Example of the Features file (all features not included)
<table>
<thead>
<tr>
<th>Set Up</th>
<th>Direction</th>
<th>Start Feature</th>
<th>Number of Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0, 0, 1)</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>(0, -1, 0)</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(-1, 0, 0)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>(0, 1, 0)</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(0, 0, -1)</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(1, 0, 0)</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.11 Example of the setup file
Chapter 4

Implementation

This chapter deals with the inner workings of the backward filling method. Having explored the execution of the code, that implements the method, this is an appropriate point at which the actual mechanisms developed in CQL and TNR can be discussed. The chapter explains the working of operations, such as ‘extrude’, ‘intersect’, ‘union’, ‘create line’ and ‘export’, which have been developed in CQL. These mechanisms form the foundation for the backward filling method. Next, the routines developed in TNR to implement the method are discussed with detailed explanations wherever necessary.

4.1 The Unigraphics API (Application Programmer’s Interface)

An introduction to the Unigraphics API is important before acquiring knowledge about how it can be used to develop mechanisms like extrude, intersect, union, create line and export. The Unigraphics API or UG API is a set of C and FORTRAN routines that provides access to part file data. CQL has been developed by implementing routines from the API to perform some of the data access and manipulation tasks via a querying
interface. Detailed information on the use of these routines can be found in the online Unigraphics documentation under the section ‘Languages’. All the operations, such as extrusion, intersection, union, creation of a line, and export, need to invoke API functions. All UG API function names begin with the prefix ‘UF_’. The API functions used to develop the operators mentioned above are:

- UF_MODL_create_extrusion
- UF_MODL_operations
- UF_CURVE_create_line
- UF_PART_export_with_options

These API functions have been discussed, along with the routines developed in CQL, in the following section. Unlike other API functions used in the implementation of CQL, these functions are applied for part data manipulation which can change the structure of the part.

4.2 Extension to CQL

4.2.1 Extrusion

The extrusion operation is the most important tool used in the backward filling method. CQL allows the extrusion of three types of objects: faces, edges and face loops. When issuing a CQL extrude query, one or more objects can be specified as the target objects. The object ID's are always specified as aggregates i.e. the object ID's are always
enclosed in braces ( ). If there is more than one object to be extruded in a single CQL statement then the object ID’s of the various objects are separated by commas. The syntax for issuing an “extrude” statement is shown below.

Syntax for extrude statement:

[along clause] [by clause] [extrude clause] [from clause]
[optional where];

- **along clause**: The direction in which the extrusion operation should take place is specified in the along clause. It begins with the keyword ‘along’ and is followed by a vector that specifies the direction. For example,

  along (0,0,1)

- **by clause**: The distance through which the extrusion operation takes place is specified in the by clause. The “by” clause begins with the keyword ‘by’ and is followed by a number that indicates the extrusion distance. For example, **by 2.5**

- **extrude clause**: This clause simply specifies that an extrusion operation is to be performed on the objects retrieved by the execution of the ‘from’ clause and optionally filtered by the ‘where’ clause. The valid objects that qualify for the ‘extrude’ operation are faces, edges and lines. A syntax error is reported if an extrusion operation is tried on an invalid object.
• **From clause:** The from-clause forms the most important part of queries. It establishes a working set of data as the set of all object IDs of the particular entity in the CAD part file. The variable declared in the query iterates through every instance ID from this working set of data.

• **Where clause:** This clause is optional and is used only to restrict the set of data formed by the “from” clause. Based on the condition specified by the where clause, objects are selected from the working set for the extrusion operation. Refer to example 2 shown below.

Some examples of extrude queries are as shown below.

**Example 1:** along (0,0,1) by 1.5 extrude x from x in (248, 542);

The object ID’s specified in the braces in the above example may be objects of the same type, or a combination of the different objects. For example, the object ID’s 248 and 542 may represent two faces, or they may represent a face and a face loop respectively.

**Example 2:** along (0,0,1) by 2.5 extrude x from x in Face

where x->name == "TOP";
In this example, the “from” clause constructs the set of all the faces in the part file. The where clause chooses the face whose name is “TOP” from this working set. This face is extruded through a distance of 2.5 in the (0,0,1) direction.

The function _cql_perform_extrude_ug is called to perform the extrusion operation on the objects specified in the query. The extrusion parameters such as the direction of extrusion and limit of extrusion as well as object ID’s are passed in to the function as two separate linked lists of nodes, where each node contains one value. The type of each object is determined before they undergo extrusion. This procedure is of importance because the extrusion operation supports ‘FaceLoop’ type of objects. Face loops are user-defined objects (UDO’s). Unigraphics provides a powerful way of defining new kinds of entities with user specified behavior. Face Loop entities have been implemented in CQL using the concept of UDO’s. In any case, the extrusion operation in Unigraphics accepts only edges and faces as valid entities. Therefore, it is necessary to obtain the object ID’s of the edges that define a FaceLoop. The type of object is determined by calling the _cql_get_cql_type function. If the object is a FaceLoop, then the edges of that FaceLoop are obtained by calling the function UF_UDOBJ_ask_udo_data. The call to the actual extrusion function (API function), UF_MODL_create_extrusion is made after all the objects, valid for extrusion, are obtained. After executing the query, CQL displays the object ID’s of the objects that were extruded.
4.2.2 Boolean Operations – Intersection and Union

In the backward filling method, the solid obtained after extrusion is intersected with the raw part. The intersection operation is necessary because, in many cases the extruded solid extends beyond the boundaries of the raw part. The solid obtained after the intersection operation represents the accurate amount of material to be removed from the raw part to machine a particular feature. After the intersection operation, the solid obtained as a result of intersection is united with the finished part, thus building the finished part into the raw part. The union operator accomplishes the task of uniting two solids. Intersection and union are classifications of Boolean operations. It is imperative that Boolean operations are performed only on solid objects. The syntax for an “intersect” statement is explained below.

Syntax for intersect statement:

```
[intersect clause] [from clause] [optional where clause];
```

- **Intersect clause**: This clause specifies that an intersection operation is to be performed on the objects retrieved by the execution of the ‘from’ clause and optionally filtered by the ‘where’ clause. The valid objects that qualify for the ‘intersect’ operation are solids. A syntax error is reported if an intersection operation is tried on invalid objects.
• **From clause:** The from clause in the intersect statement, exclusively, forms a set of solids which participate in the intersection operation. The typical format of a from clause in an intersect statement is as follows:

[from keyword] [target solid][,] [tool solids]

The first solid object ID following the 'from' keyword must specify the target solid. There can be any number of tool solids but only one target solid in an intersection operation. It is important that some portion of each tool solid lies inside the target solid for the intersection operation to take place successfully.

The example of the intersect query is as shown.

**Example:** intersect x, y from x in (123), y in (456);

The object ID's, 123 and 456, specified in the query represent solid objects. The solid object that references the variable ‘x’ is the target solid while the solid object that references the variable y is the tool solid. There can be one or many tool solids, but only one target solid. Each variable used in the query can be referenced by one and only one solid object. For example, if two tool solids were to be intersected with the target solid then the query would be framed as follows:

```
intersect x, y, z from x in (123), y in (456), z in (789);
```

The syntax for a “union” statement is similar to the intersect statement except for the keyword “union”.

Syntax for union statement:
[union clause] [from clause] [optional where clause];

- **Union clause:** This clause specifies that a union operation is to be performed on the objects retrieved by the execution of the ‘from’ clause and optionally filtered by the ‘where’ clause. The valid objects that qualify for the ‘union’ operation are solids. Again, a syntax error is reported if a union operation is performed on invalid objects.

- **From clause:** Similar to the intersect statement, the from clause in the union statement forms a set of solids which participate in the union operation. The typical format of a from clause in an union statement is as follows:

  [from keyword] [target solid][,] [tool solids]

  The first solid object ID following the ‘from’ keyword must specify the target solid. There can be any number of tool solids but only one target solid in a union operation. It is important that some portion of each tool solid lies inside the target solid for the union operation to take place successfully.

An example of a union query is as shown:

union x, y from x in (486), y in (567);

In the above example, 486 and 567 are the object IDs of the target solid and tool solid respectively. Similar to the intersect statement, if there are more than one tool solids the union statement is issued as shown below:

union x, y, z from x in (486), y in (567), z in (123);
In this example, 567 and 123 represent tool solids. These operations are performed by calling the \_cql\_perform\_boolean\_operation\_ug function. The function accepts two parameters. The first parameter specifies the type of operation to be performed i.e. intersection, union, and subtraction. The second parameter is a linked list of nodes, where each node contains one object ID. Given that the CQL statement is framed in the correct manner, the first node always holds the object ID of the target solid followed by other nodes containing the object ID's of the tool solids. The Boolean operation is performed by calling the API function UF\_MODL\_operations. After execution, CQL displays the object ID's of all the solids that participated in the operation.

4.2.3 Creation of a line

The 'create line' statement in CQL was developed with a view to machining slots and round corners or blends. As explained in chapter 4, a slot or a round corner is squared off by constructing lines between points that belong to the slot or the round corner. Therefore, the create line statement developed in CQL is restricted to one and only one format i.e. creating a line between two points. The syntax of issuing this statement is shown below:

Syntax for a “create line” statement:

\[\text{[create line clause] [between clause] [and clause]};\]
• **create line clause**: This clause specifies that a line is to be created between two points specified in the ‘between’ clause and the ‘and’ clause.

• **between clause**: This clause begins with the key word ‘between’ followed by a vector of numbers which specifies the first point of the line. For example,

  ```plaintext
  between (2.5, 3, 1.5)
  ```

• **and clause**: this clause begins with the key word ‘and’ and is followed by a vector of numbers which specifies the second point of the line. For example,

  ```plaintext
  and (-2.5, 4, 1)
  ```

An example of a create line statement is as shown below:

```plaintext
create line between (0,0,2) and (0,0,4);
```

On execution, a line is constructed between the two points specified in the ‘between’ and the ‘and’ clause. The function _cql_create_line_ug is called to perform this operation. The function takes in a linked list of two nodes, where each node contains a point. These nodes store information about a point as an aggregate value. Inside the function, each of these aggregates is further broken down to access the actual x, y, and z values of the two points. Next, the UG API function UF_CURVE_create_line is called to construct the line in the part file. This function returns the object ID of the newly created line. After execution, CQL displays the object ID assigned to the line after it is constructed.
4.2.4 Exporting an Object

The result of the intersection operation between the raw part and the extruded solid is another solid object (intersected solid), which should be united with the finished part in order to build the finished part into the raw part. At the same time, the intersected solid needs to copied on to a different layer, so that it can be displayed as a representation of the material removed to machine the particular feature. This is achieved by the export mechanism provided by UG. The export mechanism makes a copy of the object to be exported. The copy remains on the current layer and the original object is exported to its destination. The copy of the solid object is then united with the finished part and the original object is displayed on a different layer. The export mechanism in UG can be used to export an object to a layer in different part file or a layer in the same part file. The destination file may be a new file. In this thesis, the export statement in CQL is developed to export an object to an existing part file only. The function _cql_perform_export_ug is called to invoke the export mechanism. The objects to be exported along with the destination part file and layer are passed as parameters into this function.

The syntax for issuing a ‘export’ statement is shown.

[to clause] [export clause] [from clause];

- **To clause:** The ‘to’ clause can be divided into two parts. The first part begins with a key word ‘to’ followed by the name of the part file to which the object is to be
exported. The second part of this clause begins with the key word layer followed by a number indicating the number of the layer in this part file to which the object is to be exported. For example,

```
Part 1 | Part 2
      |       
to test_part layer 2
```

In part 1 of the above example, test_part is the name of the part file to which the object will be exported. In part 2, the number 2 indicates the number of the layer of the part file to which the object will be exported.

- **Export clause**: This clause specifies that an export operation is to be performed on the objects retrieved by the execution of the 'from' clause. The export operation can be performed on any kind of object validly defined in Unigraphics.

An example of the export statement is as shown below:

```
to test_part layer 2 export x from x in (248);
```

In the above example, 'test_part' is the name of the part file to which the object is exported and the integer 2 indicates the layer in 'test_part' where the object will be stored. The UG API function called to perform this operation in UG is UF_PART_export. After execution, CQL displays the ID' of the objects that were exported.
4.3 Routines developed in TNR

The TNR code consists of 4 files, namely, main.tnr, classify.tnr, fill.tnr and support.tnr. These files can be found in the ~astra/thesis/code directory. In the following subsections, the routines developed in these files, which implement the backward filling mechanism, are explained with detailed description where necessary.

4.3.1 Preliminary tasks and Output Routines – main.tnr

This is the main file, which starts the execution of the code. The other files are included in this file using the ‘include’ pre-processor directive and linked during execution. The task of finding a machinable plane and then forwarding this plane for machining is accomplished here. But before any operation can be performed with part files, a license must be obtained from the license manager daemon. All these tasks are accomplished by the routines in this file. These routines have been discussed with detailed explanations where necessary.

- **main** This routine is discussed in detail in Section B.1 in Appendix B.
- **findDirectionAndPlane** This function is explained in Section B.2 in Appendix B.
- **processPlane** This function is explained in Section B.3 in Appendix B.
- **outputSetUps** This function prints the set-up information, stored in the set-up data structure, to the output file ‘setUps’.
- **outputFeatures** This routine prints the information about all the features machined, stored in the features data structure, to the output file, ‘Features’.

### 4.3.2 Feature Classification Routines – classify.tnr

The planes deemed machinable by `findDirectionAndPlane` may not be machinable even though they are planar. This is because of the existence of other non-machinable planes adjacent to one or more of its edges. Therefore, a more advanced filtering needs to be done at this point. The routines written in this file accomplish this task and are also used to identify the feature defined by a plane. Usually, a plane considered to be ‘not machinable’ at this point is stored in the set of temporarily marked planes. These planes are examined again for machining at a later stage in the algorithm.

- **classifyLoop** The algorithm for this routine is discussed in Section B.4 in Appendix B.
- **classifyEdge** This routine is discussed in Section B.5 of Appendix B.

### 4.3.3 Backward Filling Operation Routines – fill.tnr

This file consists of all the routines that invoke the mechanisms in Unigraphics to transform the finished part into the raw part. The sequence of operations, as described in Chapter 3, starts off by filling all the inside loops existing on the face. This implies the
machining of all the through holes because the face being machined is the lowest face in the selected direction. Next, the round corners are filled up followed by all the slots. The slots present on the face must be through slots also. Finally the face is projected up to the next logical level. Each of the routines in this file is discussed in detail below.

- **fillInLoops** This working of this routine has been explored in detail in Section B.6 in Appendix B.

- **fillInLoop** This routine is discussed in Section B.7 of Appendix B.

- **fillCorners** All the round corners or blends existing on the face are recognized and sent for processing in this routine. Round corners, as the name suggests, are defined by arc edges present on the outside loop of the face. Care should be taken to distinguish the arc edges that may be a part of slots. The arc edges that form a round corner are ‘outside arcs’ while the arc edges that form a slot are ‘inside arcs’. A detailed explanation of this routine is provided in Section B.8 of Appendix B.

- **fillCorner** To fill up round corners, lines are constructed, tangent to the arc edge, to enclose it. These lines, along with the arc edge, are extruded to complete the processing of the corner. This function has been discussed in Section B.9 of Appendix B.

- **fillSlots** After machining the round corners, all the arc edges existing on the outside loop of a face must represent only slots. This routine has been discussed in Section B.10 of Appendix B.
• **fillSlot** This function is discussed in Section B.11 of Appendix B.

• **project_up** This function is discussed in Section B.14 of Appendix B. If the face represents a final plane then the feature machined is entered into the Features data structure as ‘PLANE’. If the face is a part of a step, then the feature machined is listed as ‘STEP’. In case of a depression, the edges on the outer loop of the face are found. If there exists one and only one edge on the outer loop (circular edge), then the feature machined is a ‘ROUND_POCKET’, else it is listed as ‘POCKET’. In all the above cases, if a protrusion exists on the face, then the protrusion is machined first and the feature machined is listed as ‘PROTRUSION’.

• **extrudeAndCalculateVolume** This routine is explained in Section B.15 of Appendix B.

### 4.3.4 Support Routines – support.tnr

This file consists of all the routines that support the operations performed in implementing the backward filling method. In this section, these support routines have been discussed with detailed explanation wherever necessary.

• **latestSolidForFeature** This function finds the solid, associated with a particular feature, which was created most recently. This routine has been developed keeping in mind the recursive extrusion and intersection operations performed in
the process of machining an entire part. In order to obtain the volume of material machined, it is necessary to obtain the object ID of the solid obtained after the intersection operation. All the features in Unigraphics are listed as follows:

FeatureName (time stamp)

For example, an intersection feature would be listed as INTERSECTED(29). Each of the solid objects created in Unigraphics is associated with a particular feature, with a unique time stamp. Thus, there can be a number of solids associated with the same type of feature, but different time stamps. This time stamp can be used to distinguish one solid from another. Therefore, this function is called immediately after the intersection operation to obtain the object ID of the solid that was created. Based on the time stamp, the function examines all the solids associated with the ‘INTERSECTED’ feature and returns the ID of the solid with the most recent time stamp.

- cross This routine determines the cross product of two vectors. A flag indicating whether a unit vector is required is passed as a parameter into the function.
- dot This routine determines the dot product of two vectors. A flag indicating whether a unit vector is required is passed as a parameter into the function.
- break_vector This routine takes in a set of comma separated values, splits the set into an array holding all the values and returns the array
- equal This routine sets one array equal to another i.e. assigns each element of one array and to the corresponding element of the other. The two arrays must be of equal size.
• **updateSetUps** This routine is called every time a plane is processed. It updates the set-up data structure every time the direction of the set-up changes and populates the data structure with the direction of the set-up, the number of features machined in each set-up and the start feature of each set-up.

• **getVector** This routine determines a vector running between two points. A flag indicating whether a unit vector is required is passed as a parameter into the function.

• **getProjection** This function determines the projection of one vector on another.

• **isPlanar** This routine determines whether a face is a planar face. The criteria to decide whether the face is planar is that the defining direction of the face and the normal at a point on the face should be in the same direction. The dot product of the defining direction and the normal is determined. If the dot product is equal to 1 then the plane is deemed planar else, non-planar.

• **abs** This routine returns the absolute value of a number passed as a parameter.

• **findMinHeightFromLoop** This routine is discussed in section B.16 of Appendix B.

• **extendCorner** This routine is explored in detail in section B.12 of Appendix B.

• **connectExtension** This routine performs the task of connecting the boundary points of a slot, found by calling the function `extendCorner`, by constructing line(s) between them. The algorithm for this function is discussed in section B.13 of Appendix B.
• **break_vertices** This routine splits a string of comma separated values, corresponding the X, Y, Z coordinates of two points into two arrays of three elements, each array representing a point. In CQL, when a query is issued to get the vertices of an edge, the query returns with a string of six values corresponding to the X, Y and Z coordinates of the two end points of the edge. This result is an aggregate value in the form \((x_1, y_1, z_1, x_2, y_2, z_2)\) and needs to be broken down into two separate arrays, each representing one end point. The two end points, stored in two different arrays can then be utilized for further operations.

• **getDirectionsForExtreme** In order to determine an extreme point lying on a face or an edge in a particular direction, Unigraphics needs the other two directions perpendicular to the specified direction. This function returns the two perpendicular directions given one direction. These three directions are fed to the CQL query to find the extreme point on the face or edge. The structure of this query is as shown below: 

\[
\text{[select clause]} \text{[direction 1][direction2][direction3]} \\
\text{[from clause]}
\]

The query returns the extreme point on the face or edge in the direction specified by direction 1.

Example:

\[
\text{select x->extremepoint (0,0,1)(0,1,0)(1,0,0) from x in Face;}
\]

This query returns the extreme point on all the faces in the direction \((0,0,1)\).

• **findMaxHeightFromRaw** This routine finds the maximum height or the limit to which the raw part extends in a given direction. This is accomplished by finding
the extreme point on the raw part in the given direction. The dot product of the extreme point and the given direction gives the maximum height of the raw part in that direction. This value is used to extrude a final plane in the finished part up to the raw part.

- **minLoopDia** This routine returns the minimum diameter among all the arc edges of a loop. This minimum diameter indicates the diameter of the cutter to be used to machine a particular feature. If there are no arc edges present in the loop, it returns a zero.

- **e_length** This function takes in an array of edges as a parameter and calculates the sum of the lengths of all the edges. It is invoked to calculate the perimeter of the surface enclosed by the edges.

### 4.4 Input and User Interaction

The system developed in this thesis is implemented on the Silicon Graphics (IRIX 6.5) platform for the Unigraphics (V 15.0) CAD system. The role of the user is to design the part in the Unigraphics CAD system. The input to the TNR program is the name of the part file. Through CQL, TNR opens the part file and begins execution. In addition to the finished part design, the designer inputs a raw stock designed in the same part file as the finished part. It is imperative that the raw part is larger than the finished part in all dimensions. Once the part file is ready for input, the user ensures that the part file exists
in the same directory as the code. The program creates a copy of the part file called buildfile.prt and makes this the working file.

The general format for executing a TNR program is as follows:

```
tnr -f progfile [-v vl] .... [-vN]
```

Each of the above constructs are explained below:

- **tnr**
  
  Command issued to execute a TNR program.

- **-f progfile**
  
  TNR reads the program from `progfile` instead of stdin. `progfile` is the name of the TNR file to be executed.

- **-v vl**
  
  This construct passes `vl` to the parameter of the main function. If this option is used more than once in the command line, the assignments take place sequentially and stop upon running out of either arguments or parameters, whichever happens first.

To start execution of the backward filling application, the following command should be typed in at the command prompt:

```
tnr -f main.tnr -v partfile
```
In the above command, `main.tnr` corresponds to the name of the TNR program file and `partfile` corresponds to the name of the part file in which the designed part exists. The program takes in the name of the part file as an input parameter, opens the file and begins execution.

### 4.5 Output

The code creates three output files. The first file is a part file with a `.prt` extension named `buildfile.prt`. This part file contains objects stored on three different layers. The first layer stores the finished part built into the raw part, the second layer stores a copy of the raw part, and the third and most important layer stores all the solid objects that are created as a result of the backward filling process. Each of these solid objects represents the volume removed to machine a feature, present in the original finished part. This file is created at the start of execution. The code creates a copy of the original part file and uses this file as a working copy. The build file exists in the same directory as the original part file.

The second file, called `Features` contains the features machined along with all the data related to each feature. Similar to `Features`, a third file named `setUps` is created which holds information of the set-ups required for machining the part.
4.6 Data Structures Created for Output

Two arrays, features and setups are populated every time a feature is machined. These data structures hold information that contributes towards calculating the cost estimate for manufacturing the finished part. The features array is a one-dimensional array, with each feature assigned eight spaces in the array, corresponding to the eight fields required to describe a feature. Similarly, in the setUps array, each set-up is assigned four spaces corresponding to the four fields required to describe a set-up. These data structures, along with their respective fields will be discussed in the following sub-sections.

4.6.1 Features[]

The various fields included for the description of one feature are discussed below.

- **type**: This indicates the type of feature recognized. The various types of features identified and listed are pocket, hole, round pocket, round hole, round corner, slot, step or plane.
- **volume**: This represents the volume of material removed to machine the feature specified by ‘type’.
- **solid Object tag**: This corresponds to the object ID of the solid object obtained after filling up the particular feature.
- **area**: This represents the area of the bottom surface of the feature machined.
• **perimeter**: This corresponds to the perimeter of the bottom surface of the feature machined.

• **complexity**: This factor indicates an obstruction encountered while machining. An obstruction occurs when there are interacting features. This factor takes a value of 2.0 if there is an obstruction, else, it takes a value of 1.0.

• **key diameter**: This value suggests the diameter of the cutter to be used when machining the feature. It is calculated in a different manner for different features. For example, if the feature is a through hole, then the diameter of the hole becomes the key diameter to machine that hole. If the feature is a pocket, then the key diameter is twice the radius of the smallest arc edge found on the bottom of the pocket.

### 4.6.2 setUps []

The setUps array holds information of the all the set-ups required to machine the part. Each set-up comprises of four fields as described below.

• **set-up Number**: This indicates the number of the set-up. The significance of this number is realized when mapping a particular feature to the set-up in which the feature was machined.

• **direction**: This indicates the direction of machining for the particular set-up.
- **start feature:** This corresponds to the first feature machined in the set-up. This value when used along with the set-up number can be used to map the set-up with the features listed in the *Features* file.

- **number of features:** This value indicates the number of features machined in the set-up.
Chapter 5

Demonstrations

The demonstrations shown in this section are parts that were used to test and validate the code for implementing the backward filling method. In total, nine parts have been processed. Of particular importance is part9.prt because, the backward filling method fails. The reason behind the failure is the complex geometry involved in the part. For example, cylinders intersecting other cylinders. For each part, a sequence of figures is shown to provide a general idea of the working of the backward filling method. A complete step-by-step transformation of the finished part into the raw part is illustrated for part 1 and part 2. The sequence of illustrations shown for the other parts does not necessarily indicate the order in which the features are actually filled by the algorithm, but provide a brief idea about the working of the backward filling method.

5.1 Part 1

The sequence of illustrations shown demonstrate the transformation of the finished part shown in figure 5.1 (a) into the raw part shown in figure 5.1 (b). Each sequence indicates
a change in perspective and is further divided into phases so that the filling procedure for each feature can be demonstrated clearly.

5.1.1 Phase 1 of processing

Figure 5.2 (a) illustrates the finished part before the filling operations begin. In Figure 5.2 (b), the through hole is filled and the plane on which the through hole exists is extruded to the next logical level. The extrusion process stops here because the slot containing the arc edge, shown in Figure 5.2 (b), must be processed before extruding the plane any further.
Figure 5.3 (a) illustrates the part at the beginning of phase 2 of the filling operation. In Figure 5.3 (b), slot 1 is filled. In Figure 5.3 (c) slot 2 is processed. Finally in Figure 5.3 (d) the round pocket is filled and the plane (in the form of a ‘T’) is extruded till the boundary of the raw part. This marks the end of the filling operation in this particular direction.

5.1.2 Phase 2 of processing

Figure 5.3 illustrates phase 2 of the filling operation for part 2.
Figure 5.3  Phase 2 of filling operation for part 1
5.1.3 Phase 3 of processing

Figure 5.4 illustrates phase 3 of the filling operation for part 1.

Figure 5.4 Phase 3 of filling operation for part 1
Figure 5.4(a) shows the part at the start of phase 3. In this phase the two blends are processed and the plane, on which the blends exist are extruded to the next logical level as shown by Figures 5.4 (b), (c) and (d).

5.1.4 Phase 4 of Processing

Figure 5.5 illustrates phase 5 of processing part 1.
In phase 4, the protrusion is processed first, by extruding the face on which it exists. This is illustrated by Figure 5.5 (b). Next the hole existing on the protrusion is filled. Finally, the final plane (in the form of a ‘T’) is extruded up to the height of the raw part as shown in Figure 5.5 (c).

5.1.5 Phase 5 of processing

As shown in phase 1 (refer to Figure 5.2 (b)), the extrusion process stops at the arc edge because the slot containing the arc edge should be processed before extruding the plane. Since the slot is processed in phase 2, the plane can now be extruded. After this, all remaining planes are final planes and are extruded up to the boundaries of the raw part, thus completing the final stage of transformation of the finished part into the raw part. Figure 5.6 (a) shows the part at the start of the final sequence of processing. Figure 5.6 (b) shows the plane, left un-processed in sequence 1, extruded up to the next level. Finally, Figure 5.6 (c) shows the finished part transformed into the raw part.
Figure 5.6 Final phase (phase 5) of filling operation for part 1
5.2 Part 2

Figure 5.7 (a) shows part 2 and Figure 5.7 (b) shows the raw stock. It can be imagined that the raw stock completely encloses the finished part. Illustrations of the transformation of part 2 into the raw stock have been shown in sequences in this section.
5.2.1 Phase 1 of processing

At the start of execution, the base direction is always initialized to (0,0,1). In part 2, as seen in Figure 5.8 (a), the first plane found has a protrusion existing on it. This protrusion cannot be processed before the blend on the protrusion is processed. Therefore, the algorithm stores the plane in a set of temporarily marked faces. These temporarily marked faces are processed at a later stage. The base direction is changed at this point and the faces belonging to the protrusion are processed first. By processing these faces first, the blends on the protrusion are filled and the plane stored in the temporary set becomes eligible for processing. As seen in Figure 5.8 (b), the face belonging to the protrusion is processed and the through hole on the face is filled.

![Figure 5.8 Phase 1 of filling operation for part 2](image)
5.2.2 Phase 2 of processing

Figure 5.9 illustrates phase 2 of the filling operation for part 2. In this sequence, the features existing on the outer loop of the face belonging to the protrusion are processed.

Figure 5.9 (b) shows the part after processing the round corner. Figures 5.9 (c) and 5.9 (d)
show the part after processing blend 1 and blend 2 respectively. Here, the two blends are processed as slots because they satisfy the criteria for recognizing slots by forming inside arcs.

5.2.3 Phase 3 of processing

In this phase, the planes existing on the finished part are extruded to the extent of the raw part transforming the finished part into the raw part. The sequence of figures illustrating this final transformation is shown on the next page. Figure 5.10 (c) shows the finished part transformed into the raw part.
Figure 5.10 Phase 3 of filling operation for part 2
After exploring the backward filling operation for part 1 and part 2, the transformation of the other parts processed by the method into their respective raw parts can be visualized. Nevertheless, sections 5.3 to 5.8 demonstrate the filling mechanism on the other parts by illustrating the processing of some of the prominent features existing on them.

5.3 Part 3

Figure 5.11 illustrates part 3 before the filling operation with some prominent features.

Figure 5.11 Part 3
Figure 5.12 illustrates part 3 after processing round pocket 1 and Figure 5.13 shows part 3 after processing round pocket 1 and pocket 2.

Figure 5.12  Part 3 after processing Round pocket 1

Figure 5.13  Part 3 after processing pocket with protrusion
Figure 5.14 illustrates the part after processing the protrusion. In this case the area around the protrusion is filled up to the height of the protrusion. Figure 5.15 shows the part after processing all the round pockets.

Figure 5.14  Part 3 after processing protrusion

Figure 5.15  Part 3 after processing all remaining round pockets
Figure 5.16 shows the part after processing the simple pocket and the step. Figure 5.17 shows the last phase of processing for part 3. The mechanism stops processing any further because of the cones present on the part and thus the processing of part 5 remains incomplete.

Figure 5.16  Part 3 after processing the simple pocket and the step.

Figure 5.17  Final phase of machining of part 3
5.4 Part 4

As seen in figure 5.18, the prominent features existing in part 4 are slots and steps. It should be noted that the though hole should be machined before step 2 in the actual machining operation. Figures 5.19 through 5.24 shown in this section illustrate the processing of all the features named in Figure 5.18.

![Diagram of Part 4](image)

**Figure 5.18 Part 4**
Figure 5.19  Part 4 after processing step 1

Figure 5.20  Part 4 after processing step 2
Figure 5. 21 Part 4 after processing Slot 1

Figure 5. 22 Part 4 after processing slot 1
Figure 5.23  Part 4 after processing slot 2 and slot 3

Figure 5.24  Part 4 transformed into raw stock
5.5 Part 5

Figure 5.25 shows part 5 before the filling operation with some prominent features.

Figure 5.25 Part 5

Figures 5.26 through 5.31 shown in this section illustrate the filling operation on the features named in Figure 5.25.
Figure 5. 26  Part 5 after processing pocket with through hole and protrusion.

Figure 5. 27  Part 5 after processing pocket 2
Figure 5. 28  Part 5 after processing pocket 4

Figure 5. 29  Part 5 after processing rectangular pad 1 and simple hole
Figure 5.30  Part 5 after processing rectangular pad 2

Figure 5.31  Part 5 transformed into raw stock
5.6 Part 6

Figure 5.32 illustrates part 6 before the filling operation with some prominent features.

The processing of the features named in figure 5.32 is illustrated by Figures 5.33 through 5.38 shown in this Section.
Figure 5.33  Part 6 after processing simple hole

Figure 5.34  Part 6 after processing the step feature
Figure 5.35 Part 6 after processing round pocket

Figure 5.36 Part 6 after processing rectangular pocket 1
Figure 5. 37  Part 6 after processing rectangular pocket 2

Figure 5. 38  Part 6 transformed into raw stock
5.7 Part 7

Figure 5.39 illustrates part 7 before the filling operation with some prominent features.

The processing of the features named in Figure 5.39 is illustrated in this section. Figures 5.40 through 5.44 illustrate the processing of part 7. While processing step 1 in the above figure, the arc edge of the step, which is formed because of through hole 3 is recognized as a part of a slot. Figure 5.40 shows the processing of step 1. The slot if filled first and the step is extruded to the next logical level.
Figure 5. 40 Part 7 after processing step 1

Figure 5. 41 Part 7 after processing through hole 1 and step 1
Figure 5.42  Part 7 after processing through hole 2 and step 2

Figure 5.43  Part 7 after processing through hole 3
Figure 5.44 Part 7 transformed into raw part
5.8 Part 8

Figure 5.45 illustrates part 8 before the filling operation with some prominent features.

![Diagram of Part 8](image)

**Figure 5.45 Part 8**

Figures 5.46 through 5.51 illustrate the processing of all the features named in figure 5.45.
Figure 5.46 Part 8 after processing through hole

Figure 5.47 Part 8 after processing pocket 1.
Figure 5.48  Part 8 after processing slots 1 and 2.

Figure 5.49  Part 8 after processing slots 3 and 4.
Figure 5.50  Part 8 after processing the step feature with through hole

Figure 5.51  Part 8 transformed into raw stock
5.9 Part 9

Part 9 is of particular importance because the backward filling method fails in this case. The reason behind this failure is the complex geometry involved in the part, such as cylinders intersecting other cylinders. The sequence of figures shown in this section illustrates the processing of this part and also points out the stage where the method fails to process the feature formed because of the intersection of cylinders. The program terminates with an error at this point. The error is thrown by Unigraphics and is discussed further in this section. Figure 5.52 shows part 9.
5.9.1 Phase 1 of Processing

Figure 5.53 illustrates phase 1 of processing for part 9.

(a) Part 9 at the start of the filling operations.
(b) Part 9 after filling slots.
(c) Final planes extruded and united with part 9
(d) Part 9 after extruding the plane as shown in (c)

Figure 5.53 Phase 1 of filling operation for part 9
5.9.2 Phase 2 of Processing

Phase 2 of the filling operation for part 9 is shown by Figure 5.54.

(a) Part 9 at start of phase 2

(b) Part 9 after filling the slots

Figure 5.54 Phase 2 of filling operation for part 9
5.9.3 Phase 3 of Processing

Figure 5.55 (a) shows part 9 at the start of phase 3 of the filling operation and Figure 5.55 (b) shows part 9 after the slots are processed.
5.9.4 Phase 4 of Processing

The backward filling method fails in this sequence. From a different perspective, part 9, after filling the slots as shown in Figure 5.55, is illustrated as shown in Figure 5.56. The solid obtained after extruding the plane (shown in Figure 5.56 (b)) cannot be united with the part and therefore, Unigraphics throws an error. The program terminates at this point, thus stopping further processing of part 9. The algorithm developed in this thesis cannot process features, such as cylinders intersecting other cylinders. This is a drawback of the backward filling method developed and should be addressed.

Figure 5. 56 Failure phase of filling operation on part 9
Chapter 6

Conclusions and Future Scope

6.1 Conclusions

The method developed in this thesis successfully eliminated some of the problems inherent in the previous methods for feature recognition and volume computation. From a manufacturing perspective, the backward filling method developed in this thesis extends its scope towards predicting important factors, such as set-ups required for manufacturing a part, material removal volume, area of surface machined, etc., that contribute towards estimating the cost of manufacturing the part. The method is easy to understand and imagine from a layman’s point of view, and a designer, who usually knows little about the manufacturing aspects of the part, can also get an overview of how the part will be actually processed.
The backward filling method implemented here accomplished the following tasks:

- Identifying complex features, such as blends, slots and through holes, that involve cylindrical geometry.
- Providing a method to handle interacting features such as, protrusions existing in pockets, pockets existing inside other pockets etc.
- Calculating accurate removal volumes for machining features.
- Predicting the set-ups required for machining a part
- Predicting the diameter of the cutter required for machining each feature existing on the part.

In general, the method developed here did not restrict itself to feature identification and volume computations. It extended its scope towards determining parameters required for estimating the cost of manufacturing the part. Although this thesis does not deal with the computation of the manufacturing cost, it provides a path, which can be further developed to attain the goal of cost estimation.

6.2 Drawbacks of the Backward filling Method

This section summarizes the handicaps of the backward filling method. As seen in Chapter 5 which describes the working of the method on various parts, the backward filling method tries to overcome problems concerning feature recognition and interaction that have not been addressed before. Nevertheless, there are issues, such as processing
non-prismatic geometry, curved surfaces, spline edges and ellipse edges, that the backward filling method developed in this thesis cannot handle. Furthermore, processing features involving the intersection of cylindrical surfaces is another area that needs more work. The algorithm begins with the arbitrary initialization of the direction of the set-up. Since the selection of this initial set-up or approach direction is arbitrary, the algorithm may yield different results if the initial approach direction is changed. Also, this method has not been developed to process rotational or turned parts. Figure 6.1 is an example of a part that complies with all the assumptions stated for the backward filling method in Chapter 3 and yet would fail to be processed.

![Interaction between a protrusion and a through hole](image)

**Figure 6.1 Interaction between a protrusion and a through hole**

In the above part, the backward filling method would fail to process the inside loop on face A, formed partly by the protrusion and partly by the through hole. This would be the case if the direction of machining were the positive Z direction as shown in the figure. On the other hand, if the −Z direction is considered to be the direction of machining then the
algorithm would fill the through hole and then process the protrusion separately. Figure 6.2 shows the part shown in Figure 6.1 in a different perspective. As seen in the figure, the algorithm can identify the inside loop formed by the through hole and will fill the through hole if face B is selected for processing before face A. In this case, the two features (the through hole and the protrusion) would be processed separately, even though they interact as shown in Figure 6.1

Figure 6.2  Perspective 2 for part shown in figure 7.1

6.3  Future Scope

To improve the usability of the method, the following additions could be made to the approach:

- Developing the method to handle feature interaction between cylinders.
The algorithm developed in this thesis does not provide a method to process features where cylinders intersect other cylinders. This problem is pointed out in section 5.9 of Chapter 5.

- Processing features that involve non-orthogonal geometry.
  This problem is important because the method developed in this thesis cannot process features like chamfers and planes that do not intersect at right angles.

- Extending the backward filling method to process rotational or turned parts.
  If the backward filling method can be extended to process turned parts, a more robust and standardized algorithm could be developed that can process different types of parts.

- Extending the method to process geometry other than planes and cylinders such as cones, toruses and spheres.

- Developing the backward filling method for 4 or 5 axis machines.

These are limitations and need to be addressed to make the backward filling method a more robust and standardized algorithm.
References


Appendix A – CQL and TNR
A.1 Introduction to the Cad Query Language (CQL)

To successfully produce a cost estimate, a cost estimator needs CAD model data. In order for the estimate to be of any use to the designer, it must relate cost to the CAD model features [16]. In the attempts made in the past at feature based estimation, in the absence of feature based modelers, mechanisms had to be devised to aggregate geometric elements into features. This either resulted in loss of information, as with feature recognition, or in significant discomfort to the designer, as with feature identification. The complexity of the part to be estimated, in both cases, was restricted. Feature based modeling offered the capability to overcome the problems mentioned above, without restricting the complexity of the part to be estimated. CQL was designed to enable direct extraction of feature data from CAD systems through a querying interface. CQL has currently been implemented on the SGI-IRIX and HP-UX platforms using the C language, for the Unigraphics CAD system. CQL is designed for use through two types of interfaces, namely an interactive interface, where the user types in CQL queries and through the TNR programming language, which will be discussed further in this chapter. The procedure involved in processing a query in CQL remains the same, irrespective of the interface used. Given that the query issued is syntactically and grammatically correct, CQL extracts the required data from the CAD system through it’s Application Program Interface (API). The API is a set of C and FORTRAN routines available to external applications, to access CAD system data. Initially, when CQL was developed, it was designed only to extract information out of the CAD system and did not provide for
writing to the CAD part files. At present, it is possible to modify the CAD part geometry, by issuing CQL statements that have been developed in the past one-year. The required data is then passed on as character string to TNR or displayed in the form of rows and columns on the user screen, as the case may be.

The basic elements of a CQL query are:

- entities, which represent all possible CAD entities that may be present in the CAD part file
- variables, which are user specified character strings that represent a single unit or aggregate data
- constants, which may be integers, real numbers, character strings, boolean values, the null character, or an aggregate constant
- methods, which are means to entity information.

A.2 Important Clauses and Constructs in CQL

A.2.1 The From Clause

The from-clause forms the most important part of queries. It clearly identifies the set of entities or data to be extracted or on whom the required methods are to be applied. The from-clause is frequently used to construct a working set of data extracted from the CAD part file, so that it can be manipulated by the remaining portions of the query. It
establishes a working set of data as the set of all object ID’s of the particular entity in the
CAD part file. The variable declared in the query iterates through every instance ID from
this working set of data.

Example: CQL> select x->volume from x in solid;

In this example, the from clause constructs the working set which in this case is a set of all the
solids existing in the part file.

A.2.2 The Where Clause

Very often, the working set of data constructed from the from-clause needs to be filtered
based on some user-defined conditions. This is achieved by the use of the where-clause.
To facilitate this filtering process, a variety of conditional operators are provided in CQL.
Atomic conditions operate on either one or two operands, with the exception of the for-
condition. Compound conditions may also be constructed, which result from the
combined effect of several atomic conditions. The where statement may assume two
forms, the regular-where statement and the explicit-where statement. The use of the
regular where statement results in faster execution of the query as compared to the
explicit-where.

Example: CQL> select x from x in Edge where x->name == "ARC_EDGE";
In this example, the From clause constructs a set of all the edges in the part file. The where clause filters the edge named ‘ARC_EDGE” from the set. Finally, the select clause selects this filtered edge and displays its object ID.

A.2.3 The Dot (‘.’) and the Arrow Operator (‘->’)

The syntax of CQL is designed to associate variables and entities with methods to form a number of expressions. This is achieved through the use of two important operators.

- The dot operator (‘.’)
- The arrow operator (‘->’)

Both these operators indicate that the method following them is to be executed on the instances obtained from prior portions of the expression. The arrow operator is used when a method is to be executed on a variable, as in x->getSolid, while a dot operator is used when a method is to be executed on an entity or is chained to another method, as in ‘Feature.featureName’ and ‘Feature.getExps.expName’. The use of the arrow operator makes it explicit that a method is being executed on a variable and not on an entity or method.
A.3 Queries Developed in CQL

Previously, CQL queries were devised only to extract data from the CAD part file. A CQL query thus belonged to one of the following categories:

- **Select Query**, which serves to extract CQL objects/object data from the CAD part file
- **System Query**, which provide the means to manipulate CAD part files (opening, closing) and to control a CQL session.
- **Suppress/Unsuppress Query**, that lead to the suppression/unsuppression of selected Cad model features and helps in feature volume computation.

A.3.1 The Select Query

The select query is the most frequently used query in CQL. It consists of three distinct clauses, the *select* clause, the *from* clause and the optional *where* clause. After the required data has been extracted by the execution of the from clause and optionally filtered by the where clause, the select clause is used to specify the format and the portions of the working data set to be retrieved. The data to be selected is specified using the select expressions. The syntax of CQL allows several such select expressions, separated by commas, to be specified.
Example: CQL> select x->volume from x in Solid w
    where x->name == "RAW_PART";

In this example, the From clause builds the working set of all the solids existing in the part file. From this working set, the where clause picks the solid which is named “RAW_PART”. Finally, the select clause selects this solid and executes the ‘volume’ method using the object ID of the solid. The volume is displayed as the final answer.

A.3.2 The System Query

These type of queries are not really queries, but in reality are commands used to perform administrative tasks. The main kinds of system queries developed in CQL are the Open, Close, Cad, and Quit queries. The ‘Open’ query consists of the ‘open’ keyword followed by the name of the part file to open for querying. CQL allows several part files to be open at the same time. However, only one of them can be active at any given time. All the non-system queries are directed at the most recently opened CAD part file. The ‘close’ command results in the closing of all open part files. Since CQL is only functional in Unigraphics, this implies the closure of all Unigraphics part files. Closing a file does not result in any changes being saved. Examples of ‘open’ and ‘close’ queries are as shown.

CQL> open part1;
CQL> close part1;
The examples shown here are self-explanatory. Part1 is the name of the part file to opened or closed.

The Cad query is used to specify the Cad system whose files will be operated on. This query consists of the ‘cad’ keyword followed by the name of the CAD system to operate on, or a portion of the name. The execution of this query causes the software license manager daemon for the corresponding CAD system to be contacted for a license. The ‘quit’ command consists of either the ‘quit’ or the ‘exit’ keyword and results in the termination of the querying session. At this time, all the software licenses are returned.

Example of a ‘cad’ query is as shown.

```
CQL> cad "ug";
```

The name of the cad system is specified in double quotes. This query performs the task of getting a license from the cad system, Unigraphics.

### A.3.3 The Suppress and Unsuppress Queries

The Suppress and Unsuppress queries provide a mechanism to compute the volumes of features. This is based on the concept that suppression of a feature causes its contribution to the volume of the parent solid to be ignored by the CAD system. Thus, knowing the volume of the parent solid before and after suppression of the feature of interest, the
volume contributed to the solid by the feature can be determined. Both the suppress and unsuppress queries return the ID’s of all features in the suppressed state in the part file at the end of the query.

A.4 Templates and Rules (TNR)

A.4.1 Introduction

CQL is designed to allow queries to be embedded in the programming language, TNR. TNR is a scripting language developed at the Center for Advanced Software Systems Integration (CASSI), Ohio University. It has been designed specifically for the purpose of enabling software integration and supports interfaces to several data resources. This includes ASCII text files, relational databases, STEP files and computer networks. With the support of CQL, TNR programs can be used to extract CAD data by issuing embedded CQL queries.

Templates and rules are the procedures that are used to implement “write” and “read” contexts in TNR [17]. In a template context, file-objects are treated as output devices while in a rule context, file-objects are treated as input devices. Templates and Rules allow a logical separation of the tasks of information retrieval and that of information insertion with respect to the several data resources that can be interfaced with TNR. TNR was designed with two motivations in mind:
• To extract data from complex data resources,
• To generate data resources by specifying templates.

TNR provides high level pattern scanning and processing methods, functions to access DBMS’s, PART 21 file accessing facilities, and socket networking API. It can be easily applied to develop program for data manipulation and transport between different data resources. TNR combines compiling and execution, if the program is free of syntax error, into one step.

A.4.2 TNR with CQL

Three main TNR constructs enable the use of CQL queries:

• **cad_system** *(name)* This construct is used to specify the CAD system to be used.
  
  *name* is a character string referring to the name of the CAD system. This results in the license being obtained for the CAD system concerned. The license obtained is returned when the TNR program terminates. This statement has to be used before commencing a CQL session.

• **cad_part** *(part_name)* This construct specifies the name of the part file to be opened for querying.

• **cql_exec(s[, a][, delim])** This is the mechanism for issuing CQL queries. The embedded query consists of the statement of the query (s), an optional array (a[]), which holds the results of the query and an optional delimiter (delim), which is a
character string which separates results of the query. Each element of the array holds a row of the query result. For example:

\[
cql\_exec("select x from x in Solid", \text{solids[]})
\]

The above statement issues a CQL query into the part file that is most currently open and gets all the instances of a solid present in the part file. The function returns with the number of rows if it is a select query.

### A.4.3 Special Functions used from TNR

To achieve the goals of this thesis, TNR has been used for embedding CQL statements and queries, thus retrieving data from the CAD database and modifying the part geometry. Some of the special functions developed in TNR are described briefly below.

- **split(s, v1, v2, ...)** This function splits the string (s) unto arguments v1, v2, ...

  This function is used when a CQL query returns with a number of parameters stored into a single array. For example, consider the following query.

  \[
cql\_exec("select x, x->volume from x in Solid", \text{solids[]});
\]

  The array \text{solids[]}, will contain all the instances of a solid along with their respective volumes. Each element of the array holds a string, which in turn holds the object ID of the solid and it's volume. Using the split function, each element
of this string can be broken up into different parts and the information can then be used as required.

- **merged(s, fs, v1, v2, ....)** This function converts all the arguments specified by v1, v2,.... to strings and concatenates them to form the string ‘s’, using the delimiter specified by ‘fs’. Sometimes, while issuing a CQL statement, the user may want to perform the same kind of operation on, or retrieve the same kind of data from, more than one object. For example, if the user needs to know the ‘type’ of different objects in the part file, then this may be achieved by issuing one single statement in CQL, instead of issuing separate statements for each of those objects. Using the merged command in TNR, the object IDs of various objects can be concatenated as one string and passed on to the CQL statement for execution.

- **extract(s, n, re)** This function consecutively tries regular expression ‘re’ in the string ‘s’ for ‘n’ times. Each of the solids created by an intersection operation is listed in Unigraphics as ‘INTERSECTED’ and has a unique time stamp assigned to it. To retrieve the volume of the most recently created ‘intersected’ solid, it is necessary to identify the solid based on its time stamp. ‘Extract’ has been used to retrieve the time stamp associated with each of the solids. These time stamps are sorted to obtain the most recently created solid object.
Appendix B - Pseudocode
The six different processes performed in the backward filling method (explained in chapter 3) are described here along with supporting algorithms, so as to provide an understanding of the working of the backward filling method. The algorithm is explained with step-by-step description of each part of the algorithm with figures wherever necessary. Statements in the pseudocode described here that require the execution of a CQL statement begin with the word CQL.

B.1 Main

main(partfile) /* Name of partfile passed into the function as a parameter. */
{
    
    CQL - Obtain license from the license manager daemon of the Cad system
    CQL - Open the part file
    CQL - Select all the solids existing in the part file
    if(number of solids in the part file != 2) 
        Throw Error and terminate;
    else
        CQL - Create the buildfile by exporting the finished part and the raw part
    
    Initialize a defining direction. /* This direction is the base direction for machining. The base direction can be any one of (0,0,1), (0,0,-1), (0,1,0), (0,-1,0), (1,0,0) or (-1,0,0). In this thesis the defining direction has been always initialized to (0,0,1) */
    While(findDirectionAndPlane(base direction, &plane))
    {


process the plane;
output the set-up information;
output the features information;
}end while
}

B.2 Find Direction of Machining and a Machinable Plane

The primary task accomplished is the search of a machinable plane. Since the backward filling method always starts machining the lowest plane in a particular direction, this algorithm searches for the lowest machinable plane in a particular direction and returns the plane.

findDirectionAndPlane(base direction, plane)
{
    CQL - Find all the planes of the finished part that are not in the marked set of planes and not in temporary marked set of planes and store it in an array, found_planes[].
    /* The marked set of planes contains all the planes that are not machinable, while the temporary marked set contains planes that are temporarily un-machinable. */
    if(number of planes found == 0){ /* This means that the only planes existing are temporary marked planes */
        CQL - select all planes in the finished part that are not in the marked set. /* These are all the planes in the temporary marked
set. This statement is necessary to eliminate the planes that were stored in the temporary set but were machined at a later stage. */

if(number of planes in temporary marked set == 0)
    return false; /* There are no more planes to machine. */
else{
    for(i=0; i< number in temporary set; i++){
        if(edge linked to temporary[i] does not exist){
            plane = temporary[i];
            CQL - find the defining direction dl of temporary[i];
            updateSetUps(base direction, dl);
            break;
        } end if
    } end for
} end else
} end if

/* The else condition indicates that there are planes in the finished part other than the ones in the marked and the temporary marked set. */
else {
    CQL - find the defining direction, dd of found_plane[0];
    max = dot product(base direction, dd);
    d = dd;
    For(i=1; i< number of planes in found_plane[]; i++){
        CQL - find the defining direction, dd of found_plane[i];
        m = dot product(base , dd);
Find maximum dot product between the planes. Dot product = 1 breaks the loop.

} end for

updateSetUps(base direction, d);

base = d;

/* Search for the lowest plane in the newly assigned base direction begins */

minValid = false;

for(i=0; i< number of planes in found_plane[]; i++){

CQL - find the defining direction, dd and defining point. dp of found_plane[i];

if(dot product(base, dd){

m = dot product(base, dp);

If(minValid == false){

minValid = true;

min = m;

plane = found_plane[i];

}

else{

if(m < min){

min = m;

plane = found_plane[i];

}end if

}end else

}end if

}end for
B.3 Process the Machinable Plane

In this part of the algorithm, the plane in consideration is checked for machinability. This is done by classifying the outside loop of the plane as machinable or non-machinable. If the outside loop is deemed un-machinable, the plane is stored in the set of temporarily marked planes, else the plane is processed.

processPlane(plane) {
    CQL - Find the outside loop of the plane;
    if (classifyLoop(plane, outside loop) == un-machinable) {
        temporary_marked_set[number in temporary_marked_set] = plane;
    } else {
        fillLoops(plane);
        fillCorners(plane);
        fillSlots(plane);
        projectUp(plane);
        marked_set[number in marked set] = plane; /* Store the plane in the marked set after processing */
    }
    return;
} end processPlane
B.4 Classify a Face Loop

This algorithm classifies a face loop as to whether it belongs to a depression, a hole, or a step. The algorithm classifies the loop as belonging to a depression, step or a hole. It returns a zero if the face loop is not machinable. To classify the loop, it calls classifyEdge to classify each edge of the loop. Depending upon the nature of each edge, the class of the entire loop is decided. It also returns the up-edges and the down-edges found on the loop.

classifyLoop(plane, loop){
    CQL - Find all the edges of the loop, e[]
    Loop class = INITIAL /* Assign any arbitrary number to loop class.
    This step is to trigger the classification process. */
    For(i=0; i< number of edges in e[]; i++) {
        switch(loop class){
            case INITIAL:
                switch(classifyEdge(plane, e[i])){
                    case UP_EDGE: loop class = DEPRESSION; numUpEdges++; break;
                    case DOWN_EDGE: loop class = HOLE; numDownEdges++; break;
                    case UN-MACHINABLE: return false;
                } end switch
            case HOLE:
            case DEPRESSION:
            case STEP:
        } end switch
    } end For
}

switch(classifyEdge(plane, e[i])){
    case UP_EDGE: loop class = STEP; numUpEdges++; break;
    case DOWN_EDGE: loop class = HOLE; numDownEdges++; break;
    case UN-MACHINABLE: return false;
} end switch

case STEP:
    switch(classifyEdge(plane, e[i])){
        case UP_EDGE: loop class = STEP; numUpEdges++; break;
        case DOWN_EDGE: loop class = STEP; numDownEdges++; break;
        case UN-MACHINABLE: return false;
    } end switch

case DEPRESSION:
    switch(classifyEdge(plane, e[i])){
        case UP_EDGE: loop class = DEPRESSION; numUpEdges++; break;
        case DOWN_EDGE: loop class = STEP; numDownEdges++; break;
        case UN-MACHINABLE: return false;
    } end switch
} end switch

)end for
return loop class;
} end classify loop

B.5 Classify an Edge

Here, an edge is classified as an up-edge, down-edge or un-machinable. To classify an edge, the adjacent face to the face in consideration and sharing this edge is determined.
The dot product of the defining directions of these two faces sharing the edge is determined. If the adjacent face is planar and the dot product is not equal to 0, it means that the adjacent face is not perpendicular to the face in consideration. This kind of a planar face is deemed as an inclined face and the edge is classified as un-machinable. Similarly, if the adjacent face is cylindrical and the absolute value of the dot product does not evaluate to 1, the edge is classified as un-machinable. This algorithm is used by classifyLoop for classifying each edge of the loop.

classifyEdge(plane, edge){
    CQL - Find the adjacent face to plane and sharing the edge.
    CQL - Find the defining direction and defining point of both the faces
    If (adjacent face is planar){
        If(dot product(defining direction of main face and adjacent face) != 0)
            return 0;
    } end if
    else if(abs(dot product(defining direction of main face and adjacent face)) != 1)
        return 0;
    Find the vector running from the defining point of main face to the defining point of the adjacent face.
    If(dot product(vector and defining direction of main face) >= 0)
        return UP_EDGE;
    Else
        return DOWN_EDGE;
    } end classifyEdge
B.6 Fill all the Inside Loops on a Plane

This algorithm fills all the inside loops existing on a plane. It searches for an inside loop on the plane and fills the inside loop by calling fillInloop. The inside loops that are a part of a hole existing on the face are filled here. Since the plane in consideration is the lowest plane all the inside loops filled up represent through holes.

```
fillInloops(plane)
{
  CQL - Find all the inside loops existing on the plane, in_loops[];
  For(i=0; i< number of loops in in_loops[]; i++){
    switch(classifyLoop(inside loop)){
      case UN-MACHINABLE: error; break;
      case HOLE: /* Inside loop represents a hole */
        fillInLoop(inside loop);
        break;
      case PARTIAL_BOSS: break; /*this indicates the existence of a partial boss. Refer figure 4.3*/
      case BOSS: break; /* This indicates the existence of a boss */
    } end switch
  } end for
} end fillInLoops
```
B.7 Fill the inside loop representing a through hole

The inside loop is extruded in a direction opposite to the defining direction of the face on which the loop exists. This is the procedure for filling up through holes.

```c
fillInLoop(inside loop){
    Direction of extrusion, = reverseDirection(defining direction of plane on which inside loop exists);

    CQL - Find any vertex on the edges of the inside loop;
    start of extrusion = dot product(defining direction of plane and vertex);

    CQL - Find all the edges of the loop, e[];
    end point of extrusion = findMinHeightFromLoop(e[], dir_extrusion);
    /* The algorithm for findMinHeightFromLoop is explained further in
```
extrudeAndCalculateVolume(e[], direction of extrusion, distance of extrusion, &volume, &area, &object_tag);

/*The extrudeAndCalculateVolume algorithm is explained in section B.15. It returns the removal volume, area of surface machined and an object tag corresponding to the solid object created after intersection */

Features[numFeats].vol = volume;
Features[numFeats].area = area;
Features[numFeats].solid = object_tag;
Features[numfeats].perimeter = e_length(e[]); /*Calculates the total of the lengths of all edges in e[] */
Features[numFeats] = minLoopDia(e[]);
numFeats++;
}

B.8 Fill all the round corners/ blends existing on the plane

This algorithm searches for all the round corners or blends formed by the outside loop of a plane. It finds all the arc edges on the outside loop of a plane. Arc edges on the outside loop may represent a round corner or a slot. If the arc edge is an outside arc, it represents a round corner else it represents a slot. It simply calls fillCorner to fill the round corner.

fillCorners(plane){
    CQL - Find all the edges of the outside loop of the plane, e[];
for(i=0; i< number of edges in e[]; i++) {
    CQL - Find the type of the edge, e[i];
    If (edgeType is an arc edge) { /* This arc edge may be part of a slot
        or a round corner */
        CQL - Find any vertex of this arc edge;
        CQL - Find the normal, N to the arc edge at the vertex;
        vector V = center of arc - vertex;
        If(dot product(N and V) < 0) /* This is an outside arc shown in
            figure B.2 */
            fillCorner(plane, e[i]);
        } end if
    } end for
} end fillCorners

Refer to figure B.2 for a graphical representation of the various parameters calculated in
the above algorithm.
B.9 Fill a round corner or blend

This algorithm develops a set of lines tangent to the arc edge that defines a round corner. An example illustrating the lines that are tangent to the arc edge is shown in figure B.3. If the angle enclosed by the arc is less than or equal to 90° then two tangent lines are constructed. If the angle enclosed by the arc is between 90° and 180° then 3 tangent lines are constructed and so forth. These construction lines are extruded along with the arc.
edge to fill the round corner. Figure B.3 illustrates the squaring off procedure for a round corner. Since the blend shown in the figure encloses an angle of 180°, three construction lines are drawn to square off the blend. Figure B.4 illustrates the filling up procedure for a round corner. The solid obtained after the extrusion is intersected with the raw part and united with the finished part.
Figure B. 3  Squaring off procedure for a round corner

Figure B. 4  Filling procedure of round corner
fillCorner(plane, edge) {

direction of extrusion = reverseDirection(defining direction of the
    face on which the round corner exists);

CQL - Find the cylindrical face adjacent to the edge;
CQL - Find all the edges of the cylindrical face except the arc edge
in consideration, e[];
CQL - Find all the vertices on these edges, v[];
end of extrusion = dot product (direction of extrusion, v[0]);

For(i=1; i< number of vertices in v[]; i++){
    min = dot product(direction of extrusion, v[i]);
    If(min < end of extrusion)
        end of extrusion = min;
}

CQL - Find any vertex on the arc edge
Start point of extrusion = dot product(vertex and direction of
    extrusion);

CQL - find the start angle and end angle of the arc edge;
angle enclosed by arc = abs(start angle - end angle);

If (0° < angle enclosed by the arc ≤ 90°)
    CQL - Construct two lines tangent to the arc edge;
If (90° < angle enclosed by the arc ≤ 180°)
    CQL - Construct three lines tangent to the arc edge;
If (180° < angle enclosed by the arc ≤ 270°)
    CQL - Construct four lines tangent to the arc edge;
If (270° < angle enclosed by the arc ≤ 360°)
    CQL - Construct five lines tangent to the arc edge;
Store the lines constructed and the arc edge in edges[];
extrudeAndCalculateVolume(edges[], direction of extrusion, &volume,
&area, &object_tag);

Features[numFeats].vol = volume;
Features[numFeats].area = area;
Features[numFeats].solid = object_tag;
Features[numFeats].perimeter = e_length(e[]); /* Calculates the total
of the lengths of all
edges in e[] */

Features[numFeats] = minLoopDia(e[]); /* Calculates the minimum
diameter amongst all the
arc edges present in e[].

numFeats++;
} end fillCorner

B.10 Fill the Slots defined by the Outside Loop of a Plane

This algorithm searches for slots that may be formed by the outside loop of a plane. It finds all the edges of the outside loop that belong to the slot and then connects the boundary points of the slot with one or two construction lines, depending upon the nature of the slot. If the edge on which one of boundary points exists is extended and if the extension of this edge meets the other boundary point, the boundary points are connected with a single line, else two lines are constructed between the boundary points. This procedure is termed as squaring off a slot and is explained in detail in section B.1.13.
fillSlots(plane) {
    direction of extrusion = reverseDirection(definition direction of the face on which the slot exists);

    found Slot = 1;
    do {
        CQL - Find all the edges on the outside loop of plane, e[];
        For (i=0; i< number of edges in e[]; i++) {
            If (edgeType is not an arc edge) {
                found Slot = 0;
            } Else {
                CQL - Find any vertex on the arc edge;
                CQL - Find the Normal, N to the arc edge at the vertex;
                vector V = center of arc - vertex;
                If (dot product(N, V) > 0) { /* This means it is an inside arc */
                    extendCorner(e[], i, 1, &tangent_edges[], &sharpcorner1[]);
                    extendCorner(e[], i, -1, %tangent_edges[], &sharpCorner2[]);
                    /* The algorithm for extendCorner will be discussed further in this chapter. In a nutshell, extendcorner returns the tangent edges of the slot, as shown in figure B.5. It also returns the two boundary points of the slot (also shown in figure B.5) which are also called sharp corners. */
                    connectExtension(c[], sharpCorner1[], sharpcorner2[]);
            } Set the direction of extrusion to reverseDirection; 
        } 
    } 
}
Figure B. 5  Example illustrating the tangent edges of a U-slot.

/* The algorithm for connectExtension will be discussed further in this chapter. This function connects the sharp corners of a slot and includes the newly constructed lines in the array of edges to be extruded (tangent edges of slot). */

start of extrusion = dot product(direction of extrusion, vertex on arc edge);

end of extrusion = findMinHeightfromLoop(tangent_edges[], direction of extrusion);

distance of extrusion = start of extrusion - end of extrusion;

fillSlot(c[], direction of extrusion, distance of extrusion)

break; /* break out of for loop. Need to get a new set of arc
edges after filling one slot */

} end if
} end else
} end for
} While (found slot);
} end fillSlots

B.11 Fill a Slot Formed on the Outer Loop of a Plane

This algorithm determines the start point and end point of extrusion and calls extrudeAndCalculateVolume to fill up the slot.

fillSlot(e[], direction, distance){
    
    CQL - Find any vertex on the arc edge of the slot;
    extrudeAndCalculateVolume(e[], direction, distance, &volume,
        &area, &object_tag);

    Features[numFeats].vol = volume;
    Features[numFeats].area = area;
    Features[numFeats].solid = object_tag;
    Features[numFeats].perimeter = e_length(e[]);
    Features[numFeats] = minLoopDia(e[]); /*Calculates the minimum diameter amongst all the arc edges present in e[].
    numFeats++;
} end fillSlot
B.12 Finding the edges that are tangent to the arc edge defining a slot

While iterating through the array of edges belonging to the face, fillSlots calls the function extendCorner and passes the edges of the face along with the position of the arc edge of the slot in the array of edges. For example, as shown in figure B.6 the edges of the face with the slot are numbered. Each number denotes the position of the edge in the array of edges belonging to the face. The arc edge belonging to the slot is the fifth element in the array of edges and is at position 4. This algorithm returns the edges that are tangent to the arc edge along with the boundary points of the slot. It searches for tangent edges on either side of the arc edge.

![Diagram of a 3D model showing faces and points labeled A, B, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 with annotations for Faces 0, 1, 2 and points A, B.]

Figure B.6 Example illustrating the search procedure for tangent edges of a slot
For example, in figure B.6, the extendCorner algorithm will detect edge 5 as a tangent edge (first direction for search) and then detect edges 3, 2 and 1 (second direction for search). To determine if two edges are tangent to each other, the algorithm determines the normal vectors to the faces that are adjacent to the face with the slot and sharing the edges. For example, in figure 4.10, Face 1 shares edge 1 with the face with the slot while Face 2 shares edge 2 with the face with the slot. Point A is the common vertex shared by edges 1 and 2. The dot product of a unit normal drawn to Face 1 at point A and a unit normal drawn to Face 2 at point A evaluates to 1. Therefore, Face 1 is tangent to Face 2. This satisfies the tangency condition between edge 1 and edge 2. The tangency condition between edge 1 and edge 0 fails and point B is returned as a sharp corner of the slot.

```
extendCorner(e[], i, search_direction, tangent_edges[], sharpcorner[]){
    i = position of arc edge in array of edges, e[];
    While(ExtendEdge){
        next = (position of arc edge + search_direction) % (number of
               edges in edges[]);
        /*The variable dir_search can be +1 or -1 depending on which side of
        the arc edge the search is conducted. Referring to figure 4.10,
        if dir_search is +1, then edge 5 will be detected. Similarly,
        if dir_search is -1, then edges 3, 2 and 1 will be detected. */
        Find the common vertex between e[i] and e[next];
        CQL - Find the face adjacent to the plane and sharing the edge e[i];
        CQL - Find the normal N1 to this face at the common vertex;
        CQL - Find the face adjacent to the plane and sharing the edge
               edges[next];
```
CQL - Find the normal $N_2$ to this face at the common vertex;
if(dot product ($N_1$ and $N_2$) $! = 1$){ /* faces are not tangent to each other */
    sharpcorner = common vertex;
    extendEdge = 0;
} end if
else{
    Add the edge $e[next]$ to the array of tangent edges;
    i = next;
} end else
} end while
} end extendCorner

B.13 Constructing Lines to Square off a Slot

This algorithm is used to construct lines between the boundary points or sharp corners of a slot. Depending upon the nature of a slot, one or two lines are constructed to square off the slot. The algorithm returns the newly constructed lines so that they can be extruded along with the tangent edges to fill the slot.
connectExtension(e[], point A[], point B[])

CQL - Find the normal $N_1$ to Face A at point $A_i$ (refer to figure B.8)

Vector $V_1 = \text{Point B} - \text{Point } A_i$; /* refer to figure B.7 for an
illustration of the normal $N_1$
at point A and the vector
$V_1$ between point A and point B*/

If (dot product ($N_1$ and $V_1$) > 0) {
  Point $P = \text{Point A} + (N_1 \cdot \text{dot product}(N_1, V_1))$;

  CQL - Construct line $L_1$ between point A and point P /*refer to
  figure B.8*/

  e[number in e[]] = $L_1$;

  CQL - Construct line $L_2$ between point B and point P
  e[number in e[]] = $L_2$;

  return;
}

} end if

/* figure B.8 illustrates the squaring off procedure when point A is
lower than point B with reference to the + Z direction. */

else{

  Vector, $V_2 = \text{point A} - \text{point B}$ /* vector running from point B to
  point A */

  CQL - Find Normal $N_2$ to E2 at Point B

  If (dot product ($N_2$ and $V_2$) > 0) {
    Point $P = \text{Point A} + (N_1 \cdot \text{dot product}(N_1, V_1))$
    CQL - Construct line $L_1$ between point A and point P
    e[number in e[]] = $L_1$;
  }
}
Figure B. 7 Illustration of a normal at a point on a face

Figure B. 8 Illustration of squaring off procedure with 2 lines
CQL - Construct line $L_2$ between point $B$ and point $P$

```c
    e[number in e[]] = L2;
    return;
} end if

} end else

Construct line $L_0$ between Point $A$ and Point $B$.

```c
    e[number in e[]] = L0;
    return;
} end connectExtension
```

Figure B. 9  Illustration of the squaring off procedure with one line.
The final task in processing a plane is to extrude the plane to the 'next logical level'. If the plane represents a depression or a step, the extrusion limit is determined by calculating the 'next logical level' in the current part. If the plane represents a final plane, the extrusion limit is determined by computing the highest vertex on the raw part, in the direction of extrusion. In this part of the algorithm, it is important to check for protrusions that may be existing on the plane. Thus, the height of the protrusion should also be considered while computing the limit of extrusion.

```c
projectUp (plane) {
    CQL - Find the defining point of the plane;
    CQL - find the defining direction of the plane;
    Direction of extrusion = defining direction of plane;
    Start point of extrusion = dot product (defining point, defining direction);
    CQL - Find the outside loop of the plane;
    loop class = classifyLoop(plane, outside loop);
    switch (loop class) {
        Case FINAL_PLANE:
            If (protrusion){ /* check to see if there are any protrusions on the face. This can be done by checking if there are any inside loops on the face because at this stage inside loops represent protrusions only. */
                end of extrusion = \min(\text{height of all protrusions on face});
    }
}
```
else {
    Find the highest vertex on the raw part in the direction of extrusion;
    End point of extrusion = 1.1 * direction of extrusion * highest vertex. /* In this case the plane is extruded to a height that is 10% more than the raw part. */
} end else
break;

Case STEP: Case DEPRESSION:
If(protrusion)
    End point of extrusion = min(height of all protrusions);
min = next logical level. /* Refer to section 3.3.3 in chapter 3 for the algorithm to calculate the next logical level. The height of the protrusion should also be considered while computing the next logical level. */
if(min < End point of extrusion)
    end point of extrusion = min;
break;
} end switch

extrudeAndCalcVolume(face, direction of extrusion, distance,
                    &volume, &area, &object_tag);
Features[numFeats].vol = volume;
Features[numFeats].area = area;
Features[numFeats].solid = object_tag;
CQL - find all the edges on the outside loop of face, e[];
Features[numfeats].perimeter = e_length(e[]); /*Calculates the total of the lengths of all edges in e[] */

Features[numFeats] = minLoopDia(e[]); /*Calculates the minimum diameter amongst all the arc edges present in e[]. This routine is explained in detail in chapter 5. */

numFeats++;

} end processplane

B.15 The Extrusion, Intersection and Union operations

This algorithm issues CQL statements to do the extrusion, intersection and union operations. Along with performing these operations, it computes the volume of material removed, the area of surface machined and the perimeter of surface machined. Then the solid object obtained from the intersection operation is exported to a different layer where it is displayed to represent the material removed from the raw part to machine the particular feature.
extrudeAndCalcVolume(objects[], direction of extrusion,
        distance of extrusion, volume, area, object_tag) {
        CQL - Extrude the objects in objects[];
        CQL - Intersect the extruded solid with the raw part;
        CQL - Obtain the volume of the solid obtained after intersection;
        Area of bottom surface of feature machined = volume/ distance of
        extrusion;
        object_tag = CQL - Obtain object ID of intersected solid;
        CQL - Export the solid obtained from the intersection operation to a
different layer. /* This creates a copy of the solid. */
        CQL - Unite the copy of the intersected solid with the finished part.
    } end extrudeAndCalcVolume

B.16 Determining the "Next Logical Level" to Find the Limit of Extrusion for
a Feature

This algorithm is used for determining the end point of an extrusion operation. It returns
the minimum value or the "next logical level" to which the object or objects are to be
extruded. A brief description of this algorithm with an illustration is provided in chapter 3
in Section 3.3.3.

findMinHeightfromLoop(plane, edges[], direction, start_extrusion){
    For (i=0; i< number of edges in edges[]; i++){  
        CQL - find the face adjacent to edges[i] not equal to plane;
CQL - find the edges of the adjacent face not equal to edges[i], e[];
For(j=0; j< number of edges in e[]; j++) {
    CQL - find extreme point on each edge in the direction specified by 'direction';
    h = dot(extreme point[], direction);
    minValid = 0;
    if(h > start_extrusion) {
        if(minValid == 0) {
            if(minValid == 0) {
                min = h;
                minValid = 1;
            } end if
        else{
            if(h < min) {
                min = h;
            }
        }
    } end findMinHeightFromLoop
    return min;
} end findMinHeightFromLoop