AN OPEN FBD SYSTEM FOR GENERALIZED ORTHOHEDRAL COMPONENTS

by

Yogesh Kulkarni

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Mechanical Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

Abstract

Increased demands on productivity in manufacturing have lead to increased automation of design and manufacturing processes. Automating CAD and CAM independently does not necessarily provide the "best" solution for the overall product realization process. Computer integration of both these activities is the potential solution.

This thesis proposes a feature-based design approach for three dimensional block structured components with generalized orthohedral geometry. The component is viewed as a collection of blocks connected by topological links. The geometry and topology are explicitly defined. The geometry of the feature consists of an external shape and an internal shape, while the topology is modeled using a graph. The basic external shape of each block in a component is a rectangular parallelopiped which fixes overall size and location of the block. Using the idea of encapsulation, families of blocks with different internal shapes can be developed; all these internal shapes are assumed to be encapsulated in a block with rectangular external shape. The simple nature of the external shapes makes it easier to define operations like create, addition/deletion of blocks, topological connectivity specifications, etc. The internal shape provides the necessary information for geometry specific operations such as display, volume computation, etc.

The feature library consists of constructive as well as subtractive features. The modeling of free-form surfaces using the idea of encapsulation is proposed and an extruded B-Spline surface is developed and implemented as an example.

An editing method has been developed to maintain topological consistency during geometrical modifications of the component and to propagate the editing changes with minimum user intervention.

Dimensioning and tolerancing support is provided using a graph representation. Capabilities such as automatic creation of a dimensioning scheme, change of datum, detection of over-dimensioning and under-dimensioning, etc are provided.

C++ is used to obtain an object-oriented implementation of the feature- based design concepts developed in this work. The user-interface is developed in X-Windows/Motif toolkit while the graphics display is done using PHIGS.

Contents

Table of Contents i

List of Figures iii

List of Tables v

1 Introduction **1**

1.1 Background 1

1.1.1 Solid Modeling 1

1.2 Feature Based Modeling Techniques 3

1.2.1 Feature Based Design 4

1.3 Object oriented implementation 4

1.4 Literature Review 6

1.5 Research Objectives 7

1.6 Organization of the thesis 8

2 Feature Based Design Model **10**

2.1 Introduction 10

2.2 Orthohedral block-structure 11

2.3 Data Representation 12

2.3.1 Blocks 12

2.3.2 Constructive and Subtractive Features 16

2.3.3 Topological Links 18

2.4 Operations 22

2.4.1 Creation 24

2.4.2 Editing 25

2.4.3 Algorithm for Editing 34

2.4.4 Change-Length(*Node N, Axis A, New-Length L)* 37

2.4.5 Display 38

2.4.6 Polygon-Set Display Algorithm 42

2.4.7 Common Edge Detection Algorithm 43

2.4.8 Algorithm for function In-line(edge $E1(V_1, V_2)$, vertex P_1) 45

2.4.9 Merging of the polygons 45

3 Encapsulation **48**
3.1 Introduction 48
3.2 Basics 48
3.3 Advantages of Encapsulation 53
3.4 Extension of the Feature Library 53
3.5 Swept B-spline Encapsulated Block 55
3.5.1 Operations 56

4 Dimensioning and Tolerancing **58**
4.1 Introduction 58
4.2 Background 59
4.3 Deficiencies in current D & T data models 61
4.4 Basic Concepts 63
4.5 Dimensioning and Tolerancing model 64
4.5.1 Graph Representation 64
4.5.2 The D & T Graph Model 65
4.5.3 Data Structures 65
4.5.4 Operations 69
4.5.5 B-spline Dimensioning 74
4.6 Example of Basic Dimensioning 75
4.7 Bspline Dimensioning 78

5 Implementation **83**
5.1 Introduction 83
5.2 Object Oriented Implementation 83
5.2.1 Object Class Definitions 84
5.2.2 Templates 91
5.3 Graphical User Interface 93
5.3.1 Interface Structure 93

6 Conclusions and Recommendations **97**
6.1 Conclusions 97
6.2 Recommendations 98

References **100**

A Class Definitions **103**

List of Figures

2.1 Data Representation 13
2.2 External and Internal Shape 14
2.3 A representation of the MIN, MID, and MAX planes. 15
2.4 Constructive and Subtractive feature plane assignment 17
2.5 Symmetric Nature of the topological links 18
2.6 Center Face Match 19
2.7 Center Edge Match 19
2.8 One Corner Match 20
2.9 Two Corner Match 21
2.10 Four Corner Match 21
2.11 Shared Plane Match 22
2.12 Component and its graph representation 23
2.13 Editing a component with Four Corner Match 26
2.14 Blocks with TCM before Editing 28
2.15 Blocks with TCM after Editing 28
2.16 Plane fixing in Editing 29
2.17 Length Linking in Editing 30
2.18 Cycle in Graph 32
2.19 Component having Cycle before editing 33
2.20 Inconsistency in Component editing having Cycle 33
2.21 Block-wise Display 39
2.22 Common Edge Detection 41
2.23 Polygon Set Display 42
2.24 The possible ways of common edge 44

3.1 Constructive Block 50
3.2 Subtractive block 50
3.3 Constructive Cylinder 51
3.4 Subtractive Cylinder 51
3.5 Constructive Wedge 52
3.6 Subtractive Wedge 52
3.7 Constructive B-spline 53
3.8 Constructive Wedge encapsulated in Block 54

4.1 Common dimensioning for co-planer surfaces 61
4.2 Different dimensioning schemes for cylinder 62
4.3 Dimensioning scheme for B-spline 63
4.4 The RDG and the SDG 66
4.5 Dimension Graph 67
4.6 Graph Data Structure and Dimension Class 68
4.7 Specifying dimension between two planes 71
4.8 Change of Datum Plane 72
4.9 Over-dimensioning 73
4.10 Example Component 75
4.11 Dimensioning the component 76
4.12 Specifying the dimension between two planes 77
4.13 Change in Datum 79
4.14 Example Component 80
4.15 Bspline Dimensioning 81

5.1 Block Hierarchy 86
5.2 Topology Hierarchy 90
5.3 Main Window Interface 94

List of Tables

5.1 Object Classes 84

Chapter 1

Introduction

The integration of computer-aided design (CAD) and computer-aided manufacturing (CAM) requires a large amount of information transfer between the design and manufacturing functions. Compatibility of the information representation scheme in both domains is still a key issue in integration. The ease with which data transfer takes place influences productivity and cost effectiveness of the product. As yet, there does not exist a single scheme that fully supports a seamlessly integrated environment for CAD/CAM activities.

Presently available CAD data models can provide unique and unambiguous geometric representation of a part; however, they can only provide information about the nominal size of the part, and generally are unsuitable for complete product definition. From the viewpoint of designers and computer-aided manufacturing (CAM) applications, conventional CAD systems are still not convenient to use [1]. The product models stored in CAD databases can support only limited technological information.

Feature based design offers a tool for mapping the designer's abstraction into a representational form.

1.1 Background

1.1.1 Solid Modeling

A solid modeler is the central element in many contemporary computer-aided design systems. CAD has come a long way since its inception in 1963 by Sutherland [2].

The main motivation was to automate the process of design as much as possible and thus reduce human intervention. This brings about accuracy and consistency. Early methods in CAD were limited to drafting capabilities. 2D wireframe representation of solid objects, which was common then, was lacking in volumetric information and true representation of solidness. Two methods of solid representation, Boundary Representation (B-rep) and Constructive Solid Geometry (CSG), became dominant approaches and remain so to a large extent.

B-rep is based on Adjacency Topology. A solid object is represented by a collection of external and internal (cavity) shells. Each shell is a closed volumetric region and is divided into lower level topological entities in the following hierarchical fashion : faces$\rightarrow$loops$\rightarrow$ edges $\rightarrow$ vertices [3]. This decomposition yields a graph structure. The topological entities are associated with corresponding geometric entities: surfaces, curves, and points. Since geometric entities are defined parametrically, the dimensions of the solid are not represented explicitly in the model. Also, since topological decomposition is done at the global level for an object, there is no association between shape features, such as holes and slots, and their constituting topological entities, such as faces and edges.

In *CSG*, the solid is built with the help of standard solid primitives by boolean operators, viz., union, difference and intersection. The solid is represented by a tree with the solid primitives as leaves and boolean operators as internal nodes. Difficulties arise when the part becomes complicated. Features that do not belong to the primitive set must be constructed by several primitives and operators; therefore, they are not uniquely represented. Further, because CSG is an *unevaluated* scheme, attaching dimensional and tolerance information to low level entities is impossible unless the whole CSG tree is evaluated. Current CAD systems are mismatched in their level of abstraction with downstream application [1]. A purely geometric solid modeler does not recognize a part in terms of its application. For example *a thread*, in a solid model, will be characterized by its length, radius, and pitch. In feature based design, these geometric parameters would be supplemented by higher level information such as thread form, material, surface characteristics, manufacturing methods, process planning rules, etc. This makes the product definition more complete.

It is clear that solid modelers cannot be used to drive applications such as process planning or manufacturability evaluation because some of the information needed by these tasks is totally absent from the solid model database [4].

It may therefore be concluded that feature-based design is a necessary tool for product definition and for integration of design and manufacturing.

1.2 Feature Based Modeling Techniques

Much of the initial work on features was motivated by a desire to device methods to extract part geometry from geometrical modelers so that process plans, Group Technology (GT) codes, and Numerical Control (NC) programs could be generated. Thus, the manufacturing view of features is that features represent shapes and technological attributes associated with manufacturing operations and tools.

Many approaches exist for creating feature models in a geometric modeling context. To provide a framework for comparison, it is convenient to classify these methods into three broad groups.

1. *Interactive feature definition* : The geometric model is created first, and then features are defined by the user by picking the entities on an image of the part. Dimension and tolerance study for CAM-I [5] is based on this approach.

2. *Automatic feature recognition* : In this approach also a geometric model is created first; then a computer program processes the database to discover and extract features automatically.

3. *Design by features* : The part geometry is defined directly in terms of features and geometric models are created from features. Three sub-categories of the *design by features* approach are discussed below :

   (a) *Feature databases un-associated with solid models* : The user inputs feature information textually using a customized syntax. When the processes are selected to create a process plan, the reasoning process can be driven by high level parameters, such as feature type, generic parameter values, tolerances, and attributes. The availability of a solid model may not be necessary.

   (b) *Destructive modeling with features* : A part model is created by boolean subtraction of features from the base stock model .

   (c) *Synthesis by features* : This approach allows the user to design by adding or subtracting features without a starting base stock. The features stored in the libraries may be applications oriented. This is the approach used in the ASU feature testbed [4], which consists of two shells, one for part design and the other for mapping and applications. The shells contain mechanisms for defining generic features in design.This allows user organizations to customize the system, avoiding the difficulty of working with a hard coded set of features.

1.2.1 Feature Based Design

Features are regarded as groupings of geometrical and topological entities that need to be referenced together.With the advent of the *design by feature* approach, the definition now seems to be much broader : *features are the elements used in generating, analyzing, or evaluating designs* [6].

The main goal is to make product definition as complete as possible. For example, a *step* not only implies a cylindrical geometry characterized by diameter and length, but it can also associated with a turning process, tolerances, surface finishes and process planning sequences which are unique to a *step*. As another example, consider a shaft and a pin. Both are geometrically similar i.e. cylindrical. But in the domain of feature based design they could carry completely different information. The type of fits recommended for the shaft are likely to be different from those for the pin. They may also differ in size range and manufacturing / heat-treatment processes. In short, they are different features, even though they may share similar geometry.

Features such as holes and slots provide a convenient language for specifying mechanical parts, and facilitate automatic process planning and other downstream activities in the life cycle of machinable products [7]. This is direct result of the fact that features are application oriented rather than being based purely on geometry.

1.3 Object oriented implementation

Object oriented implementation is now common in feature based modeling. The reason behind this is that there exists a natural mapping between the object-oriented methodology and feature based design.

Object oriented programming is a technique for programming that is based on the following principles [8] :

1. *Data hiding* refers to the concept that our only access to the data in the object (or *class* in C++ terminology) is through "methods" defined in those classes. This "information hiding", when used to maximum effect has several benefits including reliability, understandability, and (in some cases) efficiency.

2. *Inheritance* is the creation of a new class as an extension or specialization of an existing class. It allows the conceptual relationship between different classes to be made explicit.

3. *Polymorphism* allows us to send identical messages to different objects and have each object respond appropriately.

4. *Dynamic binding* is binding of the specific code as late as possible, i.e. at run time. When the *display* method is called for an object, the class of the object is determined first. Then, the actual function to be executed for the method is determined by looking up the method name and class identifier in the method table.

The parameters that characterize the feature correspond to the data members in the class definition and the methods that are devised specifically for that feature correspond to the class member functions. The clustering of feature specific knowledge within each feature object allows the feature to handle and manage changes internally, while the global operations are uniform. For example, a *Block* feature can be characterized by *length, breadth* and *height.* These quantities become data members of *class Block.* If the application needs a volume computation function, say *Volume()*, then this function becomes a member function of *class Block* which returns the positive value of the product of its *length, breadth* and *height.* Object oriented implementation also exploits similarities between members of feature families by using property inheritance. For example, if we want a feature corresponding to rectangular pocket, we can define *class RectPocket* by inheritance from *class Block.* We may add some data members and member functions or just change the behavior of some member functions. For example, the *Volume()* function for this class will return the negative value of the product of its *length, breadth* and *height.*

The ability to specify methods along with the data members is very important in feature-based design. Conventional solid modelers do not provide methods related to the shape except for a few simple geometry-based functions such as volume and area calculations. Feature-based design needs to provide higher level methods for the features such as process planning, strength calculations, etc.

Objects are never concerned with the internal operations on other objects, although changes may be requested through the proper protocol.

Inheritance is also used heavily to characterize a family of features having similar characteristics. For example, *class Cylinder* can have as subclasses *class RoundCylinder, EllipticalCylinder, TaperedCylinder*, etc. Every subclass adds specialization to its superclass. It also can add/modify the functionality supported in its superclass. For example, superclass *Cylinder* may have only *radius* and *height* as data members ; however, the subclass *EllipticalCylinder* may add one more data member to define the minor axis, along with some changes in functions such as *Volume* computation. Thus, inheritance captures the inbuilt characteristics of a family of components, making the feature library extendable with minimal efforts.

1.4 Literature Review

Most of the early work in feature-based design concentrated on finding procedures and methodologies to extract feature information from existing geometric modelers in a form that was suitable for generating process plans, GT codes and NC programs. The manufacturing view was the leading influence on the field [9] until design by features [10] came into the picture.

The IMPARD System [11] was built around the GeoMod solid modeler to evaluate designs of a subclass of injection molded parts based on some simple manufacturability criteria such as wall thickness, corner radius, boss dimension, melt flow length, etc.

Casu et al. [12] proposed a Feature Based design system that uses Euler operators and some macro operators.

Falcidieno et al. [13] formulated a Structure Face Adjacency Hypergraph (SFAH) that represents a feature based model by addition of a set of attributes which complete the form feature information.It also stores information about assembling the component.

Requicha [7] suggested definitions of machinable features and proposed system architectures for feature based design and manufacturing.Volumetric features are parametrized solids and surface features are groups identified in a part made up of faces. It is based on functional (design) features and CSG. Operators can be applied to geometry, feature type, tolerance and geometric constraints. The input is parsed into a feature based representation, which is converted by an Expander into a CSG tree.

Shah [4] used a parallel representation in the ASU feature testbed : a boundary model and a CSG model with a union and difference operator. It consists of two shells, one for definition of the part (Feature modeling shell) and the other for definition of application specific objects. The FMDS (Feature modeling shell) consists of procedures for defining generic features, adding features to the library, and using those features in design. The FMPS (Feature Mapping Shell) extracts and reformulates product data as needed by the application. A custom interpretive language was developed to set up the application's computation and extraction procedures. The customization is left to organizations using it.

Hijazi [14] developed a prototype of a functioning feature based design system for orthohedral components. The system supported creation, editing and display functionality and was written in C++ using 3D GMR graphics routines.

Chennapragada [15] extended the capabilities of Hijazi's model in the area of geometric and topological editing. A mesh generation capability was developed for component's designed with constructive orthohedral primitives. A component's data could be saved in DXF format which could then be read into commercial software such as AutoCAD for further processing and annotation.

Yang [16] developed a similar prototype but with the data represented as a binary tree; this was implemented in the Silver Screen 3D CAD/Solid Modeling software package.

Pro/ENGINEER [17] is a successful commercial package developed using the parametric, feature-based approach. This unique, fully associative suite of mechanical design automation software includes application-specific products which address the complete spectrum of product-development activities.

1.5 Research Objectives

The current generation of computer-aided design and manufacturing software has made available tools for partial automation of some of the tasks related to product development and manufacturing. It has not only created islands of automation, it has created islands of optimization, too [4]. It is possible to find widespread deficiencies in the current state of feature-based design.

Most solid modelers provide good support for geometric modeling, but lack a rich set of topological features. The interaction between geometry and topology needs to be handled better.

Most of the current solid modeling systems are of a very generalized nature and tend to be complex. By limiting oneself to a well defined class of shapes the complexity can be reduced. Care should be taken to ensure that the subset that is supported is large enough to encompass the majority of anticipated component shapes. The subset that is considered in this thesis is the family of generalized orthohedral components.

Current feature-based design systems also lack good dimensioning and tolerancing support. The present work develops and implements dimensioning related functionality that allows the user to create, modify and validate the dimensioning scheme for a component.

The present work also aims to providing a facility to design various components with the available feature set by using different topological features for specifying connectivity relationships between geometric features.

The objectives of the research can be summarized as follows:

1. To define a data representation scheme for modeling of three dimensional solid components with generalized orthohedral geometry. This class is broad enough to cover constructive blocks, subtractive blocks and a large family of encapsulated shapes which can be extended to include customized shapes. The representation scheme supports explicit geometric and topological specifications.

2. To provide a mechanism for extendability so that the current work can serve as a foundation for other higher-level applications such as process planning, automatic NC code generation, finite element analysis, etc.

3. To provide a systematic basis for a dimensioning and tolerancing scheme.

4. To provide a mechanism for customizing and expanding the system through the use of encapsulated blocks.

5. To implement a prototype system to test the above mentioned principles and concepts.

1.6 Organization of the thesis

Chapter 2 describes the basic structure of the feature based design approach developed in this thesis. Chapter 3 explains the proposed dimensioning and tolerancing scheme in detail. Chapter 4 presents the idea of encapsulation and describes the process of extending the feature set with the example of a *Wedge.* A B-Spline extruded prism feature is also developed here. Chapter 5 discusses some implementation issues related to the software prototype development. This discussion includes the object-oriented methodology as well as the graphical user interface (GUI) which was developed using Motif and PHIGS. Chapter 6 concludes the thesis with some remarks and recommendations.

Chapter 2

Feature Based Design Model

2.1 Introduction

Models are basically approximations of some real world situation, object or concept. A model consists of the information that is needed by the application and all other parameters are ignored. The success of a CAD model depends on the following factors :

1. Capturing designer's intent

2. Simplicity

3. Potential to create physically realizable objects

4. Rich set of operators to manipulate the created objects

5. Robustness

Most conventional solid modelers use a data structure that can model a large variety of shapes. Though advantageous in the sense of *completeness*, they are often lacking in terms of efficiency while working with parts that have relatively simple geometry. Efforts devoted to formalizing methods that operate on any generalized shape tend to be complex and cumbersome for simpler shapes. Thus, it may be possible to exploit the simplicity of a family of components to great advantage. One of the most common and relatively unexploited part families is the family of block-structured parts. The present work further restricts this family to the class of generalized orthohedral block-structured parts.

2.2 Orthohedral block-structure

An orthohedral block-structured object is one which only contains faces that are perpendicular to a coordinate axis. Orthohedral geometry makes it easier to decompose a component into a set of rectangular blocks positioned relative to each other. As the planes of attachment are finite and simple (each being perpendicular to a coordinated axis), positioning takes less effort than positioning more complex shapes.

It is clear that pure orthohedral geometry is not sufficient for modeling mechanical parts, since curved and sloping surfaces are often necessary. To take advantage of a *simplicity* offered by the orthohedral block-structure and still have the capability to model shapes with curved and sloping surfaces, a combined data representation is proposed. In this data representation, the shape of the part is decomposed into two aspects :

1. External shape
2. Internal shape

The external shape has the orthohedral block-structure, while the internal shape defines the actual geometry of the part. A mapping is defined between the external shape and the internal shape. Additional parameters may be needed to define the internal geometry with respect to the external block. Once the external shape gets defined completely, the mapping takes care of the internal details and thus *encapsulates* the inner geometry. This idea of *encapsulation* is dealt with in more detail later in this thesis.

Orthohedral block-structured geometry offers the following advantages :

- *Creation* : Easier to position blocks with respect to each other.
- *Editing* : Locational editing is simple to perform. Mapping between external and internal shape takes care of transmitting editing changes to internal geometry.
- *Display* : Time is saved by displaying block-wise in the initial stages of modeling.
- *Object-oriented implementation* : Extension of shapes is easier as only a new *mapping* needs to be defined. The new shape can still use all the old functions that were applicable to the external shape.

2.3 Data Representation

The data representation scheme that is proposed here is an extension of the approach followed by Hijazi [14], Chennapragada [15], and Yang [16]. In this scheme, a component is considered to be a set of blocks interconnected by topological links and is perceived as a solid. In the following sections, the basic data structure is explained in more detail.

Any representation scheme for generalized orthohedral components must handle two principal types of data related to the shape of the component : geometric data and topological data. Geometric data defines the shape and size parameters while topological data defines the size independent connectivity relationships among primitive features.

In our model, the geometric data is contained in the blocks that make up a component and the topological data is contained in the topological links.

Figure 2.1 shows the decomposition of data information into geometric and topological information. The geometry contains plane values with respect to a global coordinate system, while topology contains the type of connectivity between the blocks.

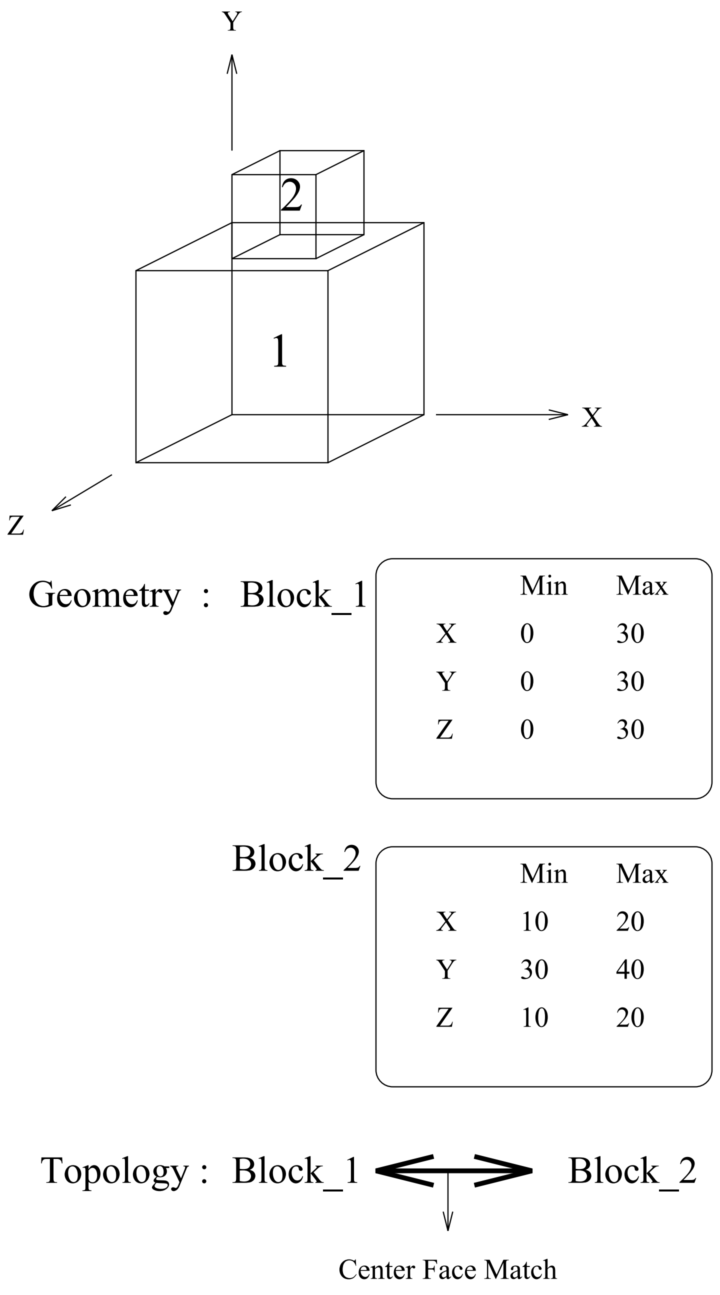
The representation of blocks and topological links is detailed in the following sections.

2.3.1 Blocks

The geometry of a block is divided into two parts, viz., external geometry and internal geometry. External geometry is represented by a rectangular parallelopiped block which adheres to the idea of orthohedral block-structured geometry. All topological connectivities are defined with respect to the external shape and are thus independent of the internal geometry of the block. Internal geometry represents the actual shape of the block, which may or may not be the same as the external shape. Functionality like display, local edit, and volume computation are specific to the internal shape. For example, a cylinder can be decomposed into an external shape and an internal shape as shown in Figure 2.2.

Basic External Shape

Each object is assumed to be encapsulated in a rectangular block defined by a set of nine planes and three lengths. Orthohedral geometry means that each plane is perpendicular to one of the three coordinate axes of the global Cartesian coordinate system (Fig. 2.3).



Block_1

	Min	Max
X	0	30
Y	0	30
Z	0	30

Block_2

	Min	Max
X	10	20
Y	30	40
Z	10	20

Figure 2.1: Data Representation

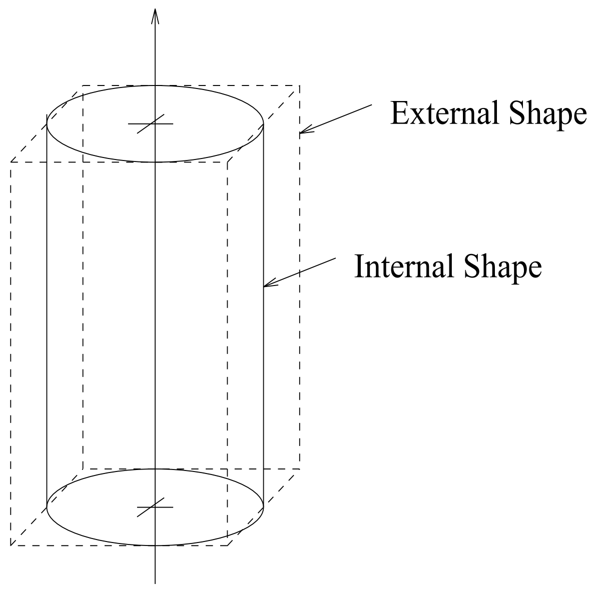


Figure 2.2: External and Internal Shape

Each axis is intersected by three planes denoted as MIN (minimum), MID (middle) and MAX (maximum). The length of the block along a particular axis, denoted Length, is defined as the distance between the MIN and the MAX planes along that axis. Three MAX planes (Xmax, Ymax, Zmax) along with three MIN planes (Xmin, Ymin, Zmin) define the bounding planes of the basic external shape.

The MID planes (Xmid, Ymid, Zmid) and the Lengths (Xlen, Ylen, Zlen) are implicitly defined with respect to the MAX and the MIN planes. The MID planes and the Length are redundant parameters which are maintained because it may sometimes be easier to define the block using one or both of these parameters. Some topological links use the MID planes for link specification. The MID planes explicitly define the position of a block with respect to the global coordinate system. The Length of the block helps in providing further explicit geometric information about the size. Given any two parameters along an axis, the other two can be computed. This is done to reduce the input required and insure that geometric consistency is maintained.

The external shape is used to define topological relationships and the relative position between two blocks. For all blocks, the user has to define the external shape completely first and then specify the parameters related to the internal shape. For a rectangular block, the external shape itself acts as the internal shape and no further information is necessary

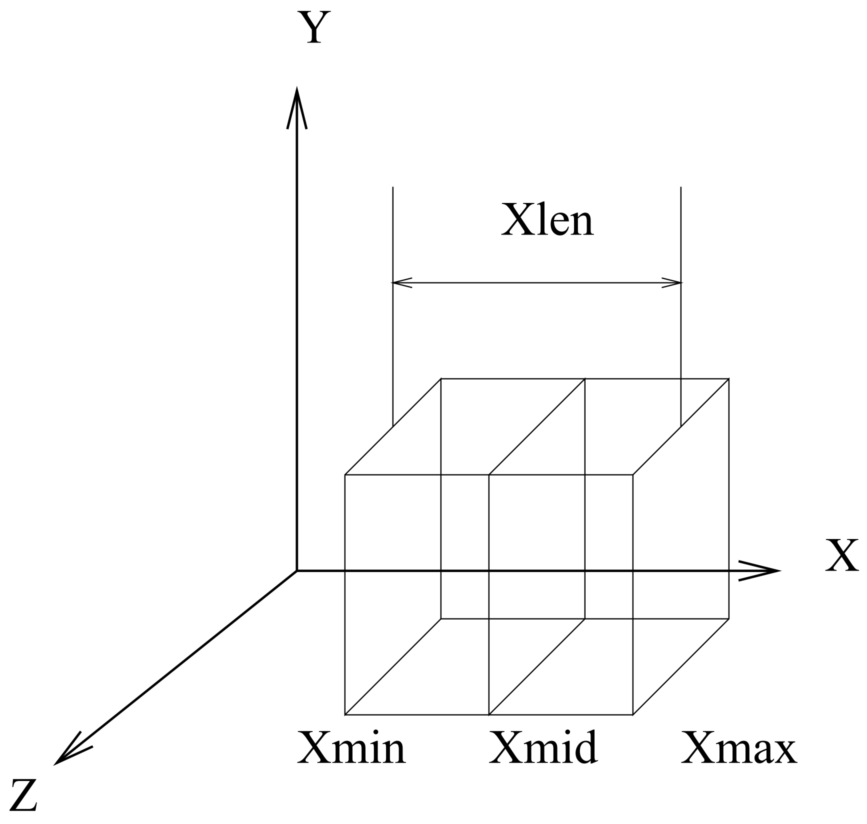


Figure 2.3: A representation of the MIN, MID, and MAX planes.

to complete the definition.

Internal Shapes

The internal shape specifies the geometry of the feature with respect to the external shape. This is done by providing a mapping between the external rectangular shape and the internal geometry. Some additional data may be necessary to complete the geometry definition. This gives us a way to extend the shape library and develop customized shapes. The internal shape is encapsulated within the external shape and thus remains local to the block.

The internal shape does not take part in global positioning of the block and does not participate in any topological links. They contribute only to the operations that are *shape specific*, such as display and volume calculations.

Once the external shape gets defined, the additional input needed depends on the type of the block. The prototype system developed by Hijazi [14] had only a constructive block feature. Later Chennapragada [15] extended the feature library to include subtractive blocks and cylinders. The present work develops a hierarchy of shapes having blocks, cylinders, wedges and extruded B-Spline surfaces.

2.3.2 Constructive and Subtractive Features

Constructive features can be defined as features in which the normals to the faces are outward from the block. Constructive features represent *solid* volume and constitute the primary structural building blocks.

Subtractive features can be defined as features in which the normals to the faces are directed inward towards the block. Subtractive features represent voids. In the current prototype system, the subtractive features are implemented as derived classes of their corresponding constructive feature classes. The different *normal* directions for constructive feature faces and subtractive feature faces are achieved by linking vertices in the counter-clockwise direction in case of the constructive features and in the clockwise direction in the case of the subtractive features.

Subtractive features differ from constructive features in the way the resting plane is assigned to the new feature from the existing feature. The assignment of the planes in the case where both the new and the existing features are subtractive is similar to the case where both are constructive features.

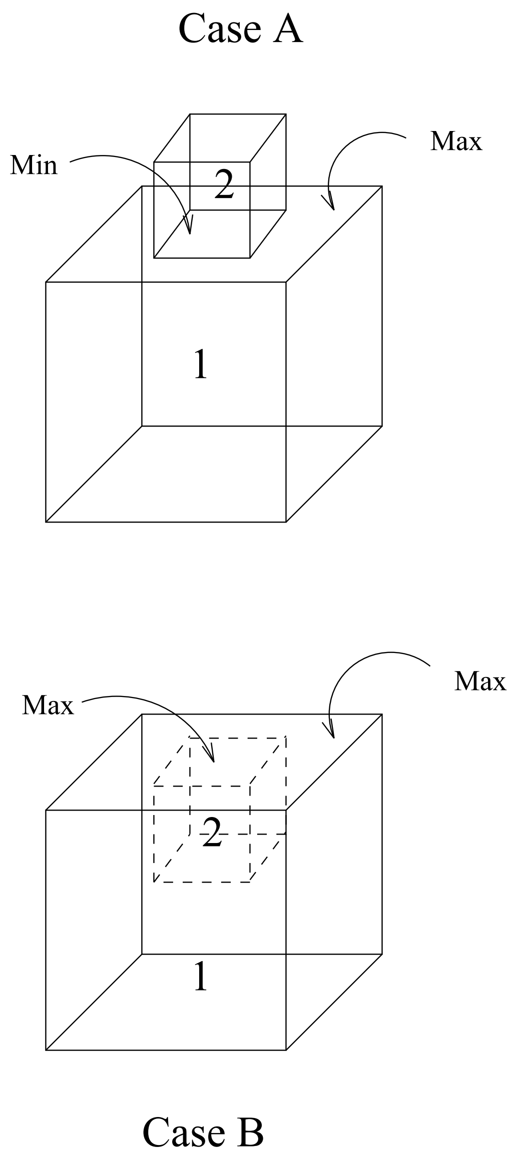


Figure 2.4: Constructive and Subtractive feature plane assignment

Figure 2.4 shows the difference in plane assignment in the case of a *center face match* topological link. In case A, the MAX plane of block-1 matches with the MIN of block-2, while in case B, the MAX of block-1 matches with the MAX of block-2. Similar differences occur for other topological links.

The current implementation provides subtractive block, subtractive cylinder and subtractive wedge features.

2.3.3 Topological Links

Topology defines relationships that must exist regardless of size between the external shapes of different blocks in the component. In the current implementation, six types of topological relations are developed, all of which are symmetrical in nature, i.e.

if $\sim$ is a topological relation, then
A $\sim$ B $\Longleftrightarrow$ B $\sim$ A
where A and B are the blocks involved in the relationship.

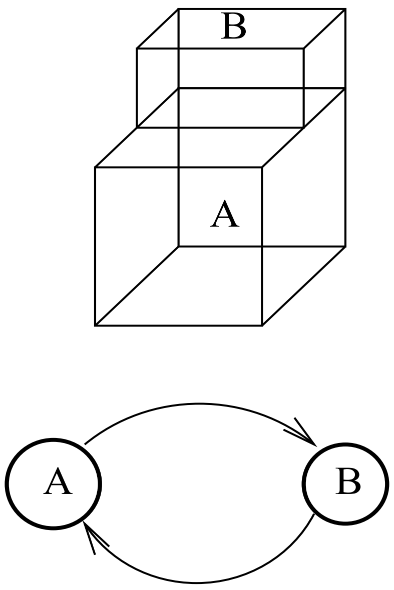


Figure 2.5: Symmetric Nature of the topological links

For example, if a block A is two corner matched with a block B, then block B is also two corner matched with block A. Figure 2.5 shows an example of a two corner match with its graph representation. The graph reflects the symmetric nature of the topological links. A complete description of all the topological links that are supported is given below. The plane assignments in all cases are for a link between two constructive blocks.

1. Center Face Match [CFM] (Figure 2.6)

   A MAX plane of one block is aligned with the MIN plane of the second block along the same axis. In addition, the two MID planes of the first block which are perpendicular to the other two axes are aligned with the corresponding planes in the second block.

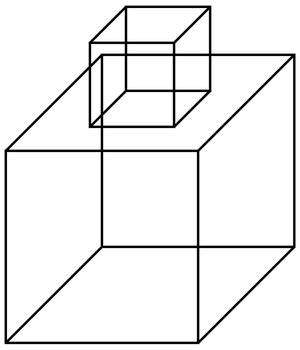


Figure 2.6: Center Face Match

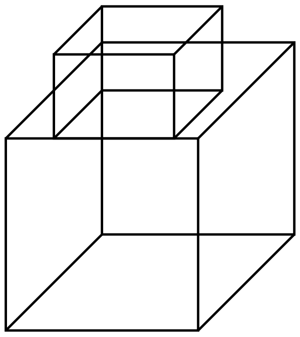


Figure 2.7: Center Edge Match

2. Center Edge Match [CEM] (Figure 2.7)

   The MAX plane of one block is aligned with the MIN plane of the second block that is perpendicular to the same axis. In addition, a MAX of the first block is aligned with the MAX plane of the second block or a MIN plane of the first block is aligned with the MIN plane of the second block which is perpendicular to a second axis. The MID planes along the third axis are also aligned.

3. One Corner Match [OCM] (Figure 2.8)

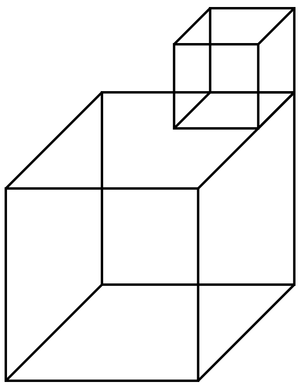


Figure 2.8: One Corner Match

   The MAX plane of one block is aligned with the MIN plane of the second block that is perpendicular to the same axis. In addition, a MAX plane and one of the MIN planes of the first block which are perpendicular to the other two coordinate axes, are aligned with the corresponding planes in the second block.

4. Two Corner Match [TCM] (Figure 2.9)

   The MAX plane of one block is aligned with the MIN plane of the second block that is perpendicular to the same axis. In addition, either two MAX planes and one of the MIN planes of the first block are aligned with the corresponding planes of the second block or two MIN planes and one of the MAX planes of the first block are aligned with the corresponding planes of the second block.

5. Four Corner Match [FCM] (Figure 2.10)

   The MAX plane of one block is aligned with the MIN plane of the second block that is perpendicular to the same axis. In addition, two MAX planes and two MIN planes

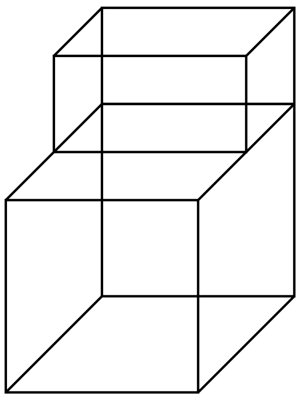


Figure 2.9: Two Corner Match

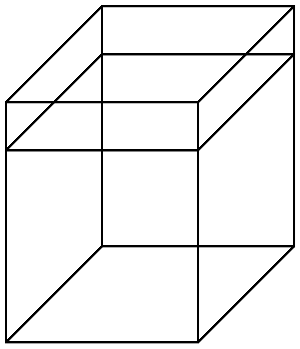


Figure 2.10: Four Corner Match

perpendicular to the other two coordinate axes are aligned with the corresponding planes in the second block.

6. Shared Plane Match [SPM] (Figure 2.11)

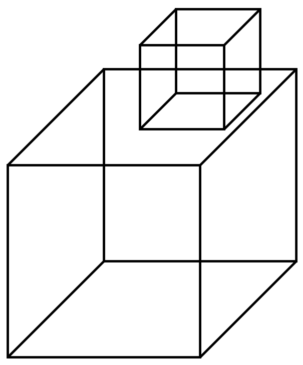


Figure 2.11: Shared Plane Match

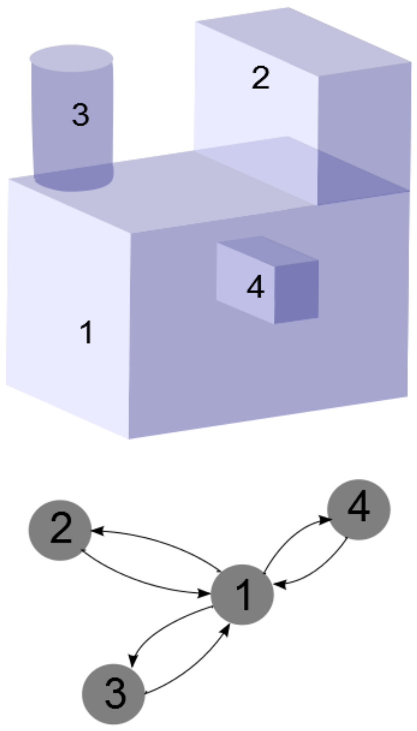
The MAX plane of one block is aligned with the MIN plane of the second block that is perpendicular to the same axis.

Graph representation of the topology

Although a component can be conceptually represented as a collection of blocks and a collection of topological links, this type of the model does not provide an adequate formal representation of the relationships between the blocks and the topological links. Therefore, in our model, a component is represented as a list of blocks and a connected graph. The graph is a directed graph, in which nodes represent blocks, and the incident edges represent the a topological links. The edges store information related to the topological link such as axis and type along with the information related to the type of topological link. This is illustrated in Fig. 2.12.

2.4 Operations

This section details some of the basic operations that must be supported in order to make the FBD system usable. All operations are discussed with respect to the data representation scheme that was presented in the previous section.



Nodes	Block_Type
1	ADDT_Block
2	ADDT_Block
3	ADDT_Cylinder
4	SUBT_Block

Arcs	Topolink_Type
a	Two Corner Match
b	One Corner Match
c	Center Face Match

Figure 2.12: Component and its graph representation

2.4.1 Creation

The creation process can be broken down into three types :

1. Creation of a component
2. Creation of a block
3. Creation of a topological link

Creation of Components

The creation of a component is initiated by the creation of the first block purely from geometric data. This block will serve as the base to which other blocks can be added to form the desired component. A list of blocks is maintained in the order in which they are added to the component. Most subsequent operations on the whole component traverse the list of the blocks and perform the corresponding operation at the block level. For example, displaying the component is done by traversing the list of the blocks and displaying them in order.

Creation of blocks

For purely geometric data input (as in the case of the initial block in a component) the user needs to specify two data values along each axis. The other two parameters are calculated internally. To add a block to an existing component, the type of topology that will exist between the new block and at least one pre-existing block must be specified. The establishment of this topological link will result in the alignment of certain planes from the pre-existing block with the corresponding planes from the new block. Since the values of the planes involved in the topological link from the pre-existing block are known, the values of the corresponding planes in the new block are established. Therefore, a portion of the new block's geometric data is defined, and the remaining data necessary to complete the definition of this block must be provided. This is done by prompting the user for the additional geometric data that is required. Other blocks are added to the component in a similar manner until the desired component is generated. As each block is created, it is added to the list of blocks that make up the component.

Many of the mechanical parts constituting an assembly bear geometric proportions with themselves or within other parts in the assembly. When there arises a need to modify any part dimensions, the related (linked) dimensions need to be changed to maintain the validity of the assembly. This automatic change in the dimensions of the parts that are geometrically related to the edited dimension can be achieved if there exist geometric length links between them. This can be done in creation mode when the user is entering the length value of a feature, at which time he/she could be given the option of linking this value to an existing one. This capability has not been implemented in the present work.

Creation of Topological links

Topological links specify the relative position of one block with respect to another. There are two ways in which topological links can be created :

1. Linking a new block to a pre-existing block
2. Linking two pre-existing blocks

Every topological link needs a base surface on which the contacting planes of the two blocks meet. Depending on the type of the topological link, more information may be requested to complete the definition of that link. The user may specify more than one topological link between two blocks, provided they are compatible. Every topological link fixes the data of the second block with respect to the first (base) block. In linking two pre-existing blocks, care should be taken to ensure that their planes lie in such a way that the requested topological link can be set up without changing the existing geometry.

2.4.2 Editing

There are two types of editing that are supported in this implementation : geometrical editing and topological editing. Geometrical editing involves changing the shape parameters of the blocks and topological editing involves changing the topological relations between the blocks.

Geometrical Editing

Geometrical editing is performed at two levels.

1. External, which involves changes in the external shape of the block
2. Internal, which involves changes in the internal shape of the block

The editing process can be broken down into the following major tasks :

- Selecting an object to modify
- Accepting a parameter to modify in the selected entity
- Validating the specified parameter
- Actually performing the specified change in the parameter
- Propagating the change correctly through the component

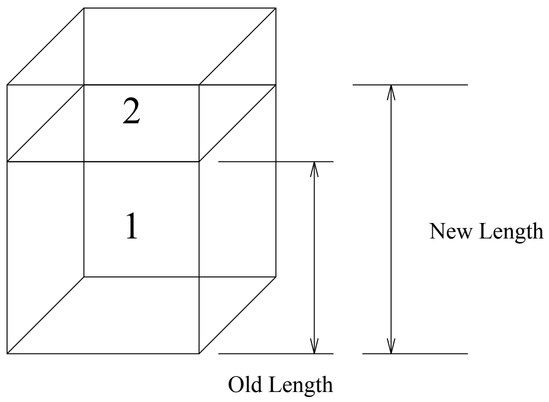


Figure 2.13: Editing a component with Four Corner Match

For example, consider the component in Fig. 2.13. The following is the step by step process to change *Ylen* of Block 1, while keeping the Ymin plane of Block 1 fixed.

- Select an object to modify :
  - Block-Id = 1
  - Axis = Y
  - Parameter = Length
- Present the value of old length and ask for new value, new-length.
- Validation of the input :
  - If the new value is same as old value; no changes are performed.
  - The new value should be a non-negative value.

- Change the *Ylen* field to *new-length.*
- Propagation : As Block 2 is connected to Block 1 by the topo-link of type *four corner match* along Y axis, modifying Ylen of Block 1 shifts the Ymax of Block 1 upwards. To maintain alignment, the Ymin and Ymax of Block 2 are shifted upwards by the same amount, new-length.

The graph structure used to model the component proves to be especially advantageous in the editing operations. The entire logic for editing can now be written as a well-structured graph algorithm [18].

Geometric editing involves much more than just changing the dimensions of the block selected by the user. Since the blocks are linked to each other by topological links, the changes in the dimensions of one block may require changes in the dimensions of other blocks that are connected to it either directly or indirectly, in order to preserve the topology.

Difficulties in the Editing process

The editing process needs to be understood clearly so as to avoid ambiguity and un-wanted results. Some of the issues that must be handled during the editing process are the following :

- Violation of topological links :

  During the editing of the component, if the geometry of a block is changed, some of the topological links may be violated unless other changes are made simultaneously. For example in Figure 2.14, a topological link of the type Two Corner Match (TCM) exists between blocks 1 and 2. After performing a geometric change on the Xlength of block 1 while holding the Xmin plane fixed, this topological link will be violated as shown in Figure 2.15. Therefore all the blocks in the component must be visited to determine which of the topological links are violated, reestablish them, and perform appropriate geometric changes.

- Specification of invariant plane :

  It is not sufficient to specify only the length to be changed; a plane of the block that remains invariant during the edit must be specified. For example, if Xlen (see Fig. 2.16) of Block 1 is to be changed, the user has option of fixing one of the following planes.

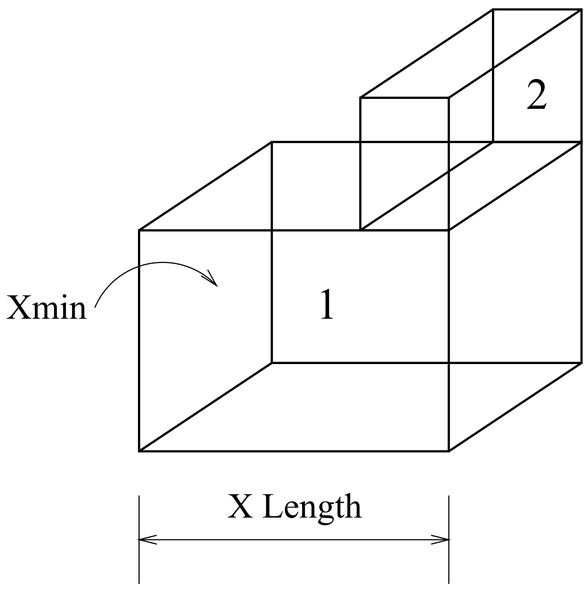


Figure 2.14: Blocks with TCM before Editing

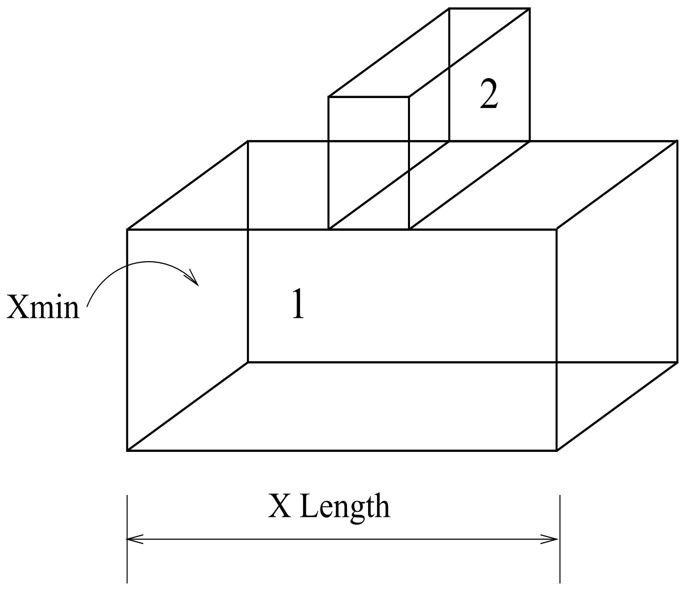


Figure 2.15: Blocks with TCM after Editing

- Xmin
- Xmid
- Xmax

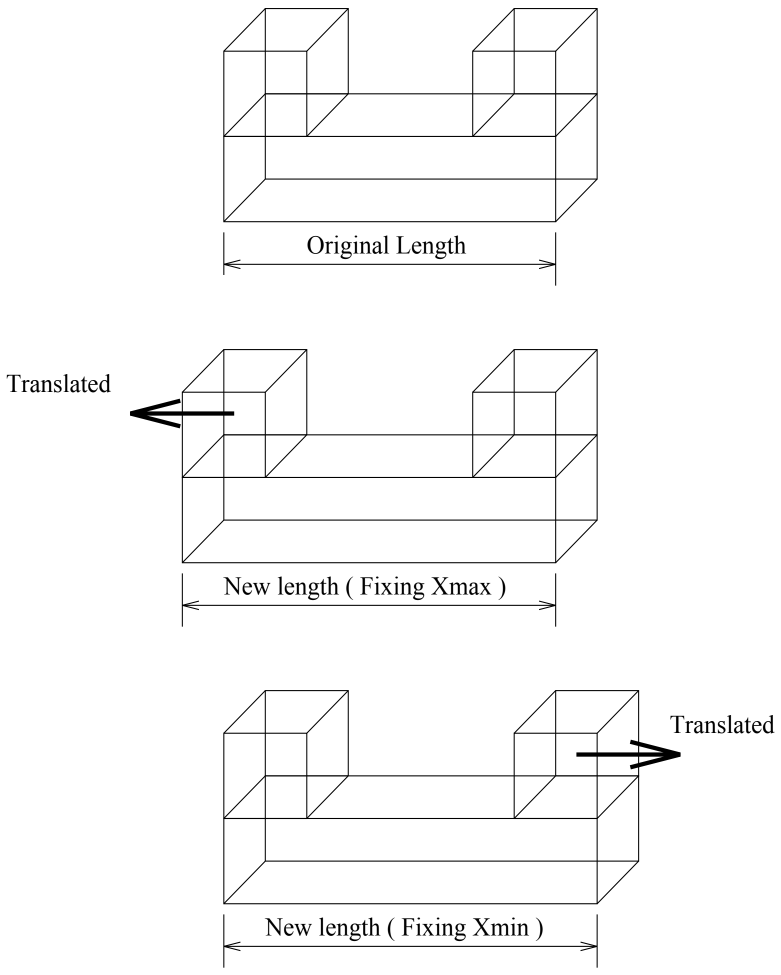
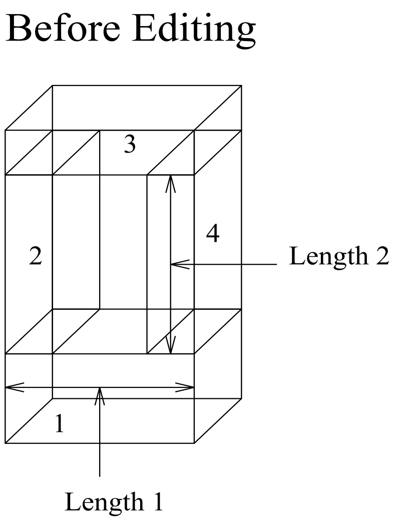


Figure 2.16: Plane fixing in Editing

- Length Linking :

  If the length to be edited is linked to another length then the editing changes must occur in the second length to maintain the length relation. Every length that is changed must be propagated to topologically linked blocks; this should be correctly propagated during the editing process. Consider the example shown in Fig. 2.17, with Length 1 linked to Length 2. Suppose Length 1 is to be changed.



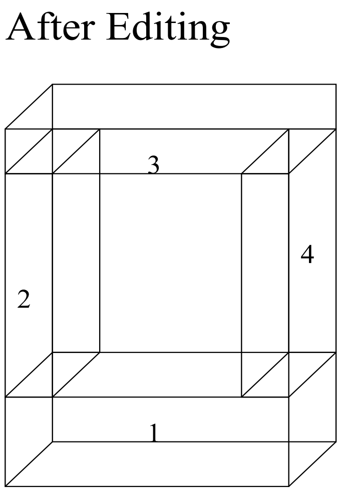


Figure 2.17: Length Linking in Editing

The changes that occur when "Length1" is modified keeping Xmin fixed, are as follows

- To maintain the topological link of type "Two Corner Match" between block-1 and block-4, block-4 translates to the right.
- Because of length linking between "Length-1" and "Length-2", Ylen of block-4 gets modified keeping Ymin constant (say). So block-4 extends in the upward direction.
- To maintain the topological link of type "Two Corner Match" between block-3 and block-4, the block-3 translates upwards.
- To maintain the topological link of type "Two Corner Match" between block-3 and block-2, block-2 extends upwards.

The final configuration is shown in Fig. 2.17(b).

- Cycles in a component :

  The previous example was a simple one in which the blocks in the component are linked in a cycle. This can be seen in the graph of the component shown in Figure 2.18. The editing algorithm must be capable of handling components with any number of cycles and propagating the geometric changes correctly.

  For example, a Two Corner Match exists between blocks 1 and block 4 in Figure 2.18. Suppose the user wants to edit the Xlength of block 1 with the Xmid plane fixed (Figure 2.19). As Xlength increases with Xmid fixed, Xmin of block 1 shifts to the left and Xmax of block-1 shifts to the right. To maintain the Two Corner Match between block-1 and block-2, Xmin and Xmax of block-2 move to the left, keeping its Xlength fixed. Similarly, to maintain the Two Corner Match between block-1 and block-4, Xmin and Xmax of block-4 shift to the right. It is clear from Figure 2.20 that to maintain the topological consistency between block-2 block-3 and between block-4 & block-3, a change in the Xlength of block-3 (Figure 2.20) is necessary. After the selection of block-3 as the modification node, the new length is internally calculated to satisfy both the topological links. During this process it is assumed that :

  - Only one length is allowed to be changed per cycle
  - The user selects a correct node for the modification of length.
  - If the edit is possible without modifying any length, it will be done in this way.

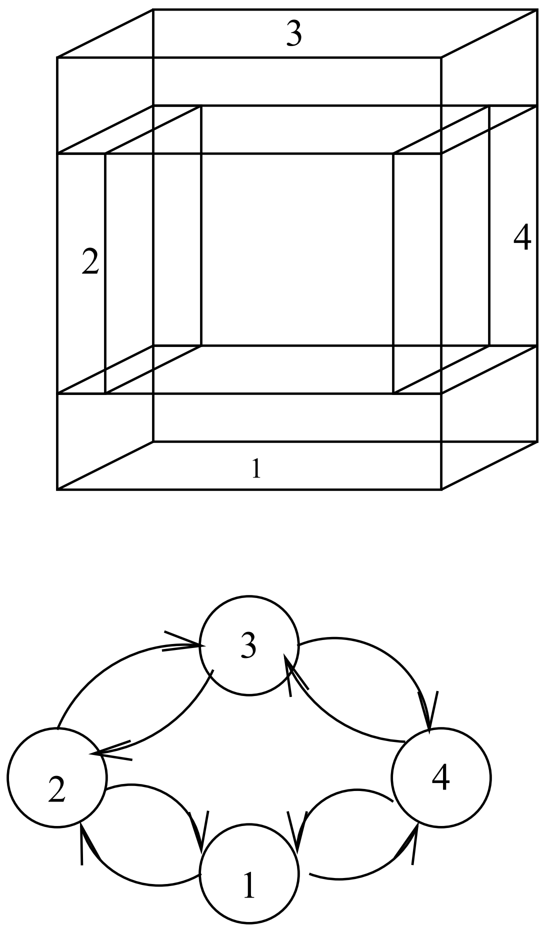


Figure 2.18: Cycle in Graph

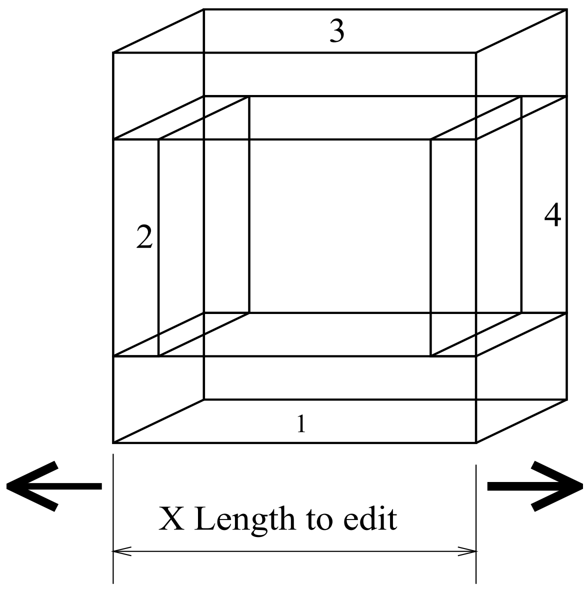


Figure 2.19: Component having Cycle before editing

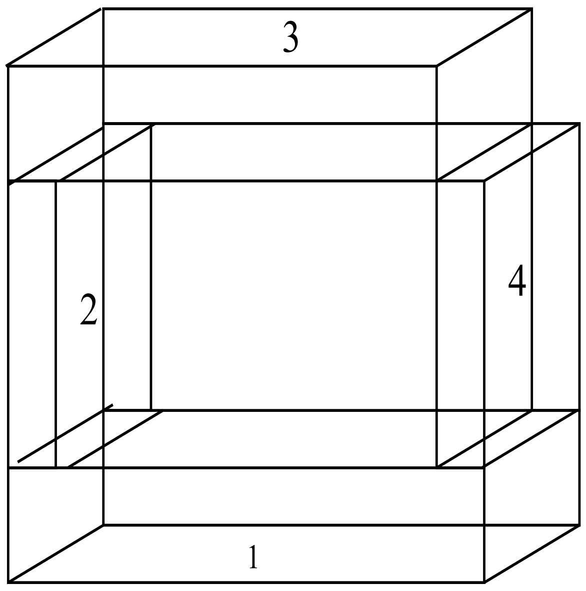


Figure 2.20: Inconsistency in Component editing having Cycle

2.4.3 Algorithm for Editing

Before presenting the method of traversal used in the editing process, several variables must be first defined.

- *Fixed* : A boolean variable associated with the lengths and planes of a block. This variable is set to "true" when the blocks are initially created. If this variable is "false", the value of the length or the plane can be changed.
- *Resolved* : A boolean variable associated with each graph node. This variable is set to "true" when all the planes and lengths constituting the block that the node represents have their *Fixed* variable set to "true".
- *Visited* : A boolean variable associated with each node. This variable is set to "true" when all cycles that the node is a member of have been traversed, and all adjacent nodes have their *Resolved* variable set to "true".
- *Modify* : A boolean variable associated with each node. This variable is set to "true" if the node is part of a cycle and its length is allowed to be modified.
- *Base node* : A node is labeled as the *Base node* when it is the current node removed from the queue of the nodes to be visited during the editing process.
- *Update* : A procedure that recalculates the values of those lengths and planes of a block that have their *Fixed* field set to "false". After recalculating their values, *Update* sets their *Fixed* variable to "true".

Before each graph traversal, the graph is decomposed into strong-components to find all possible fundamental cycles [19], and sets all the boolean variables to "false". The steps of the algorithm are as follows :

1. The block and length are selected for modification. The user inputs new length value denoted as *new-length* and *axis* along which length is to be modified. The node representing the edited block is labeled as the *edited node.*
2. Three queues are set up corresponding to the three coordinated axes, viz., XQ, YQ, and ZQ. Function Change-Length(*edited node, axis, new-length*)(see Sec. 2.4.4) is called which returns B.

3. If B = “true”,

    - The *edited node* is put in the corresponding *axis* queue, which is then set as the current queue.

    Else,

    - Exit.

4. If all queues are empty,

    - Exit.

    If the *current queue* is empty,

    - The other queues are searched.
    - The first non-empty queue is made the *current queue.*

5. The head of the *current queue* is removed from the queue and set as the current *Base node.*

6. If the *Base node's Visited* variable is “false”,

    - Non-redundant cycles of length more than two which contain the *Base node* are found and listed.

    Else,

    - Go to step 4.

7. The user is allowed to specify the order in which the listed cycles are to be processed.

8. For the selected cycle, the user is prompted to select a suitable *modification node.* The *Modify* variable of that node is set to “true” (If the selected node has its *Resolved* variable or *Visited* variable set to “true”, an error message is given and the user is asked to select another *modification node*).

9. Set the cycle traversal direction to be “forward” (i.e. the direction in which the nodes in the cycle are listed). The node next to the *Base node* is made the *current node.*

10. If the *current node* is the *Base node*,

- Go to step 14.

Else, if the *current node* is the *modification node* and if one of the planes in the direction of *axis* have *Fixed* variable value as "true",

- Go to step 12.

else,

- Partially resolve the *current node* by setting *Fixed* variable of the appropriate plane (based on available topological link information) to "true" in the direction of the *axis*.
- Go to step 11.

Else, If the *current node* has its *resolved* variable set to "true",

- An error message is given,
- Exit.

Else,

- The *current node* is resolved.
- *Resolved* variable of the *current node* is set to "true".
- The *current node* is put in the *current queue*.
- The next node in the direction of the traversal is made the *current node*
- Go to step 10.

11. Set the traversal direction to be "backward" (i.e. the direction opposite in which the nodes in the cycle are listed) and go to step 10.
12. The node that is marked *Modify* is *updated*, and its *Resolved* variable is set to "true".
13. Function Change-Length(*modified node, axis, new-length*) (see Sec. 2.4.4) is called which returns C.
    If C = "true",

    - Go to step 14.

    Else,

- Exit.

14. For all adjacent nodes to the *Base node* that are not part of a cycle containing the *Base node* and have a *Resolved* variable value of "false":

    - The topological links are imposed to resolve the node without changing its length.
    - the *Fixed* variables are set to "true"
    - the node is *updated*
    - *Resolved* variable is set to "true"

15. The *Base node's Visited* variable is set to "true"; go to step 4.

2.4.4 Change-Length(*Node N, Axis A, New-Length L)*

- Returns "true" if the length in *N* is modified to *L*.
- Returns "false" if the editing process needs to be aborted.

1. Length of the block represented by the node *N* along the given axis *A* is set as *Length OL*.

2. If $OL = L$;

    - Return "false".
    - Exit.

    Else,

    - Go to step 3.

3. List of the lengths that are linked with *OL* is set as *LenList*

4. First length in the *LenList* is set as *current length.*

5. If the *current length* has *Fixed* variable value "true",

    - An error message is given.
    - Return "false".
    - Exit.

Else,

- Set length next to the *current length* as the *current length.*
- If the *current length* is NULL,
  - Go to step 5.

  Else,
  - Go to step 5.
- First length in the *LenList* is set as *current length.* Node represented by the block corresponding to the *current length* is put in the queue of the type given by the axis of the *current length.*
- Return “true”.

2.4.5 Display

It is generally recognized that display functionality should have following characteristics :

1. Intuitive representation
2. Unambiguity
3. Realism
4. Computationally efficient

The development of CAD has seen substantial changes in the way visual data is represented. Initially, 2D wireframe and 3D wire frames were the dominant ways to display the model. These serve the purpose of presenting recognizable shape and require less computations but lack realization of volumetric shape properties. Hidden line and hidden surface algorithms have been applied to give “solidness” to the objects. Nowadays, technology is advancing to photo realistic images containing texture, and lighting/shadow representation using ray tracing techniques.

The current implementation supports the display of components in three modes. The default mode is called blockwise display, which is basically a wireframe display of each block. The second mode is a polygon set display which is specific to this implementation and shows blocks as solids. The third option is a true wireframe display of the object.

Blockwise display

In this type of display, each block is displayed as if it is a separate entity. The object oriented methodology is especially advantageous for this display functionality. When the user wishes to see component,the blockwise display of the component is accomplished by simply traversing the list of blocks and drawing the wire frame display of each block in the list. In effect, the component instructs each block in the list to display itself. The component does not need to know the type or the actual shape of each block. This is achieved by the use of virtual functions. *Block* is the super class which defines the display functionality that will draw the external shape, which is of rectangular geometry. Other subtypes of blocks are derived either directly or indirectly from *Block* and they define their own display routine which replaces the display routine of the superclass.

The application generates the data required for PHIGS primitives and performs the drawing using the polylines primitive in PHIGS. Special care is taken to invert the vertex list ordering so as to adjust face orientations when subtractive shapes are being displayed.

Blockwise display is the preferred choice for use during the creation and editing processes, due to the fact that the individual blocks and the type of topological relationships between them are clearly visible, (Figure 2.21).

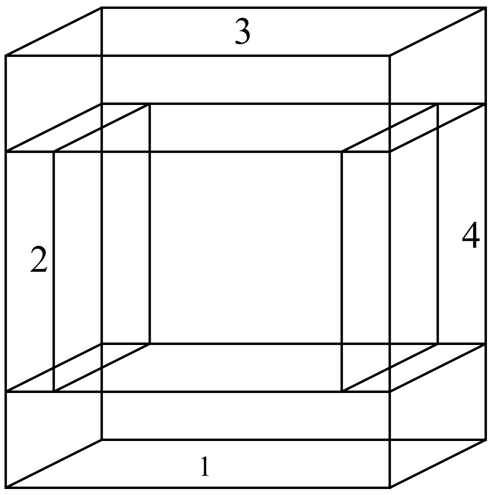


Figure 2.21: Block-wise Display

Polygon set display

In an assembly of blocks, it is usually unrealistic to see the common edges being displayed. This gives feeling that the two blocks are separate, which in reality is not the case. Therefore, a method was developed to remove the common line segments shared by the blocks. The component is then displayed as set of polygon-sets.

The whole process is broken down into four major steps :

1. *Creation of Polygons*

   Each block can generate faces which correspond to its external or internal shape. A rectangular shape generates polygons having four vertices a linked list. For a cylindrical shape, the top and bottom circular faces are approximated by twelve sided polygons and the curved surface is approximated by twelve rectangles.

2. *Formation of polygon sets*

   Class *component* constructs a list of polygons sorted according to the value to the normal of the polygons in each set. All polygons that are either coplanar or parallel to each other are grouped together. We need to detect those which are coplanar and combine them into a polygon set.

3. *Common segment detection and Polygon Merging* If the edges of two coplanar polygons share a common segment (Figure 2.22) then the two polygons are joined by listing vertices as follows. The edge of the first polygon is denoted by two vertices V_1 and V_2 and the edge of the second polygon which is common to the edge in the first polygon is denoted by P_1 and P_2 (Fig. 2.22). Starting with V_2, instead of traversing to V_2 as before, now it traverses to P_1. Then all the vertices of the second polygon are listed till the traversal comes to the second common edge point i.e. P_2. Then the listing switches back to the first polygon and continues to V_1.

   Display of the polygon sets

   After generating all the polygon sets, each set is displayed by traversing the list of polygon sets. The display is done in the Hidden Line/Hidden Surface Removal (HLHSR) mode in PHIGS (Figure 2.23).

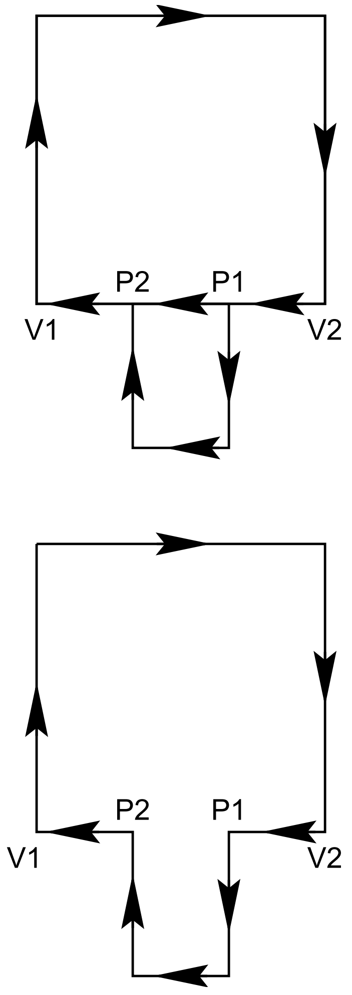


Figure 2.22: Common Edge Detection

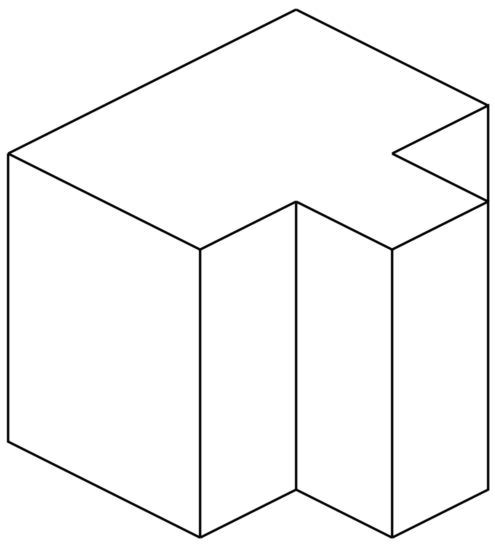


Figure 2.23: Polygon Set Display

2.4.6 Polygon-Set Display Algorithm

1. The list of blocks in the component is traversed. The polygons contributed by each block are collected.

2. The polygons are grouped together after sorting by the value of the normal to the polygon plane.

3. These groups of the polygons are further sorted in such a way that all co-planar polygons go to a single polygon set.

4. The first polygon-set in the list of polygon-sets is set as the current polygon-set.

5. If the current polygon-set is empty; go to step 9. Else, the head of the list of polygons in the current polygon-set is set as the current polygon.

6. The polygon next to the current polygon in the current polygon-set is set as the second polygon.

7. Algo. 2.4.9 is called with arguments as the current polygon and the second polygon.

8. If the value returned from Algo. 2.4.9 is NULL,

    - the second polygon is set as the current polygon,

- the polygon next to the second polygon is set as the second polygon.

Else,

- the second polygon is removed from the current polygon-set,
- the polygon next to the second polygon is set as the current polygon.

Step 7 is repeated till the last polygon in the current polygon-set becomes the current polygon.

9. If the polygon-set next to the current polygon-set is NULL,

   - Go to step 10.

   Else,

   - The polygon-set next to the current polygon-set is set as the current polygon-set
   - Go to step 5.

10. All the polygon-sets are traversed and all the polygons in each polygon-set are displayed.

2.4.7 Common Edge Detection Algorithm

- Input :
  - An Edge $E1(V_1, V_2)$ of the *first polygon.*
  - The second polygon.
- Output:
  - If there is a common edge, the vertex P of the second polygon to be traversed next from V_1 is returned.
  - If there is no common edge, a value of NULL is returned.

1. The first edge of the second polygon is made the *current edge* and is denoted by $E2(P_1, P_2)$.
2. Call $B1 = In - line(P_1, V_1, V_2)$, where $B1$ is a boolean variable. $B1$ has value "true" if P_1 lies between V_1 and V_2.

3. Call $B2 = In - line(P_2, V_1, V_2)$, where $B2$ is a boolean variable. $B2$ has value "true" if P_2 lies between V_1 and V_2.

4. Call $B3 = In - line(V_1, P_1, P_2)$, where $B3$ is a boolean variable. $B3$ has value "true" if V_1 lies between P_1 and P_2.

5. Call $B4 = In - line(V_2, P_1, P_2)$, where $B4$ is a boolean variable. $B4$ has value "true" if V_2 lies between P_1 and P_2.

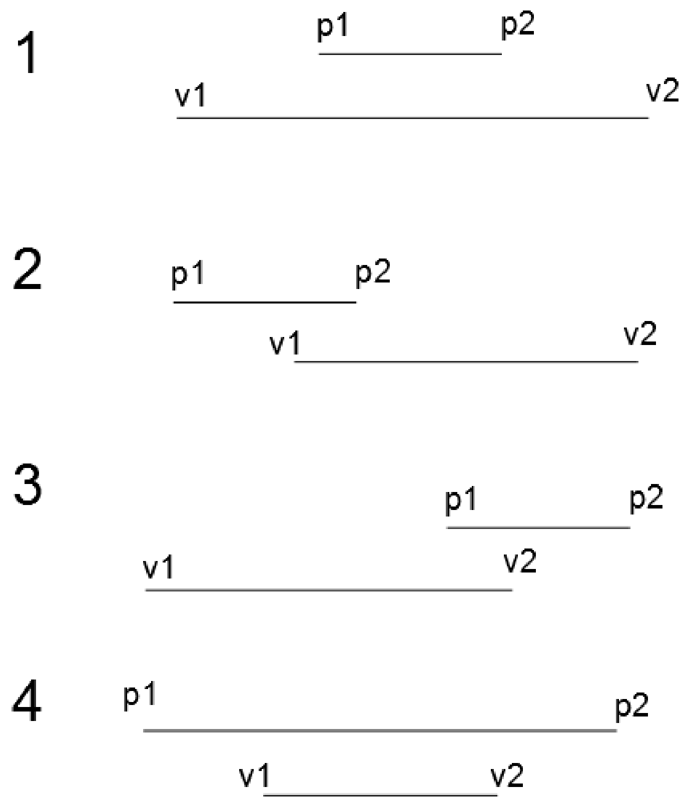


Figure 2.24: The possible ways of common edge

6. If both $B1$ and $B2$ are "true" then the point (P_1 or P_2) which is closer to V_1 is stored in list *Plist*(Fig. 2.24(1)).

7. If either of $B1$ and $B2$ is "true" then the point which can be reached from V_1 without going over the common edge is added to *Plist*(Fig. 2.24(2) and 2.24(3)).

8. If both $B3$ and $B4$ are "true" then the point which is closest to V_1 than V_2 is added to *Plist* (Fig. 2.24(4)).

9. The next edge of the second in the direction of traversal is made the *current edge*. If the *current edge* is not the first edge in the *second polygon*, steps 1 to 9 are repeated.

10. The point P in *Plist* which is closest to V_1 and can be reached without going over a common segment between the two polygons is returned. If *Plist* is empty, return NULL.

2.4.8 Algorithm for function In-line(edge $E1(V_1, V_2)$, vertex P_1)

- Output:
  - "true" if P_1 lies between V_1 and V_2
  - "false" if P_1 does not lie between V_1 and V_2.

1. Compute distance $D1$ between P_1 and V_1
2. Compute distance $D2$ between P_1 and V_2
3. Compute distance $D3$ between V_1 and V_2
4. If $D3 = D1 + D2$,
   - return "true".

   Else,
   - return "false".

2.4.9 Merging of the polygons

- Input : Polygons P_1, P_2.
- Output: A merged polygon P_1, or NULL.

1. Select the first vertex V_1 in P_1 as the current vertex. Set P_1 as the current polygon, P_2 as the second polygon. Define a new polygon P with V_1 as its first vertex. Set the traversal direction to Forward [1]. Set $NPOLY = 1$.

[1] "Forward" implies that the doubly linked list of vertices will be traversed following the "next" pointer; "Backward" implies that this list will be traversed following the "previous" pointers.

2. Select the edge of the current polygon emanating from the current vertex in the traversal direction as the current edge. Call Algo. 2.4.7 with arguments as the current edge and the second polygon and return value as V.

3. If V is NULL,

   - Set the vertex at the other end of the current edge to be the current vertex.
   - Add the current vertex to P.
   - Go to step 4.

   Else,

   - Add V to P.
   - Make the second polygon the current polygon.
   - Make the current polygon the second polygon.
   - Make V the current vertex.
   - Set the traversal direction to be such that the next edge traversed from V will not contain the common segment S.
   - Go to step 4.

4. If the current vertex is the starting vertex V_1,

   - Go to step 5.

   Else,

   - Go to step 2.

5. If $NPOLY = 1$,

   - $P1 = P$.
   - Return $P1$.

   Else,

   - Add P to the bottom of the polygon-set.
   - $NPOLY = NPOLY + 1$.

6. If P_1 contains any vertices not in P,
    - Set V_1 to be such a vertex.
    - Go to step 2

    Else, if P_2 contains any vertices not in P,
    - Set any such vertex as the current vertex.
    - Go to step 2

    Else,
    - Exit.

Chapter 3

Encapsulation

3.1 Introduction

The extendability of the feature library is based on the concept of *Encapsulation.* A large number of components that are manufactured start with the raw material as a rectangular block. The goal of process planning is to start from this rectangular shape and produce the final shape. *Orthohedral encapsulation* was thus a natural choice to start with. Bar stocks are also very common, but they usually restrict the final shape to parts with rotational geometry. This *Cylindrical Encapsulation* is not directly dealt with in the present work, but can be treated as a special case of orthohedral encapsulation.

This chapter describes how the concept of *Encapsulation* has been implemented in the current feature based design system. The process of extending the feature library is explained with an example. Finally, it is shown how components with free form surfaces can be modeled using the idea of encapsulated blocks in conjunction with a B-Spline surface.

3.2 Basics

Mechanical components come in a wide variety of shapes and sizes. To bring them all within the realm of solid modeling in exact form is not an easy task. Limitations imposed by the lack of an adequate mathematical description, computational power and human power makes it necessary to idealize them. The present system has implemented shapes with a relatively simple mathematical description. Still, it is not a simple matter to define operations and connectivities in terms of individual shapes. A layer is needed to hide aspects of the shape geometry which are not relevant to a particular operation and give only those that are necessary. This “hiding” of the true shape is called “Encapsulation”.

Encapsulation divides the real shape into two parts :

1. External shape, which gives an idealized external geometry which encloses the true shape.

2. Internal shape, which defines the actual geometry of the shape.

The definition of a mapping between the external shape and the internal shape is necessary for the definition of the encapsulated geometry. The encapsulated block may require some more parameters which may be requested from the user. The user needs to define operations for the encapsulated block which fall into the following categories :

1. *No-Change functions*, which are the same as the function developed for the external shape. e.g., addition or the deletion of the block from a component.

2. *New functions*, which are specific to the internal shape and are not relevant for the external shape, e.g., the user input of control points for a B-Spline surface.

3. *Overwritten functions*, which overwrite the corresponding external shape function completely, e.g., the display function.

4. *Add-on functions*, which add a function to the corresponding external shape function, e.g., the create function.

In our work, the external shape for all blocks is a rectangular parallelopiped with orthohedral geometry. The definition of internal shape is different for different encapsulated block types as described below :

1. Constructive block (ADDTblock)

   A constructive block feature (Figure 3.1) is a feature with six bounding faces where each face is perpendicular to one of the coordinate axes. In this case the internal shape is geometrically identical to the external shape. No additional parameters are necessary to define the geometry.

2. Subtractive block (SUBTblock)

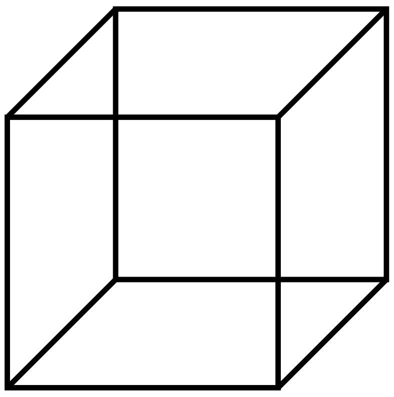


Figure 3.1: Constructive Block

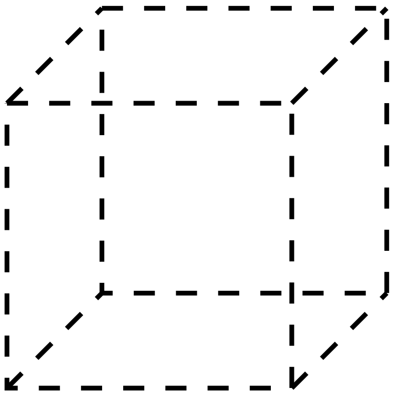


Figure 3.2: Subtractive block

A subtractive block (Figure 3.2) is geometrically identical to a constructive block, but physically it represents a negative volume. The order of the vertices in each face is the reverse of that in the faces of a constructive block.

3. Constructive cylinder (ADDTcyl)

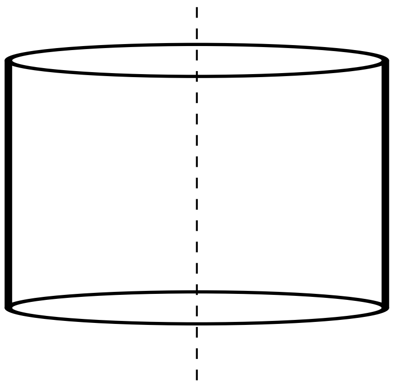


Figure 3.3: Constructive Cylinder

A constructive cylinder (Figure 3.3) is enclosed in an external rectangular block. The axis of the cylinder needs to be specified as an additional parameter. The lengths in the other two directions determine the major and minor diameters of the cylinder. If they are different, an elliptical cylinder is constructed.

4. Subtractive cylinder (SUBTcyl)

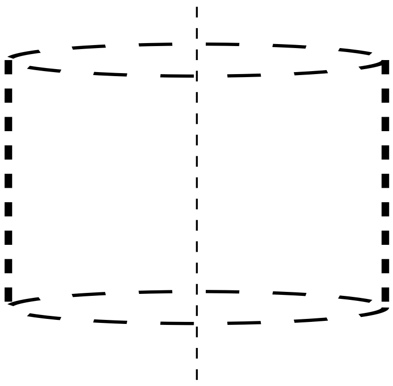


Figure 3.4: Subtractive Cylinder

A subtractive cylinder (Figure 3.4) is geometrically similar to a constructive cylinder. It differs only in the face orientations and represents a hole.

5. Constructive wedge (ADDTwedge)

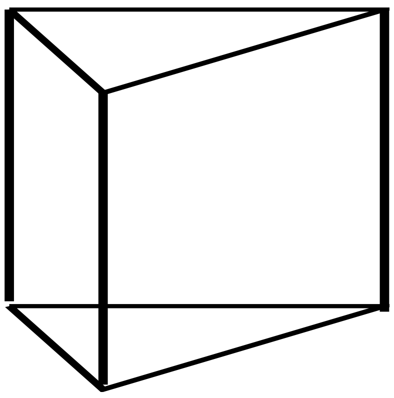


Figure 3.5: Constructive Wedge

A constructive wedge (Figure 3.5) is modeled as a right angled triangular prismatic block enclosed in the external shape. The user needs to specify the axis and the two butting planes.

6. Subtractive wedge (SUBTwedge)

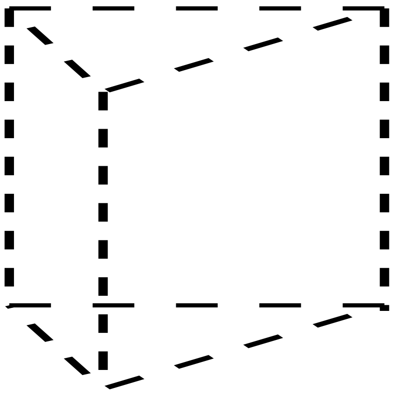


Figure 3.6: Subtractive Wedge

A subtractive wedge (Figure 3.6) is geometrically identical to the constructive wedge but differs in the orientation of the faces.

7. Constructive B-spline (ADDTbspline)

A constructive B-spline represents a block in which one surface is an extruded B-spline surface. Apart from the external block information, the user needs to specify an axis for extrusion and the control points to define the B-spline.

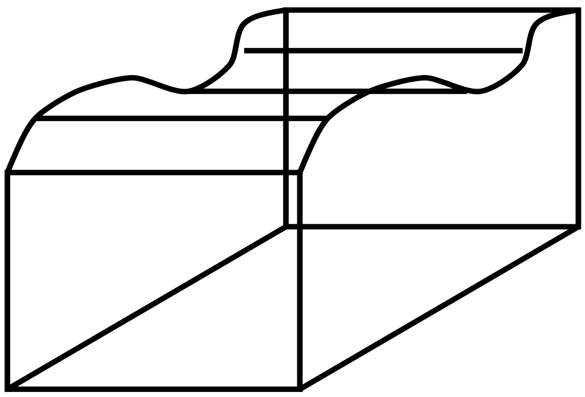


Figure 3.7: Constructive B-spline

3.3 Advantages of Encapsulation

1. It provides a mechanism to extend and customize the shape library
2. It allows handling of complex geometry while maintaining simplicity of the block structure.
3. It simplifies addition of a new shape as the functionality related to the external shape such as “Create”,“Edit”, etc. are already developed. The user needs to code only those functions which are related to internal geometry such as “local edit”, “polygon-set display”, etc.

3.4 Extension of the Feature Library

A rich feature library is a prime requirement of a feature-based modeling system. In addition, it should be possible to extend it depending on the family of parts to be modeled. This capability is provided in the current prototype feature based design system. To illustrate how this is done, the development of a right-angled wedge feature is presented in detail below. The feature is developed starting with the basic rectangular block which encapsulates it and going through the following steps :

1. Once the external rectangular block has been specified, the additional information required to define the wedge consists of :

- Axis of the wedge. The planes perpendicular to this axis are defined as the "top" and the "bottom" planes.
- Two butting planes. These are the rectangular faces of the wedge that are perpendicular to each other (Fig. 3.8).

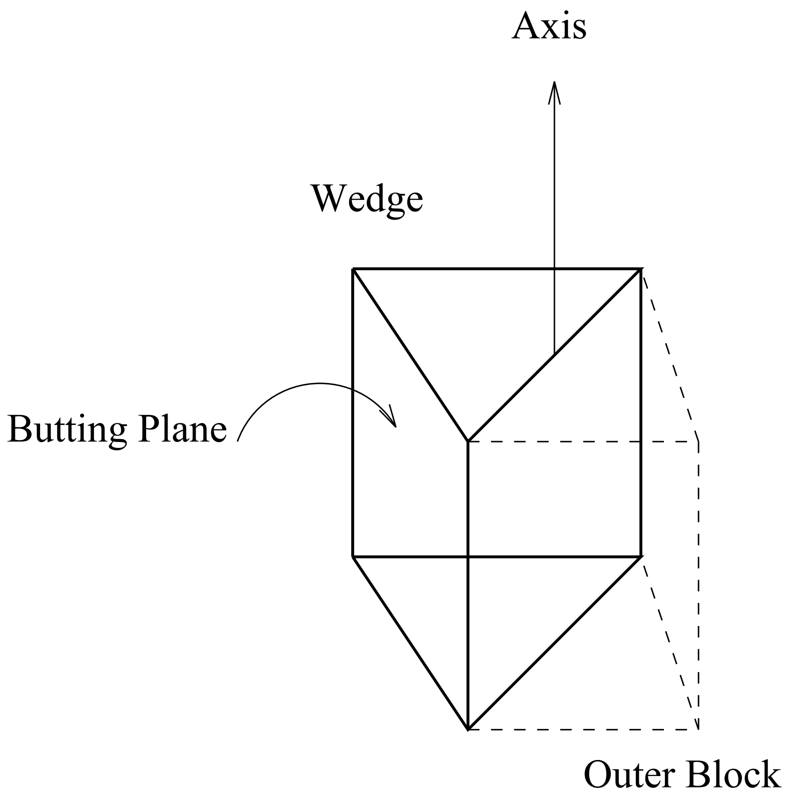


Figure 3.8: Constructive Wedge encapsulated in Block

2. Definition of Operations :

   - *Create-Polygons()* : This function returns a list of the polygons consisting of three side faces and two triangular faces representing the top and the bottom of the wedge. This function replaces the functionality developed for the external shape.
   - *Display()* : The *Display* function first calls the *Create-Polygons* function. Polygons that are displayed have two triangles (representing top and bottom) and three rectangular polygons (representing sides). This function overwrites the display functionality developed in the parent classes.

- *Create()* : The *Create* functionality adds the user input procedure for the additional data members defined for the wedge.
- *Add-Block()* : Addition of the wedge is the same as an addition of the external shape.

3.5 Swept B-spline Encapsulated Block

A B-Spline surface feature is provided to allow the user to model free-form surfaces. The idea of encapsulation is used to model this relatively complex shape, while still exploiting the simplicity offered by the generalized orthohedral block structure. Most of the functionality defined for the external shape remains valid for the B-spline block also. This reduces the number of operations that need to be implemented separately for the B-spline block. A local coordinate system is developed within the external shape and is used to define the mapping between the external rectangular shape and the internal B-Spline geometry. The mapping is defined as follows :

- The axis of the B-spline block is defined as the axis along which a B-spline curve is extruded (i.e. swept).
- The extrusion is confined between the MIN and the MAX of the external block along the axis of the B-spline.
- B-spline curve is defined in the plane normal to the axis of the B-spline block. For example, if the axis of the B-spline block is in the Z direction then the B-spline curve is defined in the X-Y plane.
- Definition of the B-spline curve requires following parameters :

  1. Order of the B-spline
  2. Knot vector
  3. Control Points

- The control points are defined in a local normalized coordinate system, in which coordinate varies from 0 to 1 over the block.

- The Bounds of the B-spline curve along the coordinate directions perpendicular to its axis correspond to the MIN and the MAX of the external block in the respective direction. For example, consider a B-spline block along the Z-axis. The x and y values of the control points range between 0 to 1, where “0” gets mapped to the MIN and “1” gets mapped to the MAX in both directions.

- As the B-spline surface is defined relative to the external shape, any change in the external shape gets reflected correspondingly to the absolute positioning of the control vertices; however their values in the local coordinate system remain unchanged.

3.5.1 Operations

- *Create()* : The Create functionality adds the user input procedure for the additional data members defined for the *B-spline.* The current implementation has limited the number of control points to twenty. The user first inputs the external shape data and the axis of the B-spline surface. The default control points are calculated internally as follows :

  - The user is prompted to choose the axis of the B-spline block.
  - A plane whose normal lies along the axis of the B-spline is selected for the curve definition. For example, if the user selects Z axis as the axis of the B-spline then Z-MAX plane is selected for the curve definition.
  - The MIN and the MAX values are retrieved from the external block geometry for the next direction (in cyclic order : X $\rightarrow$ Y $\rightarrow$ Z) to the axis of the B-spline block. Thus, in the example considered in the previous step, a default 2D curve is presented to the user with x values equally spaced between X-MIN and X-MAX, while y values are constant at Y-MID.
  - The user is allowed to change the values of these points to model the B-spline curve, after which extrusion is done to generate the B-spline surface.

- *Add-Block()* : Addition of a *B-spline* block is the same as the addition of the external shape. The B-spline does not change this function of a *Block.* It is assumed that the user, while specifying the topological link with the B-spline, chooses a plane surface of the B-spline block as the resting plane.

- *Edit()* : Modification of the external shape changes the overall size, shape, and location of the block in the component. The procedure for modification of the external shape parameters is not changed by the B-spline block. Thus, if the external shape is reduced/enlarged, the B-spline block also gets correspondingly reduced/enlarged. This is possible because the control points of the B-spline are defined relative to the shape of the external block. The user may be given an option of editing the B-spline block in the absolute coordinate system. The B-spline block adds functionality to the local edit, where modification of the positions of the control points is performed. The local edit of the B-spline surface is proposed as follows :
  - The user is presented with the list of control point values.
  - The user is allowed to change the coordinates.
  - It is assumed that the user does not change the values such that the surface interfaces with other blocks in the component; this can be done by placing all the control points inside the external shape.
- *Dimensioning* : The dimensioning scheme formulation requires contribution of the dimensioning planes from individual blocks. The dimensioning planes should completely specify the internal shape of the block. Refer section 4.5.5 for further details.
- *Display()* : The Display function overwrites the display function of the parent classes and displays the B-Spline surface. The NURB [1] surface primitive of PHIGS is used to display the B-spline surface. Although a polygon-set display facility has not yet been implemented, it would be done by generating a grid of points on the surface and approximating the surface by a set of rectangular faces on its grid.

[1]Non-Uniform Rational B-spline

Chapter 4

Dimensioning and Tolerancing

4.1 Introduction

Currently available CAD data models are adequate in terms of providing a unique and unambiguous representation of the geometry of a part, but these models only provide information about the nominal size of the part [20]. They do not support or reason about dimensioning and tolerancing (D & T) and other geometric inaccuracies. A new data representation scheme has been proposed here, based on feature-based principles which takes advantage of available product data structures and adds dimensioning capability to them.

A complete mapping of design features to manufacturing features has been the goal of several recent and current research projects in the field of feature-based design. The D & T capability plays an important role as it dictates how the part is to be made and to what specifications.

Previously, designers would design the part in such a way as to ensure proper functionality. The inclination was generally towards having tighter dimensional control. Manufacturing demands looser dimensional control because of the high cost of precision machining. Resolving these conflicting viewpoints requires a system which will design the part and perform manufacturing evaluation simultaneously so as to achieve a balanced solution.

The main aim of our research at Kansas State University is to develop such an integrated system. D & T has been developed to a large extent for components with rotational geometry. The present work tries to extend the dimensioning and tolerancing capability to generalized orthohedral components with encapsulated geometries.

4.2 Background

When CAD systems first emerged, the D & T methods that they incorporated were mostly drawing-based and needed human intervention for calculation, representation and interpretation. With advances in feature based design, efforts have been made to automate some parts of this process.

Computer aided analysis for D & T was introduced by TOLTECH (TOLerance TECHnology) [21], which addresses the problem of assigning tolerances so as to achieve minimum cost of manufacturing. It can also distribute tolerances and calculate the tolerance chains of an individual part.

Requicha [22] attempted to relate several representational issues in D & T in the CSG-based solid modeler, PADL-1. The object is created step by step with dimensional attributes built in and a dimensioned drawing was produced using the attribute information.

The major areas of interest in this field can be grouped into four categories :

1. Representation of D & T
2. Synthesis and analysis of D & T
3. Control of tolerances at different phases of Manufacturing
4. Implications of D & T in downstream CAM activities

The problem that is usually encountered in current CAD systems is a lack of support in the dimensioning and tolerancing capability. This is a serious deficiency because it implies that such systems cannot support many design and production activities that require the dimensioning and tolerancing information, e.g., fully automatic manufacturing [23].

Bing [3] proposed a new scheme which can represent variational geometry classes as well as the relationships between them. This is similar to a solid model that uses the geometrical entities to represent the nominal geometry and topological entities to represent the relationships. The nominal geometry is decomposed into some basic geometric primitives. The primitive is said to be resolved when its shape is defined from its locations. Bing has used GBBs (Geometric Building Blocks) as the basic entity to represent D & T information. The GBBs used are :

- Plane
- Line
- Point
- Feature of Size
- Curve
- Surface

Nominal features and GBBs are different. For example, for the feature cylinder, the GBBs are given by :

$$GBB = \{axis(Line)\&Radius(Featureofsize)\}$$

DOF (Degrees of Freedom) determines the number of independent parameters required to constrain a GBB fully in space.

- Translational DOF = x, y, z.
- Rotational DOF = α, β, γ.

Tolerance classes are derived based on DOFs and GBBs. Combinations of DOFs give new tolerance classes. Tolerance zones are created by applying the transformation matrices to nominal geometries.First, an FD graph (Dimension adjacency graph) is created. The user specifies the dimensions by picking the datum and target planes. Then, the arcs are developed. The FD graph transforms into an FDT (D & T adjacency graph) graph by filling in tolerance components $[(dx, dy, dz)$ and $(d\alpha, d\beta, d\gamma)]$ with corresponding tolerance values. Implicit dimensions (which can take any value) and tolerances can be overwritten if more than one edge occurs at a node.

Lin [24] proposed a data structure with the following characteristics :

1. Faces are represented by nodes
2. Relationship between faces is represented by arcs
3. The node at the top is a root (datum) node
4. All the nodes except the *root* have one and only one parent
5. Any loops in the structure represent redundant dimensioning

4.3 Deficiencies in current D & T data models

There are some major deficiencies in the representation scheme for *dimensioning and tolerancing* in currently available feature-based design systems. This section reviews their shortcomings with a few examples :

- It is a common practice in mechanical drawings to dimension co-planar surfaces with a single (common) dimension (see Fig. 4.1)

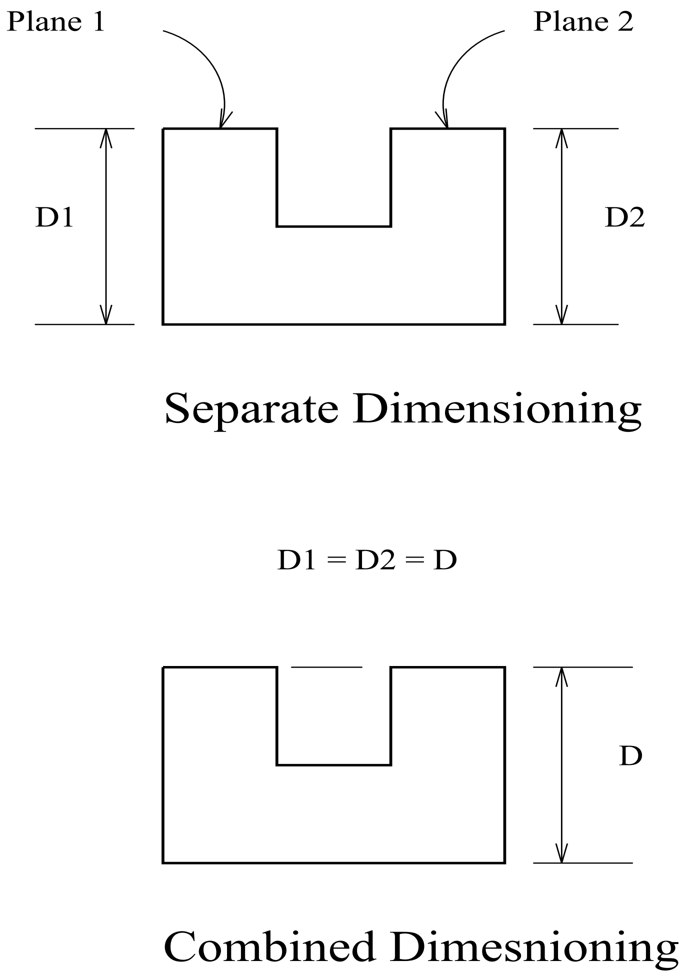


Figure 4.1: Common dimensioning for co-planer surfaces

Most of the current systems force the user to dimension plane 1 and plane 2 separately. In some cases, the user may want to dimension them separately, as he/she would like to specify different tolerances on the dimensions. The data model should provide a mechanism to combine as well as separate the dimension primitives depending on the user request.

- There could be more than one way of dimensioning a part (see Fig 4.2).

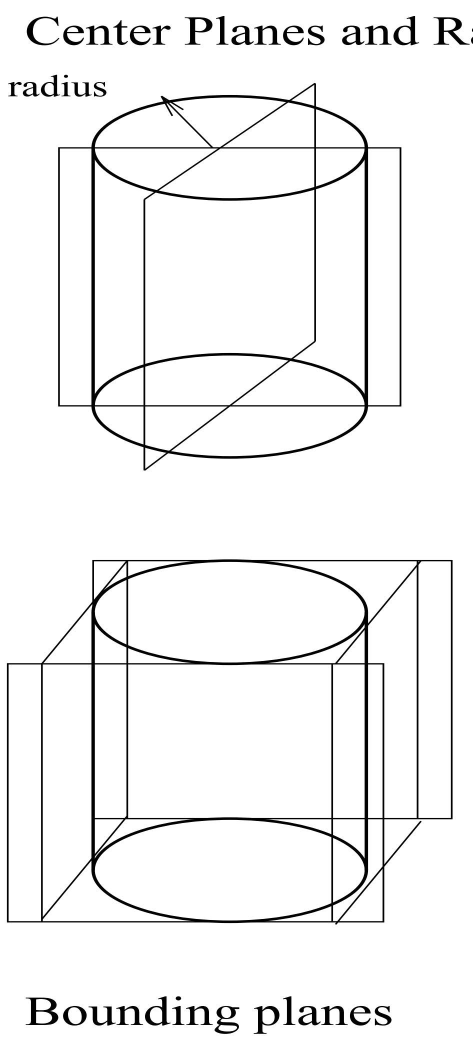


Figure 4.2: Different dimensioning schemes for cylinder

The dimensioning and tolerancing data model should have the capability to present all of them to the user and allow the user to make the choice. Most existing systems only allow one way to dimension a given feature.

- There is very little support for dimensioning free-form surfaces. The present system provides the capability to dimension individual surface points and also to group them with other dimensioning primitives if necessary (see Fig. 4.3).

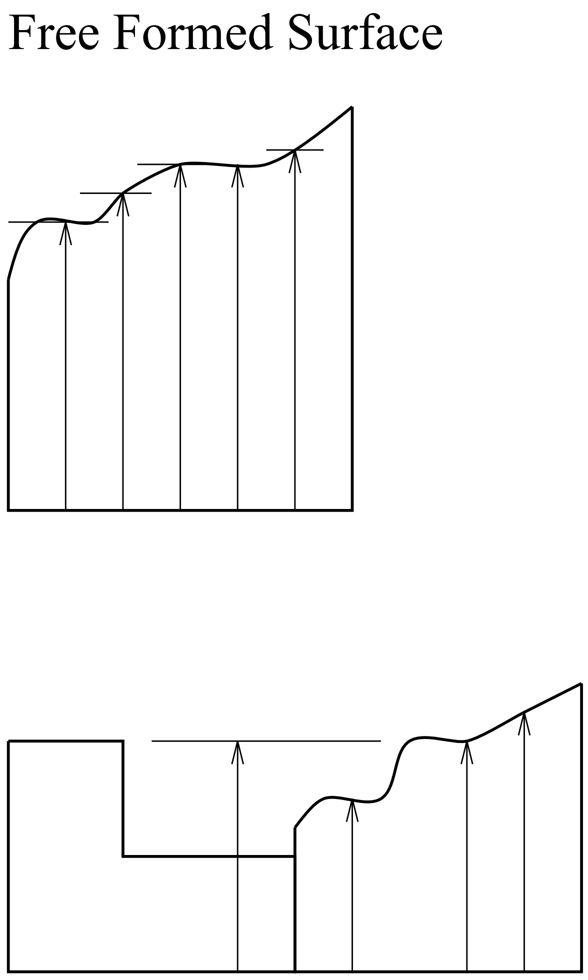


Figure 4.3: Dimensioning scheme for B-spline

4.4 Basic Concepts

The basic dimensioning primitives are :

- Dimension Planes
- Size dimensions

Most of the dimensioning is performed using dimensioning planes which are contributed by external as well as the internal shape. All dimensions are denoted as the distance between two dimension planes. Some dimensions like “radius” in the case of a cylinder are generally not modelled as a distance between two planes. For such entities, a primitive called *size dimension* is proposed. It is contributed by the internal shape.

The dimensioning information for a feature can be decomposed into the following two categories

1. External or locational dimensions

2. Internal or size dimensions

For example, the dimensioning information of a cylinder can be decomposed as follows :

- *Locational* : top and bottom planes, and two mutually perpendicular middle planes along the axis

- *Size* : radius of the cylinder

The idea of *external shape* and *internal shape* explained earlier matches this concept very well, as the locational dimensions are defined with respect to the external shape while the size dimensions are defined by the internal shape.

The external shape of each feature contributes dimensioning planes which fix its position in the global coordinate system. The internal shape contributes planes and / or parameters (which are specific to the internal geometry) to fix the internal shape with respect to the external shape.

4.5 Dimensioning and Tolerancing model

4.5.1 Graph Representation

A graph data structure is used to model dimensioning and tolerancing information. A graph contains nodes interconnected by arcs. The nodes in the D & T graph model represent one or more *dimensioning primitives* while arcs represent dimensioning and tolerancing information. Various operations required in the context of dimensioning and tolerancing can be effectively formulated in terms of graph algorithms For example, the detection of over-dimensioning corresponds to the detection of cycles in the graph.

4.5.2 The D & T Graph Model

The D & T model proposed here takes advantage of external encapsulation and internal shape geometry. Two graphs are used to represent the D & T information for the component :

1. Reference Dimension Graph (RDG)

2. Specified Dimension Graph (SDG)

The RDG is a relatively static data model and consists of a tree for each direction that has at least one plane normal to it. The tree is of depth one and has a single root. The normal directions are grouped into X, Y, Z, and "other". The root in each set acts as a datum plane with respect to which all the other planes (represented by leaf nodes in the same tree) are dimensioned. The RDG forms a base with the help of which dimensions between any two planes can be computed.

The SDG is a graph with the same nodes as the RDG, but its topology is specified by the user (see Fig. 4.4). The user never has to input the dimension while changing the SDG, those are computed internally with the help of the RDG.

The RDG and the SDG are created by default at the start (see Fig. 4.5) and are initially identical. The RDG remains unchanged in structure for the component as long as the component's topology is unchanged. It is hidden from the user, while the SDG is the one on which the user operates. The graph topology of the SDG may be changed by the user to suit the needs of the dimensioning scheme. All changes to the SDG are validated internally with the help of the RDG. Any changes to the block structure of the component such as the addition of a new block, will change both the RDG and the SDG. Geometric editing changes do not change the topology of these graphs but change the values of the dimensions stored in the dimension arcs.

There are separate RDGs and SDGs for each direction. The directions are X, Y, Z, and "Other". The "Other" direction set consists of all the directions which are not coordinate directions. is further sorted according to the value of the normal so that all the planes having the same normal will form one RDG and one SDG.

4.5.3 Data Structures

1. A dimension graph contains a list of dimensioning Nodes.

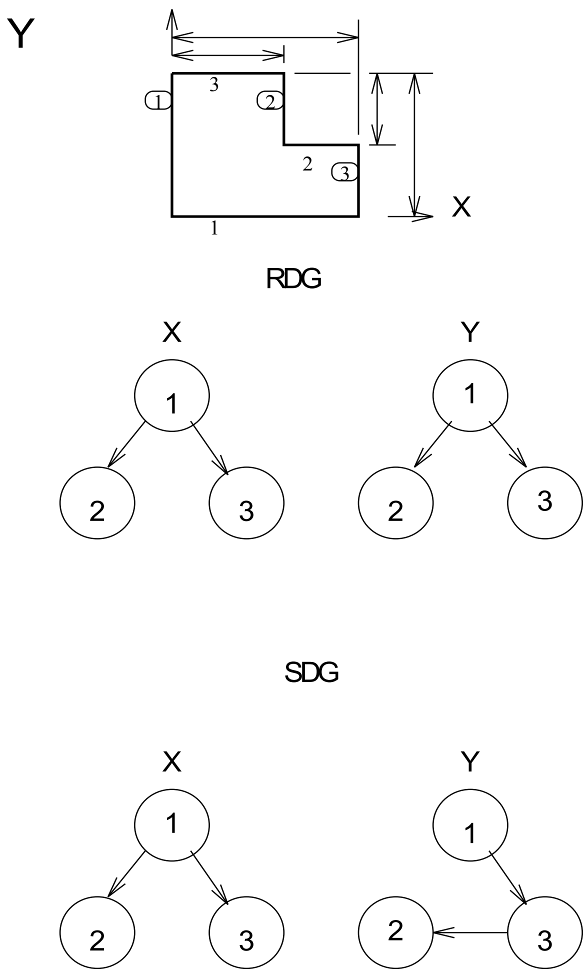


Figure 4.4: The RDG and the SDG

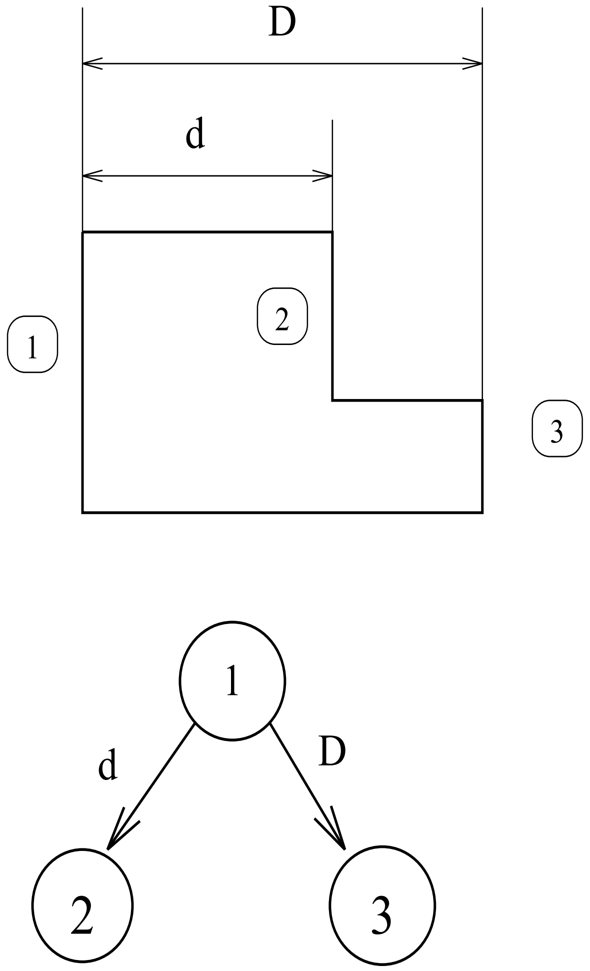


Figure 4.5: Dimension Graph

2. A dimensioning node contains information about the dimensioning primitive which it represents and also a list of dimensioning arcs going out from it.
3. A dimensioning primitive is the actual dimensioning entity stored in the dimensioning node.
4. A dimensioning arc connects two dimensioning nodes with a directional property, i.e. the source represents the reference primitive and the destination represents the target to which the dimension has been specified. Each arc stores information about dimension and tolerance values. Arcs in the RDG store reference dimension/ tolerance values while arcs in the SDG store user-specified dimension and tolerance values.
5. Each direction has a different RDG and SDG. The dimension primitives are first sorted according to the value of the normal and then contributed to the dimension graphs for that direction.
6. Tolerances are associated with each arc in the SDG. The RDG does not contain any tolerances.

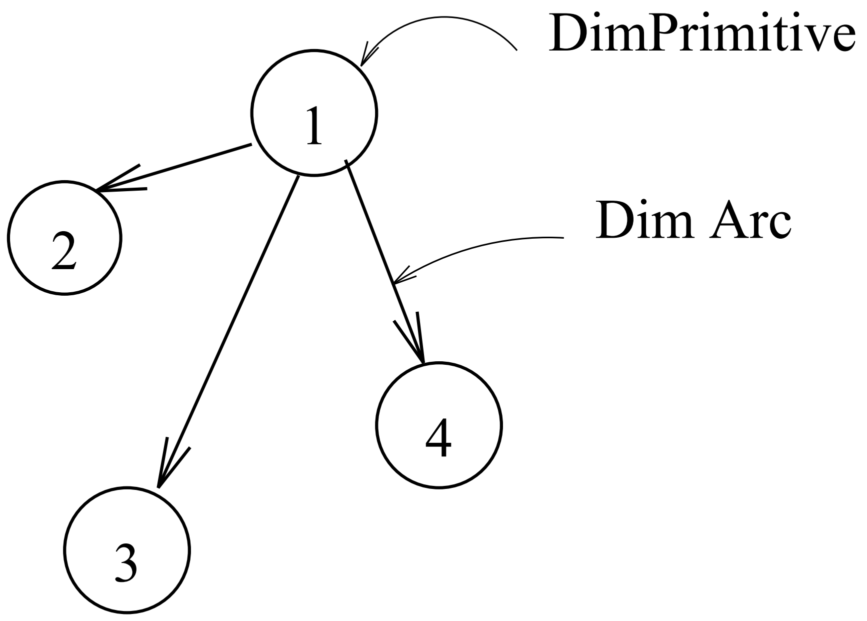


Figure 4.6: Graph Data Structure and Dimension Class

4.5.4 Operations

1. *Creation of the dimension graph*

   Depending on a block's internal shape, it contributes a preset number of dimensional primitives. Currently, only planes have been used as dimensional primitives. Each primitive gets mapped onto a dimensioning node and gets added to the dimension graph. While being added to the node list, the nodes are sorted according to axis and separate lists are maintained for each coordinate direction; planes that are not perpendicular to any coordinated axis are maintained in a fourth list. At the creation stage, the RDG and SDG have the same topological structure. In the RDG, the first dimension plane that is encountered acts as a datum dimension node and all the other planes are dimensioned with respect to the first plane. The fourth category is further sorted according to the value to the normal of the planes. Each direction has its own RDG and SDG. Angles between two planes are represented by a dimensioning arc between dimensioning nodes of two different graphs. For example, if an angle needs to be specified between a plane in the X direction and a plane in the Y direction, then an arc is set from the node representing the first dimension plane in dimension-graph-X to the node representing the second plane in dimension-graph-Y.

   The arcs store the value of the distance/angle between two primitives. For non-coordinate directions which have faces perpendicular to them, lists are maintained and processed in the same way as the coordinate axes lists. Initially, the SDG follows a datum dimensioning scheme. Once the initial SDG is complete, the user is free to make changes in its topology by redefining the connectivity while preserving all the nodes.

2. *Specifying the dimension between two primitives* :

   If the user wants to specify the dimensions between any two primitives, the following is the general procedure :

   (a) Select two primitives from the same block or different blocks. Currently, only planes that are parallel can be chosen.

   (b) Specify the dimension value and the tolerance.

   When this input is given, the two selected primitives are located in the RDG and are checked for connectedness to ensure that a path exists between them. If no such path is found, an error message is issued since the dimension cannot be specified. If a path is found, the value of the dimension is computed by adding (or subtracting) and is checked with the value specified by the user. If this matches, checks are made to see if the modification will lead to over-dimensioning or under- dimensioning (see item 3 below). If this check is passed, the SDG is modified while the RDG is kept unchanged.

3. *Change in Datum* : Datum, chain and mixed dimensioning are the three types of dimensioning schemes normally followed in specifying dimensions. In datum

   dimensioning, one reference plane is chosen and all the other primitives are dimensioned with respect to that plane. In the manufacturing view, this datum feature is made first and all the other features are manufactured or inspected with respect to the datum.

   In chain dimensioning, the reference plane is the next plane along that direction i.e. the destination primitive of the previous dimension becomes the reference primitive of the next dimension.

   Mixed dimensioning contains more than one datum primitive from which the rest of the primitives are dimensioned.

   The user may want to change the dimensioning scheme to any of the above mentioned dimensioning scheme. This is accomplished by changing to one or many datum planes in a single direction. The algorithm presented below describes the process of changing the datum plane for the case of datum dimensioning.

   - The block list and the plane list are presented to the user for selection of the datum plane.
   - Each of the remaining nodes in the appropriate SDG is checked for a path between itself and the new datum plane in the RDG. If found, an arc is set up between the new datum node and the second node under consideration. The dimension value is automatically computed by adding up the dimension values while traversing the path.
   - The process is repeated until all the nodes in the SDG are checked.

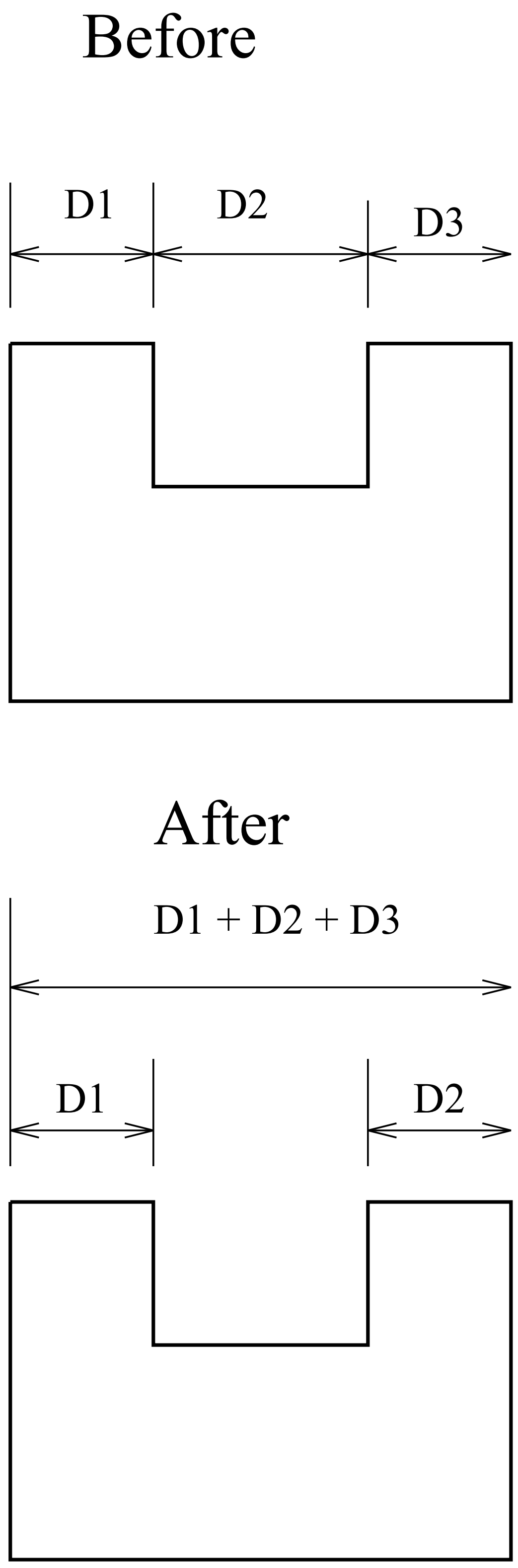


Figure 4.7: Specifying dimension between two planes

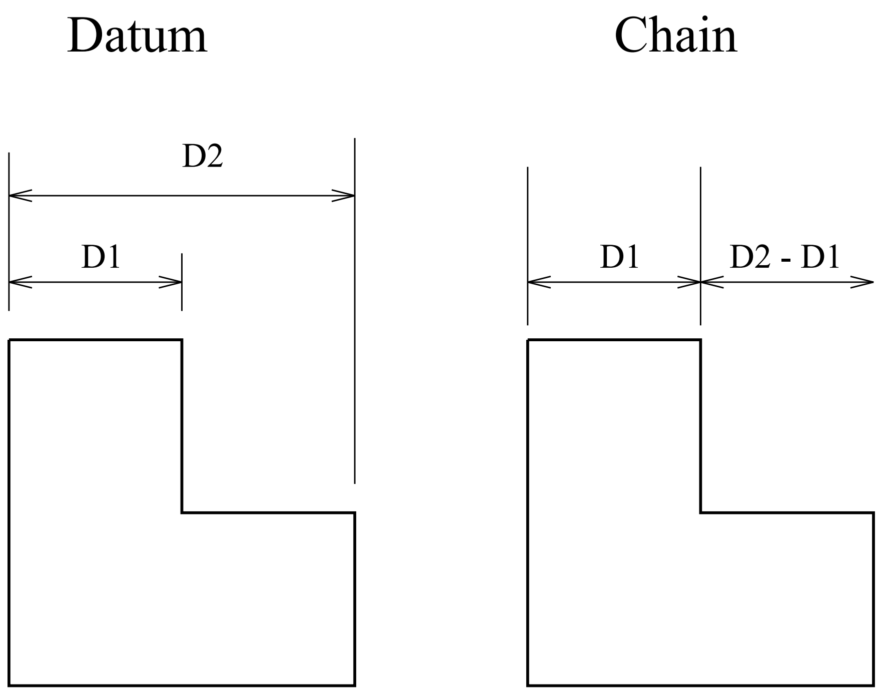


Figure 4.8: Change of Datum Plane

4. *Detection of missing or redundant dimensions* : When a user specifies a new dimension, care is taken to verify that this does not introduce redundancy into the dimensioning scheme. This is easily accomplished, because a cycle in the graph means there is a redundancy (see Fig. 4.9). Conversely, if there is no cycle, there is no redundancy. The existence of cycles can be easily checked by a graph algorithm [19]. Similarly, if any dimensions are missing, the graph becomes un-connected. This can also be easily detected using a suitable graph traversal algorithm such as a depth first search.

5. *Combination of primitives* : It has been common practice to dimension multiple coplanar faces with a single dimension. This facility is provided in our scheme by allowing more than one dimensioning plane to be combined into a single dimensioning primitive, i.e., all these planes jointly behave as a single dimensioning primitive. If the user needs to separate out a plane for any reason (such as dimensioning different tolerance values) the desired dimensioning planes can be delinked from the dimensioning primitive and a new dimensioning primitive corresponding to this plane can be created.

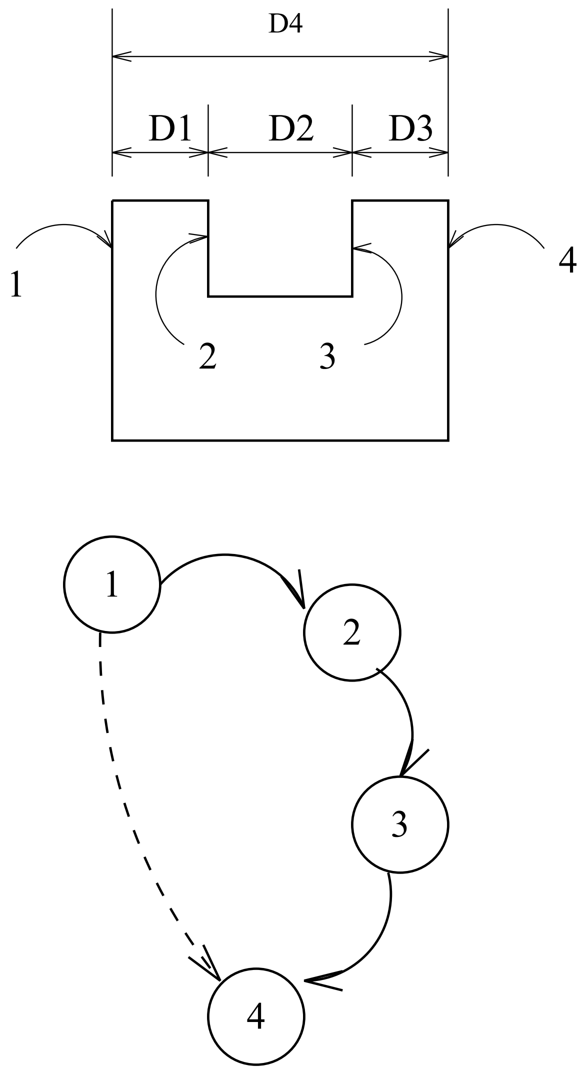


Figure 4.9: Over-dimensioning

6. *Query of Dimension* : If the user wants to query the distance between any two primitives, this option can be used. The user must first specify the two blocks and the two plane primitives. A graph traversal algorithm first locates the two primitives in the SDG, finds the unique path between them and computes the distance by summing the dimension attributes on the arcs that make up the path. Tolerance stacking is also determined in this way.

4.5.5 B-spline Dimensioning

Formation of the RDG and the SDG requires contribution of the dimensioning planes from individual blocks. These planes can be used to specify a dimension with any other dimensioning plane. It is necessary that the B-spline surface provides not only the dimensioning planes corresponding to the external rectangular block but also the dimensioning planes corresponding to selected points on the profile.

The dimensioning procedure is as follows :

- A two-dimensional view in the plane normal to the axis of the B-spline block is presented to the user. For example, if the axis of the B-spline is Z axis then the X-Y plane view is presented.
- For pre-specified intervals (say n), values x_1 to x_n are generated at equal intervals, where x_1 corresponds to the Xmin plane and x_n corresponds to the Xmax plane.
- For each value of x_i a corresponding y_i is calculated from the profile equation.
- For each i, a plane normal to Y direction passing through x_i and a plane normal to the X direction passing through y_i are contributed to the dimensioning procedure which is dealt with in detail in section 4.5.5.
- These dimensioning planes are contributed along with the side faces of the B-spline block.
- When the RDG and the default SDG are generated, all the planes are automatically included.
- The user is allowed to change the topology of the SDG as explained earlier.

4.6 Example of Basic Dimensioning

The example component is shown in Fig. 4.10 while Fig. 4.11 depicts how a component gets dimensioned by default. Internally, the dimensioning scheme uses the graph data structures for reference dimension graph(RDG) and specified dimension graph(SDG) which are also shown in the Fig. 4.11.

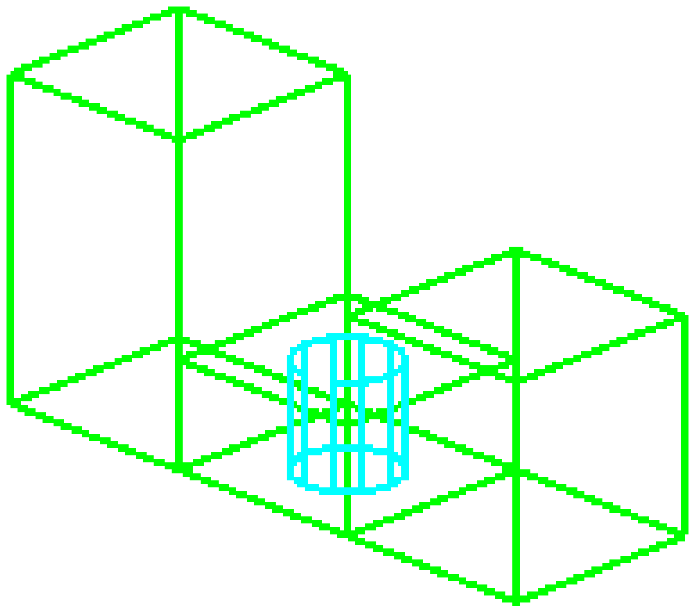


Figure 4.10: Example Component

The Fig. 4.11 shows default dimensioning scheme which is a datum dimensioning scheme along X axis.

Operations performed by the user to change the dimensioning scheme are reflected only in the specified dimension graph while the reference dimension graph remains unchanged. The operations are listed below.

1. *Specifying the dimension between two planes* :

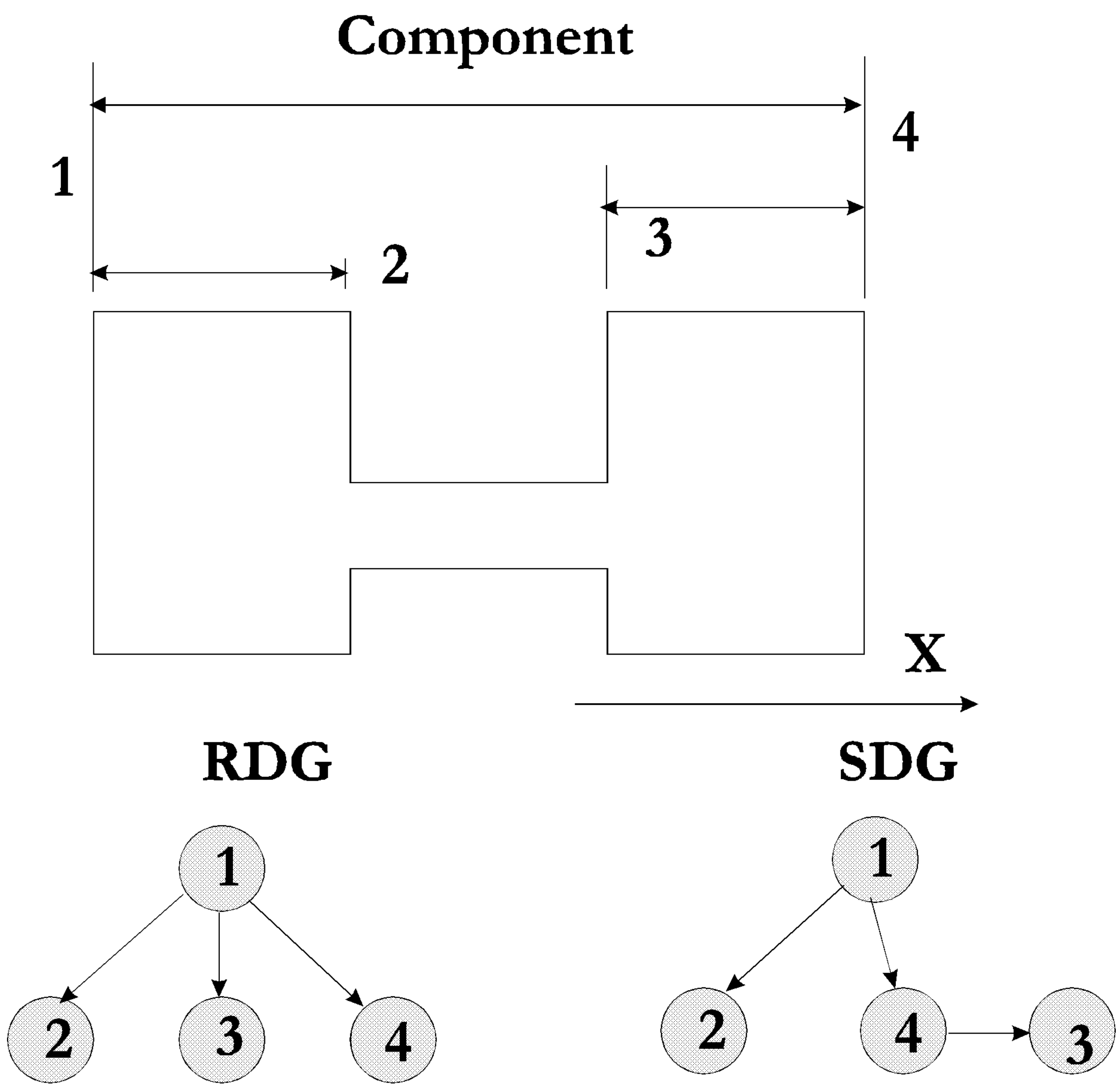


Figure 4.11: Dimensioning the component

If the user wants to specify the dimension “e” between (say) plane 6 and plane 7, following actions are performed :

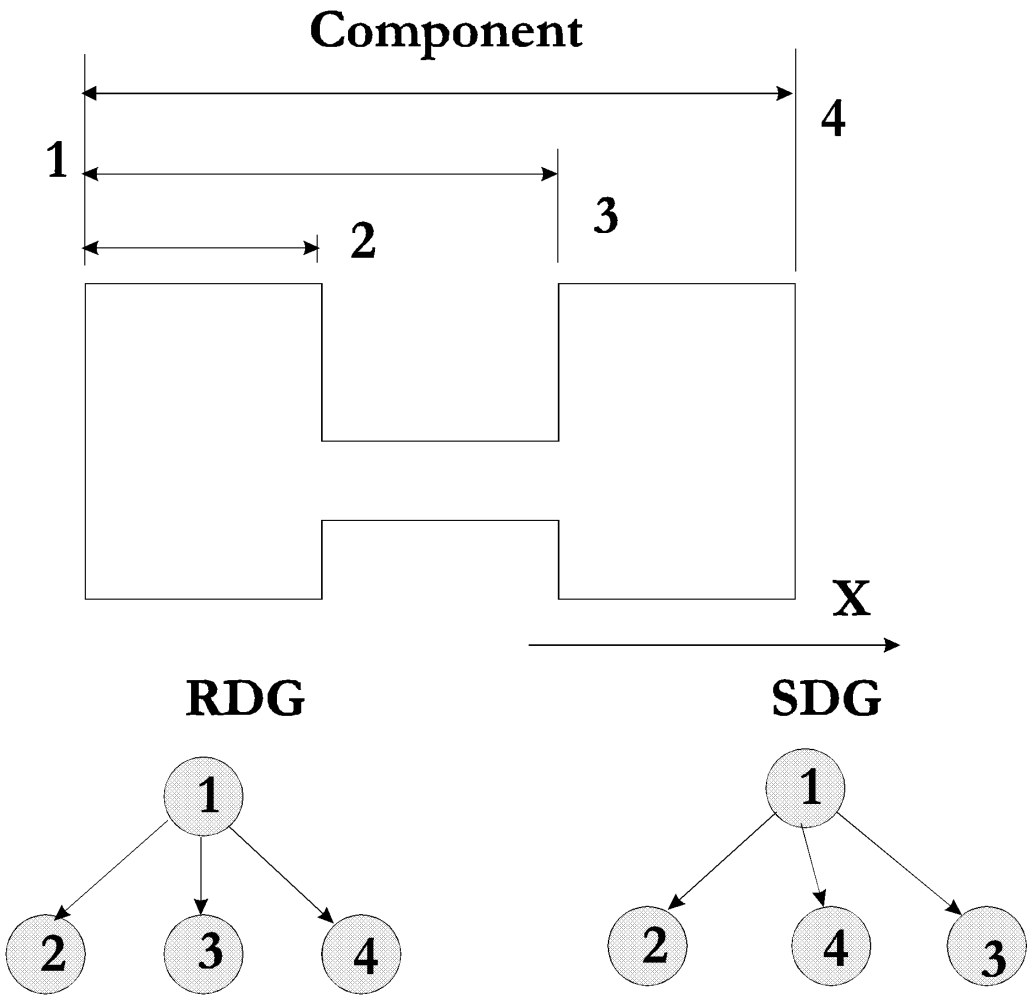


Figure 4.12: Specifying the dimension between two planes

When the input is given, the two selected primitives are located in the RDG and are checked for connectedness to ensure that a path exists between them. If no such path is found, an error message is issued since the dimension cannot be specified. If a path is found, the value of the dimension is computed by adding (or subtracting) and is checked with the value specified by the user. If this matches, checks are made to see if the modification will lead redundancy in the dimensioning. If this check is passed, the SDG is modified while the RDG is kept unchanged. The modified SDG is shown in Fig. 4.12, where a new arc is added between nodes 6 and 7 and the arc between nodes 1 and 7 is deleted as it would have created redundancy represented by a cycle.

2. *Change in Datum* :

   The user may want to change the datum plane to plane 7 in the mentioned dimensioning scheme so that all the other planes are dimensioned with plane 7 as datum plane. This is accomplished by removing all arcs emanating from previous datum plane 1 and making them emanate from the new datum plane 7 in the specified dimension graph. The modified SDG is shown in Fig 4.13.

3. *Query of Dimension* :

   If the user wants to query the distance between (say) plane 2 and plane 7. A graph traversal algorithm first locates the two primitives in the SDG, finds the unique path between them and computes the distance by summing the dimension attributes on the arcs that make up the path. The computed distance is displayed to the user.

4.7 Bspline Dimensioning

The following example of a vice demonstrates the capability of dimensioning the Bspline extruded block. The vice consists of two rectangular blocks and a Bspline extruded block. Formation of the RDG and the SDG requires contribution of the dimensioning planes from individual blocks. These planes can be used to specify a dimension with any other dimensioning plane. It is necessary that the B-spline extruded block provides not only the dimensioning planes corresponding to the external rectangular block but also the dimensioning planes corresponding to selected points on the profile of the Bspline.

The example shows dimensions along X axis (see Fig. 4.15).

The dimensioning procedure is as follows :

- For pre-specified intervals (say n), values x_1 to x_n are generated at equal intervals, where x_1 corresponds to the Xmin plane and x_n corresponds to the Xmax plane. Secant method is used to solve the non linear Bspline equation to get the parameter value for each co-ordinate point x_i.

- For each parameter value corresponding to x_i a corresponding y_i is calculated from the profile equation.

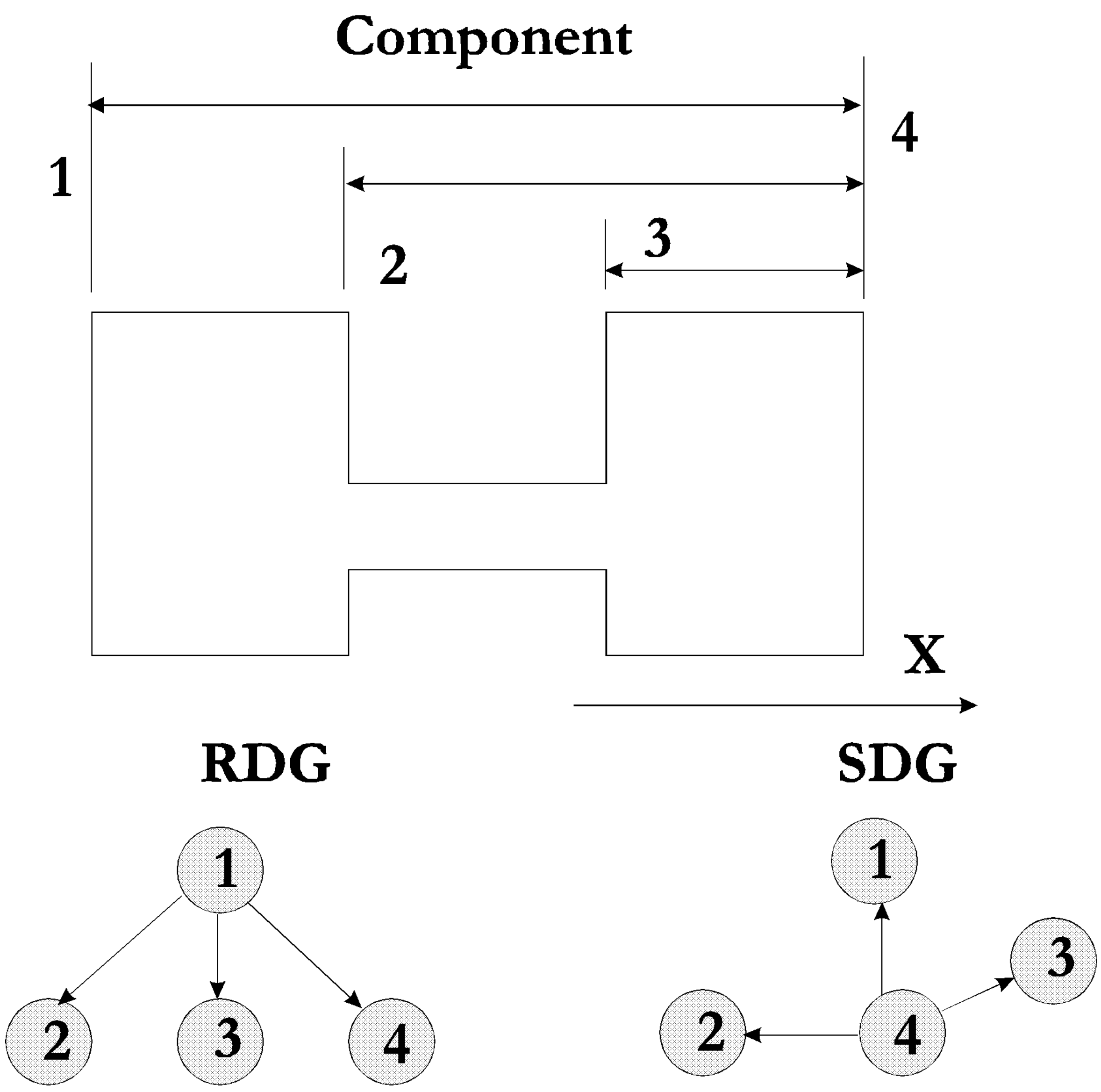


Figure 4.13: Change in Datum

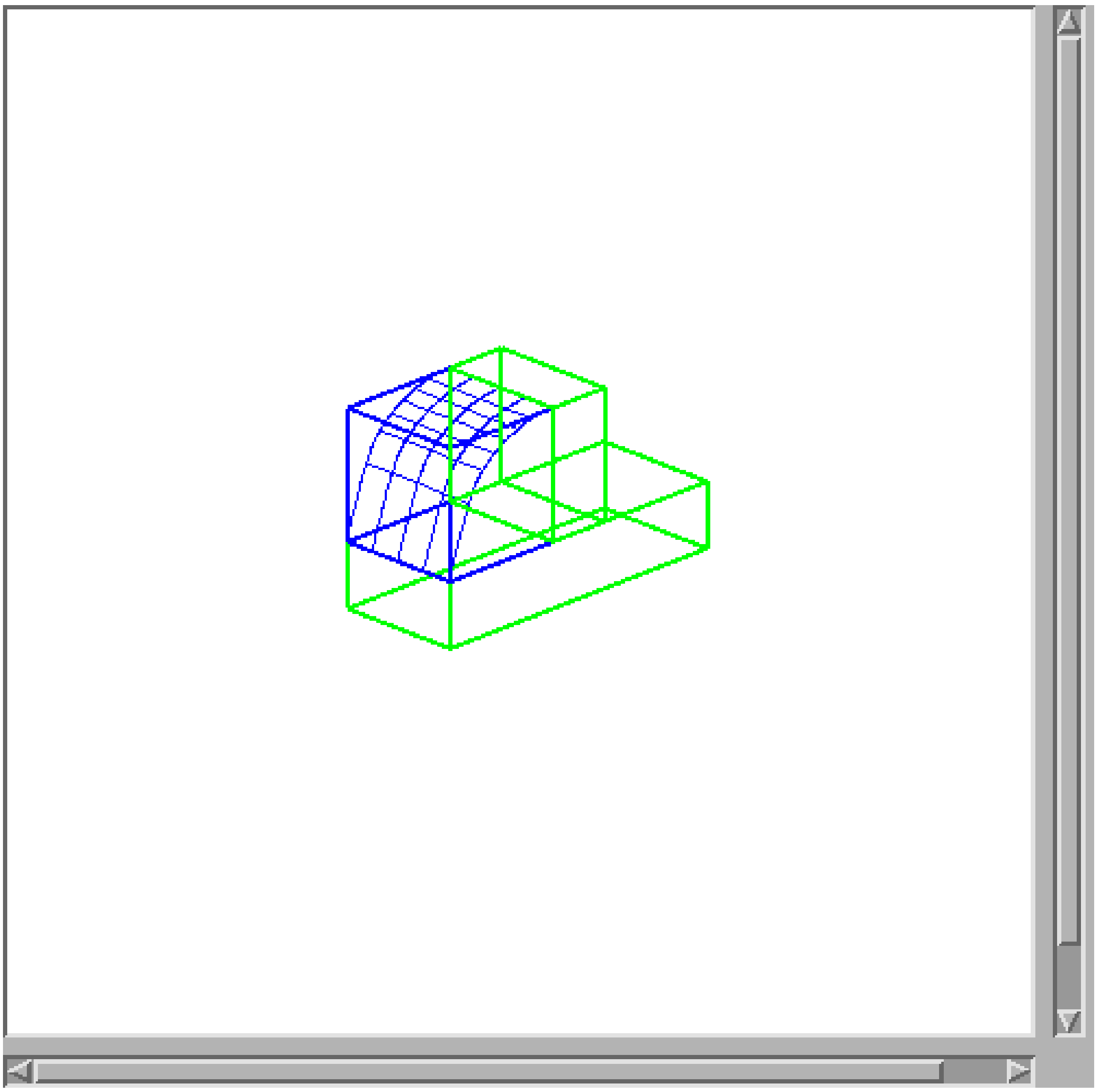


Figure 4.14: Example Component

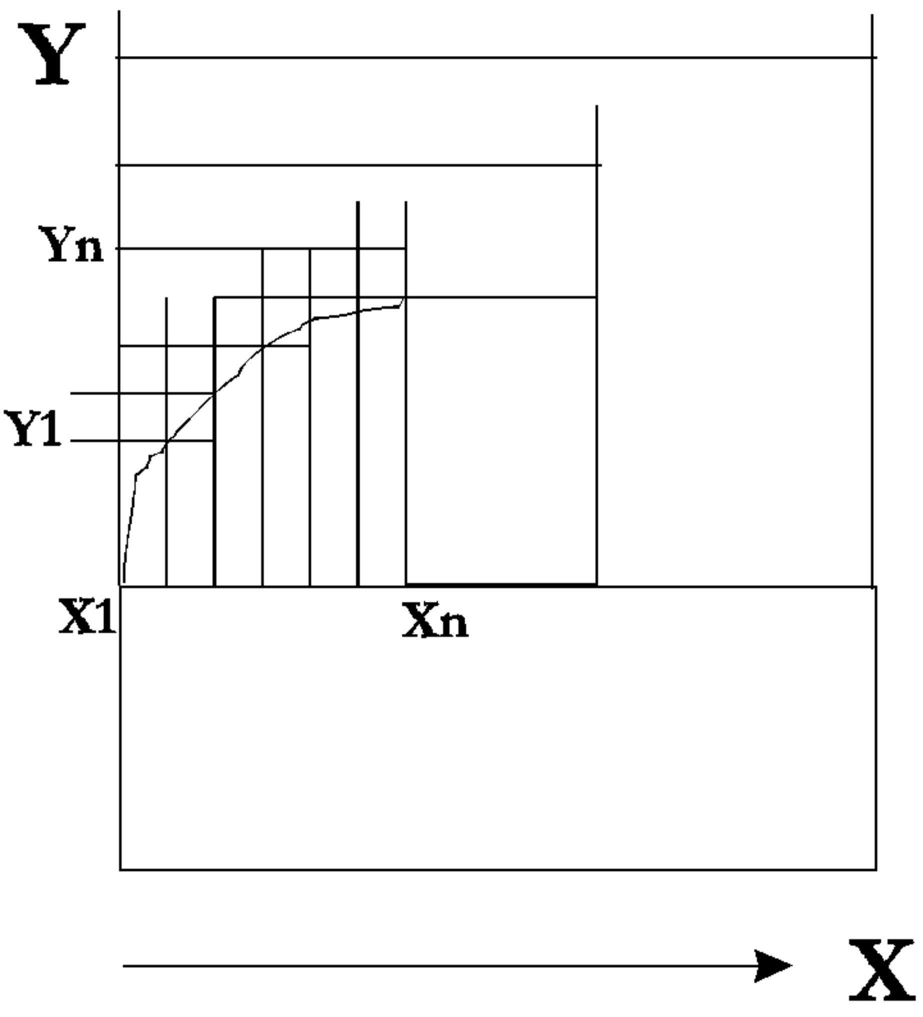


Figure 4.15: Bspline Dimensioning

- For each i, a plane normal to Y direction passing through x_i and a plane normal to the X direction passing through y_i are contributed to the dimensioning procedure.
- At the dimension graph level, these contributed planes are used dimension planes in the dimension scheme which is dealt with in detail in section 4.5.5.

Chapter 5

Implementation

5.1 Introduction

The objectives of this thesis include not only the development of an FBD approach for generalized orthohedral components, but also its implementation using an object-oriented programming approach.

Implementation provides a platform to test, verify and validate the concepts and theory.

This chapter is partitioned into two major units :

- *Object Oriented Implementation* : This unit deals with the use of object-oriented principles as applied to the current research.
- *Graphical User Interface* : This unit deals with the implementation of the user interface using X-Windows Motif toolkit and PHIGS.

5.2 Object Oriented Implementation

The choice of programming paradigm dictates the expressibility and the ease with which the concepts can be implemented. The object-oriented approach is very well suited for feature-based design implementation, due to the natural correspondence between design features and object classes. Object-oriented implementation also provides the capability to create application-specific, customized data models to extend/alter the current capabilities of the system.

More specific and complex shapes are derived from the more generalized ones. The data and operations specific to a type of block are contained in the class that represents that type of block. The functionality specific to the derived shape gets added as a specialization. All the classes know how to perform all necessary operations on themselves. Another major benefit lies in the re-usability and extendability of the feature library. Dynamic binding allows us to instantiate methods specific to the object type at run time . The virtual function model allows us to develop algorithms for generalized objects which can be overridden at run time by specific functions coded for each derived class. These characteristics are especially useful in the development of customized feature libraries.

5.2.1 Object Class Definitions

C++ is used in this work as a tool to implement the object-oriented feature-based design system. The design of classes and their interaction is too extensive to explain here in full detail The top level classes used are shown in Table. 5.1.

Class Name	**Sub-Classes**
Component	Explained later in text
Graph	ComponentGraph, DimensionGraph
Block	Explained later in text
Arc	StrArc, DimArc
Face	-
Polygon	-
Length	-
LengthList	-
Node	FeatureNode, DimNode, CycleNode, StrongCompNode
Plane	-
Vertex	-
Queue	NodeQueue
Topolink	Explained later in text
Cycle	-
StrongComp	-
PolygonSet	-
Vector	-

Table 5.1: Object Classes

A description of the most important classes is presented below.

Class Component

Class Component is the uppermost class in the application. It is conceptually an assembly of blocks along with a set of topological links. In terms of data structure, it is represented as a **Graph** with blocks as nodes and topological links as arcs.

Some of the key functions of this class are discussed here; a full list is given in Appendix A.

1. *Link-Blocks* takes a new block as an argument and appends it to the existing list of blocks.

2. *Get-max-dimension* traverses the list of blocks and gets the maximum dimension of each block; based on this it returns the maximum dimension (span) of the component. This function is used to set the *Display Window* size automatically.

3. *Display-by-phigs* displays the model in wireframe.

4. *PolygonSet-Display* traverses the block list, collects the polygons,forms the polygon-sets and then displays them.

5. *Create-default-dim-graphs* traverses all the blocks, collects the dimension primitives and forms the default graph for the RDG and the SDG.

Class Block

Class Block is the individual shape unit referred to as a "geometric feature". It is a rectangular parallelopiped with orthohedral geometry. It contains three planes and one length along each axis. In terms of data structures, it is represented as a **Node**. It contains a pointer to the parent **Component** of which it is a part (Ref. Appendix A).

Subtractive blocks and blocks whose internal shapes are different from their external shape are represented by derived subclasses of **Class Block**. The derived classes are shown in Fig. 5.1 and are discussed below.

Some of the key functions of this class are discussed here; a full list is given in Appendix A.

1. *Create-Faces()* which returns six faces of this block.

2. *Display-by-phigs()* displays the block in wireframe.

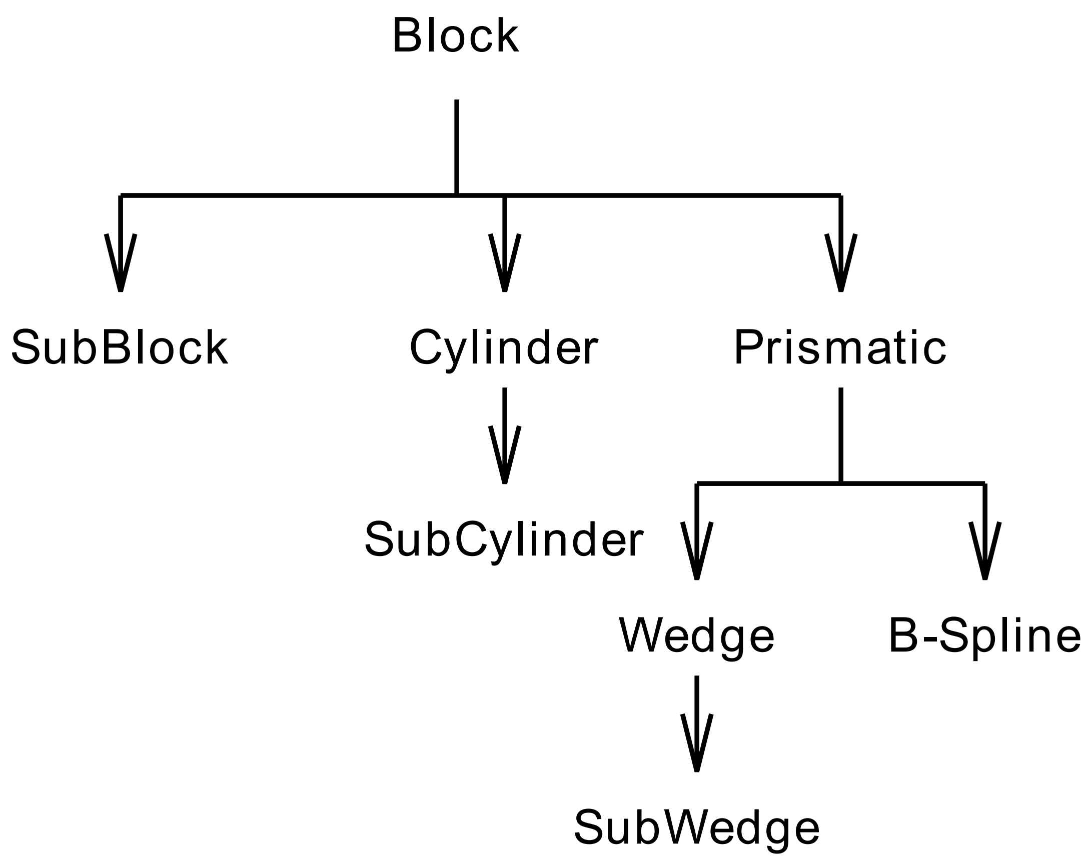


Figure 5.1: Block Hierarchy

3. *Contribute-dim-Faces()*, returns dimensioning planes corresponding to the top, bottom and two intersecting middle planes.

4. *Volume()* returns a positive volume of the block.

- - Class Name : **Subtractive Block**
  - Derived from : **Block**
  - Data Members : None
  - Member functions :
    * *Create-Faces()* which returns six faces of this block.
    * *Display-by-phigs()* which draws the block.
    * *Volume()* returns a negative volume of the block.

- - Class Name : **Cylinder**
  - Derived from : **Block**
  - Data Members :
    1. axis of the cylinder
    2. major-radius
    3. minor-radius
    4. height along the axis
  - Member functions :
    * *Create-Faces()* , which returns twelve side faces.
    * *Volume()* , which returns the volume of the cylinder.
    * *Display-by-phigs()*, draws the cylinder.
    * *Contribute-dim-Faces()*, returns dimensioning planes corresponding to the top, bottom and two intersecting middle planes.
    * *Create-Polygons()* , which returns two polygons with twelve sides each corresponding to the top and the bottom faces.

- - Class Name : **Subtractive Cylinder**
  - Derived from : **Cylinder**
  - Data Members : None

  - Member functions :
    - *Display-by-phigs()* draws the block.
    - *Volume()* returns a negative volume of the cylinder.

- - Class Name : **Prismatic**
  - Derived from : **Cylinder**
  - Data Members : the axis of extrusion
  - Member functions : - (Ref. Appendix A).

- - Class Name : **Wedge**
  - Derived from : **Prismatic**
  - Data Members : two butting planes
  - Member functions :
    - *Display-by-phigs()*, which draws the Wedge
    - *Create-Polygons()*, which returns a list of five polygons in which three are the side faces and remaining two are top and bottom faces.
    - *Contribute-dim-Faces()*, which returns the dimensioning planes.

- - Class Name : **Subtractive Wedge**
  - Derived from : **Wedge**
  - Data Members : None
  - Member functions : - (Ref. Appendix A).

- - Class Name : **B-Spline**
  - Derived from : **Prismatic**
  - Data Members :
    1. Order of the B-spline
    2. Knot vector
    3. Control Points
  - Member functions : (Ref. Appendix A).
    - *Display-by-phigs()*, which draws the block.

Class Topolink

Topolink : Topolink represents connectivity information between two blocks. Figure 5.2 shows the class hierarchy of the **Class Topolink**. It contains following data members :

- topolink-type : Type of the topological link
- Block-A-Id : first block (base block) for the topological link
- Block-B-Id : second block for the topological link

The operations defined in this class are as follows :

- *Get-Block-A-Id()*, which returns block-id of the block-A.
- *Get-Block-B-Id()*, which returns block-id of the block-B.
- *Get-topo-type()*, which returns topolink-type.

The derived classes is shown in Fig. 5.2 and are discussed below.

- - Class Name : **SymmTopolink**
  - Derived from : **Topolink**
  - Data Members :
    - arcAB : Arc from node-A to node-B
    - arcBA : Arc from node-B to node-A
- - Class Name : **Shared-Plane-Match**
  - Derived from : **SymmTopolink**
  - Data Members :
    - Plane1-Id : Resting plane on parent block-A
- - Class Name : **Center-Face-Match**
  - Derived from : **Shared-Plane-Match**
  - Data Members : None
- - Class Name : **Center-Edge-Match**

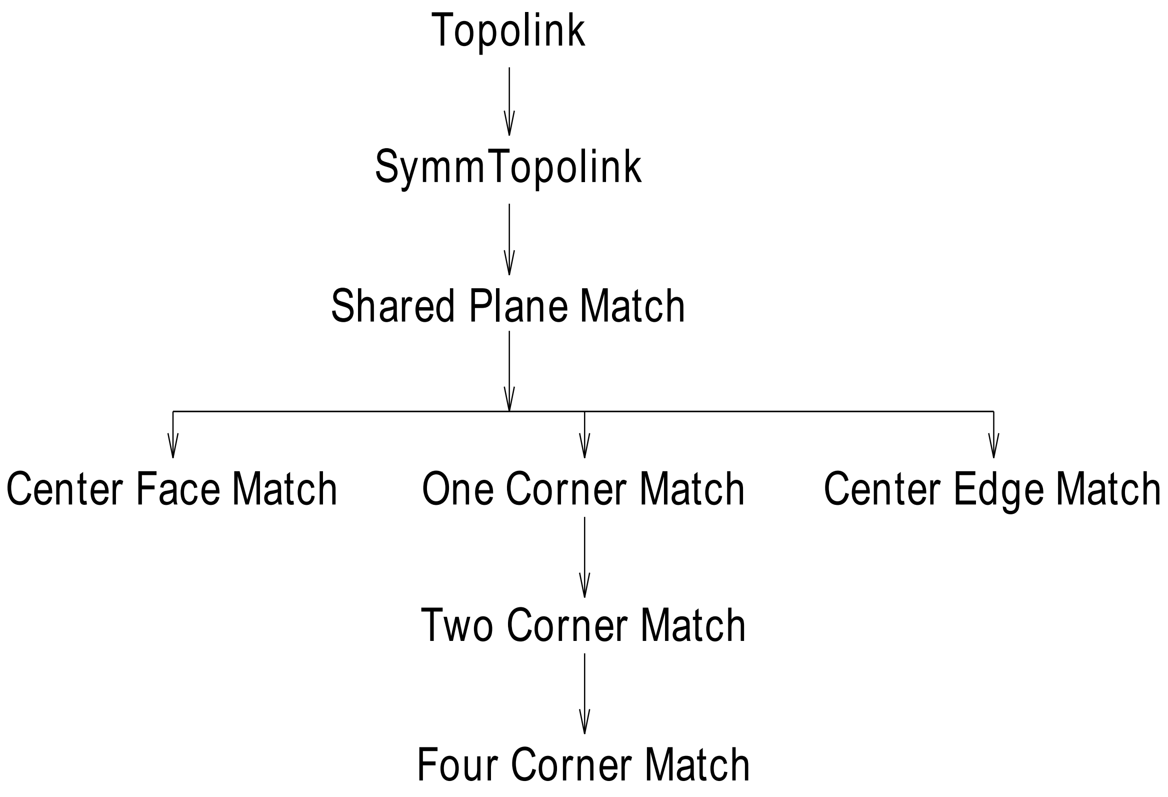


Figure 5.2: Topology Hierarchy

  - Derived from : **Shared-Plane-Match**
  - Data Members :
    * Plane2-Id : Second matched plane on parent block-A

- - Class Name : **One-Corner-Match**
  - Derived from : **Shared-Plane-Match**
  - Data Members :
    * Plane2-Id : second matched plane on parent block-A
    * Plane3-Id : third matched plane on parent block-A

- - Class Name : **Two-Corner-Match**
  - Derived from : **One-Corner-Match**
  - Data Members :
    * Plane4-Id : fourth matched plane on parent block-A

- - Class Name : **Four-Corner-Match**
  - Derived from : **Two-Corner-Match**
  - Data Members :
    * Plane1-Id : resting plane on parent block-A

5.2.2 Templates

Data structures like linked lists are very commonly used and are well defined in terms of their behavior. They only differ in the *type of object* contained in the list. For example, in the current implementation, linked lists of blocks, nodes and polygons are used in various algorithms. C++ provides a *generic* way of implementing a linked list that can contain any type of the object. This is done through the use of templates.

Templates are a relatively new feature of the C++ language. Templates provide an effective shorthand notation for defining families of classes with similar functions. These families have a similar form but they differ in the fact that they are parametrized by data types.

This work uses the template mechanism to define a family of classes related to the *List* data structure. It includes *Linear Link-list,Queues,* and *Stacks.* One of the simplest forms of a template class definition is given in the following example:

```
template <class T> class class-name
{
// ...
};
```

where T is an element type. Member variables and functions can depend on the data type T. T must be explicitly specified when template class variables are defined or declared in a program. For example :

```
template <class T> class Complex
{
private :

T real;
T imaginary;
public :

Complex( const T & r, const T & i):real(r),imaginary(i){}

T Real(){ return real;}
T  Img(){ return imaginary;}

};

main(){

Complex <double> x(1.0,2.0);
Complex <int> j(3,4);

cout << x.Real()<<" "<<x.Img() <<'\n';
cout << j.Real()<<" "<<j.Img() <<'\n';

return 0;
}
```

Two template class objects are created in the *main()* function. The object x is of the data-type *Complex* $<$ *double* $>$. The object j is of the data-type *Complex*$<$*int*$>$. Template class member functions have the values of their element types determined from the element types of the recipient object.

In the current implementation, template classes such as linked lists, stacks and queues are widely used in the implementation of graph algorithms such as *cycle detection* and *strong component detection.*

5.3 Graphical User Interface

The current implementation uses the Motif toolkit from the Open Software Foundation (OSF). The Motif toolkit is based on the X Toolkit Intrinsics (Xt), which is the standard platform on which many of the toolkits written for the X Window System are based. Xt provides a library of user interface objects called *widgets* and *gadgets*, which provide a convenient interface for creating and manipulating X windows, color-maps, events, and other cosmetic attributes of the display. In short, widgets can be thought of as building blocks that the programmer uses to construct a complete application [25].

5.3.1 Interface Structure

The main window area (see Fig. 5.3) is made up of four major interface items, viz.:

- Main Menu : consists of common functionality like file operations, display, utility functions, etc.
- Child Menu : presents either "design" options or "process planning" options depending on the mode selected.
- Message Area : displays prompts for input to the user and provides limited on-line help.
- drawing area : displays the component.

The *Main Menu* has the following options and sub-options :

- File :

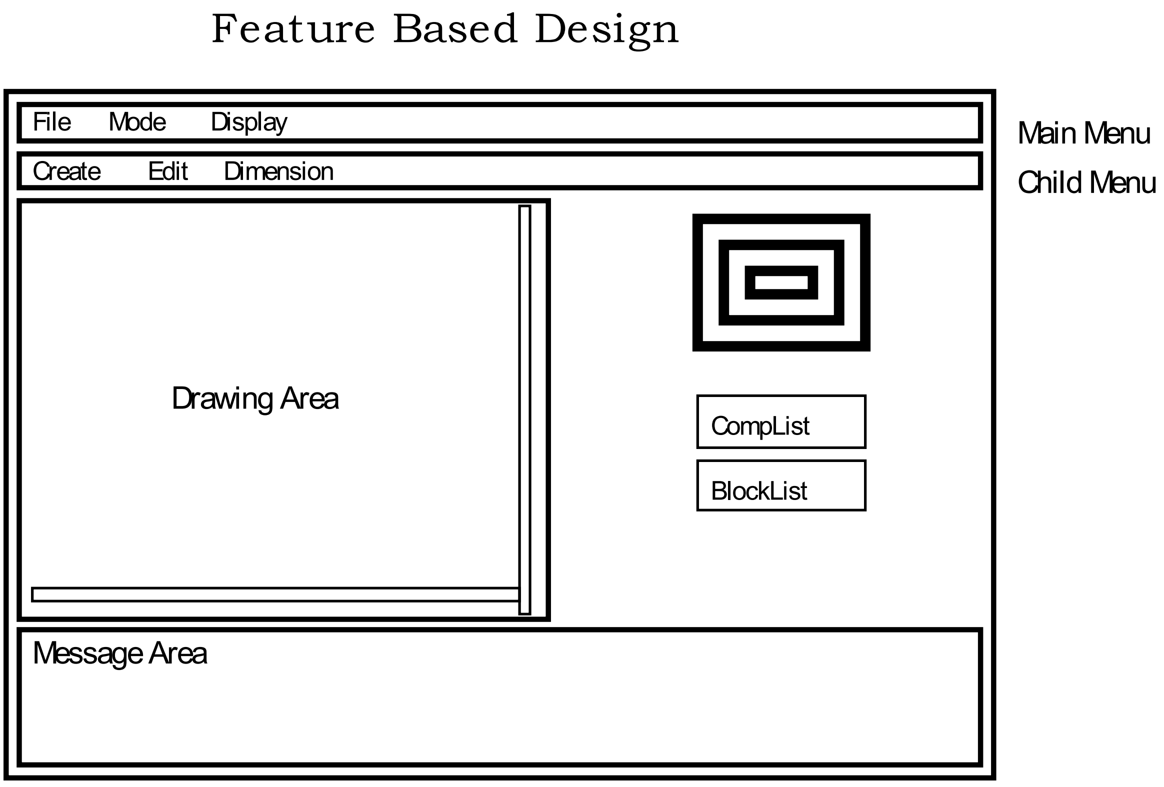


Figure 5.3: Main Window Interface

  - New : Begins a new component.
  - Retrieve : Retrieves the component stored in a file
  - Save : Saves the component to a file
  - Exit : Allows the user to exit the system

- Mode :
  - Design : Presents design related child menu below this main menu.
  - Manufacturing : Presents manufacturing related child menu below this main menu.

- Display :
  - Display-type : Sets display type to polygon-set.
  - Scale * : Scales the component
  - Rotate * : Rotates the component

- Utilities * :
  - Query
  - Mesh
  - Win-Resize
  - Volume
  - Geometric Consistency

- Settings * :
  - Window Settings

The *Child Menu* is of two types viz. "Design" and "Manufacturing". The "Design" child menu has following options and sub-options

- Create : presents the user with following options
  - New Component : allows the user to create a base block which requires purely geometric information

*Not supported yet

  - Add Block : Presents the shapes in the feature library to add to the existing base block. The shapes available are
    * ADDT-Block : Constructive Block
    * SUBT-Block : Subtractive Block
    * ADDT-Cylinder : Constructive Cylinder
    * SUBT-Cylinder : Subtractive Cylinder
    * ADDT-Wedge : Constructive Wedge
    * SUBT-Wedge : Subtractive Wedge
    * ADDT-B-spline : Constructive B-spline
  - Add Topolink : add a topological link between two pre-existing blocks.
- Edit :
  - Geo-Edit : allows the user to edit the geometric data of the component.
  - Add-Geomlink : allows the user to link lengths
  - Delete-Geomlink : allows the user to delink lengths
- DimTol :
  - Default-Dim : creates a default dimensioning scheme for the current component.
  - Update-Dim : re-creates the dimensioning scheme if the user has performed any "addition" or "editing" changes to the current component.
  - ChgDatum : allows the user to change the datum for dimensioning.
  - Query : allows the user to query the dimension between any two planes.

PHIGS is a higher level graphics library that is used to display the components. It provides basic objects and associated functionality with the help of which the user can build the model. The internal display procedure is abstracted from the user so that almost no lower level routines need to be written by the user.

Chapter 6

Conclusions and Recommendations

6.1 Conclusions

A major problem in CAD/CAM is to device appropriate representations for modeling product information in a computer. Feature-based design offers a tool for mapping a designer's abstraction into a representational form that is suitable for manufacturing and other downstream applications.

In the research work presented herein, we were able to successfully develop and implement a feature-based design system for generalized orthohedral components.

A data representation scheme was developed for modeling three dimensional solid components with orthohedral external shape and a wide variety of internal shapes. The geometry and topology are explicitly represented.

The feature library is extendable and customized to suit the needs of any given application. This extendability is also exploited to provide support for free-form surfaces (extruded B-Spline) through the mechanism of encapsulation. This capability offers flexibility in enriching the feature set and supporting customized features.

The family of features currently supported includes :

- Constructive Block
- Subtractive Block
- Constructive Cylinder
- Subtractive Cylinder

- Constructive Wedge
- Subtractive Wedge
- Constructive swept B-Spline surface

Operations such as *Create*, *Edit*, *Display*, *Save* and *Retrieve* are also supported.

The architecture of the prototype system is *open*, providing the infrastructure for additional development to enhance its functionality and modeling capabilities.

Feature-based design systems are generally lacking in dimensioning and tolerancing support. The present work develops and implements a robust dimensioning and tolerancing model with the capability to create, modify and validate part dimensioning schemes.

C++ is used to implement the proposed feature-based design concepts using the object-oriented methodology. The user-interface is developed in the X-Windows Motif toolkit, while the graphics display is done using PHIGS. The user-interface and application classes are separated as much as possible, thereby providing the capability to change the user interface with minimal modifications in the object classes themselves.

6.2 Recommendations

The FBD prototype developed can be used to model components with orthohedral, cylindrical or prismatic geometry, with limited support for free-form surfaces. This limitation can be reduced by adding new geometric features to the available set of geometric features. The mechanism of encapsulation could be used to model much wider classes of geometric shapes and topological links.

Since the prototype FBD system supports both design and manufacturing features, it could be extended to generate process planning data.

The D & T facility that is provided has limited support for tolerance analysis. Useful work could be done in this area to provide the capability for tolerance synthesis and validation. The user interface for the D & T portion of the software has a lot of room for improvement.

In the present prototype FBD package, the features are not written in internationally recognized formats. This capability would allow other applications such as finite element analysis or NC code generation to use the current solid modeling system for modeling the components. Thus, support for international standards such as PDES/STEP and industry standards like DXF will greatly improve the usability of the software.

References

[1] J. Sheu, L.C.and Lin, "Representation scheme for defining and operating form features," *Computer Aided Design*, vol. 25, pp. 333–347, June 1993.

[2] I. Sutherland, "Sketchpad:a man-machine graphical communication system," *Proceedings of the SJCC*, vol. 23, pp. 329–49, 1963.

[3] B.-C. Zhang, *Geometric modeling of dimensioning and tolerancing.* PhD thesis, Arizona State Univeristy,USA, 1992.

[4] J. J. Shah and M. T. Rogers, "Functional requirements and conceptual designs of the feature-based modelling system," *Computer-Aided Engineering Journal*, pp. 9–15, 1988.

[5] R. Johnson, "Dimensioning and tolerancing - final report," tech. rep., R84-GM-02-2,CAM-I,Arlington,Texas, 1985.

[6] J. J. Shah, "Assessment of features technology," *Computer Aided Design*, vol. 23, no. 5, pp. 331–343, 1991.

[7] J. Requicha, A.A.G.and Vandenbrande, "Form features for mechanical design and manufacturing," in *ASME Computers in Engineering Conference*, (Anaheim), pp. 47–52, 1989.

[8] B. Strostrup, *The C++ Programming Language.* Addison-Wesley, 2nd ed., 1991.

[9] A. Grayer, *A computer link between design and manufacture.* PhD thesis, Univeristy of Cambridge,UK, 1976.

[10] J. Cunningham, J. J.and Dixon, "Designing with features:the origin of features," *ASME Computers in Engineering Conference*, pp. 237–243, 1988.

[11] M. Vaghul, M. ,Dixon, J.R., Zinsmeister, G.E.and Simmons, "Expert systems in a cad environment: Injection moulding part design as an example," *Proc. ASME Computers in Engineering conference*, pp. 77–82, 1985.

[12] B. Casu, L.and Falcidieno, "A feature based modelling system built on top of euler operators," *Proc.,CG International,Springer Verlag*, pp. 471–488, 1989.

[13] F. Falcidieno, B.and Gianninir, "Extraction and organization of form features into structured boundary model," *Procs.EUROGRAPHICS*, pp. 149–159, 1987.

[14] I. Hijazi, "An object-oriented feature-based design methodology for solids with orthohedral geometry," Master's thesis, Kansas State Univeristy,USA, 1993.

[15] V. Chennapragada, "Feature-based design orthohedral solids using constructive and subtractive features," Master's thesis, Kansas State Univeristy,USA, 1995.

[16] Y. Hanju, "Feature based design for orthohedral solids using predefined solid primitives, cylinder, rectangular parallelopiped block ,triangular prism and cone," Master's thesis, Kansas State Univeristy,USA, 1996.

[17] "Pro/engineer." software packge for CAD/CAM applications by Parametric Technology Corporation, USA.

[18] T. Howard, "Evaluating phigs for cad and general graphics applications," *Dept. of Computer science,Univ. of Manchester.UK*, 1991.

[19] N. Reingold, E.M.Nievergelt,J.and Deo, *Combinatorial Algorithms :Theory and Practice.* ,Prentice Hall Inc.NJ.USA., 1977.

[20] C. Roy, U.and Liu, "Feature-based representationalscheme of a solid modeler for providing dimensioning and tolerancing information," *Robotics and Computer-Integrated Manufacturing*, vol. 4, no. 3/4, pp. 335–345, 1988.

[21] O. Bjorke, *Computer-Aided Tolerancing.* Tapir Publishers,Norway, 1978.

[22] S. Requicha, A.A.G.and Chan, "Representation of geometric features , toleranances , and attributes in solid modeler based on csg," Tech. Rep. 48, University of Rochester, NY,USA, 1985.

[23] A. Requicha, "Toward a theory of geometric tolerancig," *Journal of Robotics Research*, vol. 2, no. 4, pp. 45–60, 1983.

[24] C. Lin, *Feature Based Design and Tolerancing.* PhD thesis, Kansas State Univeristy,USA, 1992.

[25] D. Heller and P. Ferguson, *Motif Programming Manual.* O'Reilly and Associates,Inc., volume six a ed., 1994.

Appendix A

Class Definitions

- *Class Component*

```
class Component {
    int     component_id ;          // component number
    char    component_name[20] ;    // Component name (string)
    int  block_id_counter ;   // for id assignment of blocks
    int  num_polygonsets;  // Num of polysets attched to firstset
    Component   *nextc;    // next component
    Block  *comp_block ;    // to the first block of block list
    PolygonSet  *firstset; // head of the list of polygon sets
    Graph    *comp_graph ;    // graph which represents this comp
    DimensionGraph  *ref_dim_graph; // Reference Dimension Graph
    DimensionGraph  *spec_dim_graph;// Specified Dimension Graph
public :
    /*
    **          Link List functionality
    */
    void        Link_Blocks( Block *cb);
    void        Link_PolygonSets(PolygonSet* pp);
    /*
    **          Display functionality
    */
```

```
        virtual double      Get_max_dimension();
        virtual void        Display_by_phigs(int type);
        virtual int         Check_coplanarity(Polygon* ,Polygon*);
        virtual void        PolygonSet_Display();
        /*
        **          Dimensioning functionality
        */
        virtual void Create_default_dim_graphs();
        /*
        **          Editing functions
        */
        virtual void Copy_for_Edit( Component* co);
        /*
        **          File operations
        */
        virtual void Write_to_file( FILE *fp);
        virtual void Write_to_Venkys_file( FILE *fp);
};
```

• *Class Block*

```
class Block {
    protected :
        /*
        **      Block Identification
        */
        int     block_id ;      // block number
        int     block_type;     // Block type
        char    block_name[20]; // block name
        /*
        **      Link List members
        */
        Block   *nextb ; // next block in link list of blocks
        Block   *prevb ; // previous block in link list of blocks
        /*
        **      Graph(Node) Representation
        */
        Node    *block_node ;// Node which represents this block
        /*
        **      Geometric Representation
        */
        Plane   *Xmin , *Xmid , *Xmax ; // Planes along X axis
        Plane   *Ymin , *Ymid , *Ymax ; // Planes along Y axis
        Plane   *Zmin , *Zmid , *Zmax ; // Planes along Z axis
        Length  *Xlen , *Ylen , *Zlen ; // Lengths along 3 axes
        /*
        **      Connection to Component of which it is a part
        */
        Component   *block_comp ;   // parent component
    public :
        //...
        virtual void        Initialize_flags(int boo);
        virtual double      Get_max_dimension();
        virtual void        Update(int axis);

        virtual Face*       Create_Faces();
        virtual Polygon*    Create_Polygons(){return NULL;}
        virtual void        Display_by_phigs();
        /*
        **      Dimensioning Functionality
        */
        virtual Face*       Contribute_dim_Faces();
};
```

- *Subtractive Block*

```
class SubBlock : public Block
```

```
{
    public:
        // ...
        virtual Face*  Create_Faces();
        virtual void   Display_by_phigs();
        /*
        **      Dimensioning Functionality
        */
        virtual Face*   Contribute_dim_Faces();
};
```

- *Cylinder*

```
class Cylinder: public Block
{
    protected :

        int  axis ;     // height axis
        double  major_radius ;  // Major radius
        double  minor_radius ;  // Minjor radius
        double  height ;    // height along axis
    public:
        // ...
        /*
        **      Display functionality
        */
        virtual Polygon* Create_Polygons();
        virtual void   Display_by_phigs();
        /*
        **      Dimensioning Functionality
        */
        virtual Face*  Contribute_dim_Faces();
};
```

- *Subtractive Cylinder*

```
class SubCylinder: public Cylinder
{
    public:
        // ...
        /*
        **      Display functionality
        */
        virtual void        Display_by_phigs();
};
```

- *Prismatic*

```
class Prismatic:public Block
{
    protected :
        int prism_axis;
    public:
        // ..
        /*
        **      Display functionality
        */
        virtual Polygon*    Create_Polygons();
        virtual void Display_by_phigs();
        /*
        **      Dimensioning Functionality
        */
        virtual Face* Contribute_dim_Faces();
};
```

- *Wedge*

```
class Wedge:public Prismatic
{
    protected :
        int butting_plane1;// Plane matching with Block
        int butting_plane2;// Plane matching with Block
    public:
        /*
        **      Display functionality
        */
        virtual Polygon* Create_Polygons();
        virtual void Display_by_phigs();
};
```

- *Subtractive Wedge*

```
class SubWedge:public Wedge
{
    public:
        /*
        **      Display functionality
        */
        virtual Polygon* Create_Polygons();
        virtual void Display_by_phigs();
};
```

- *B-spline*

```
class Bspline:public Prismatic
```

```
{
    protected :
        int     u_order;    //  Order in U direction = 3
        int     u_num_pts;  //  Number of points in U direction
        float   u_knot_vector[23];  //  Ends with multiplicity 3
        Vertex  u_control_pts[20];  //  Control points
    public:
        // ...
        /*
        **      Display functionality
        */
        virtual void Display_by_phigs();
};
```