THE CAD QUERY LANGUAGE: TOWARDS DESIGN-CONCURRENT
COST ESTIMATION

A Thesis Presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Master Of Science

by

Prahlad S. Athreya

November, 2002
ACKNOWLEDGMENTS

The successful completion of this work has been largely possible because of the foresight and guidance of the omnipresent Dr. Judd. I will always be grateful to him, for teaching me by example and showing me that there are no shortcuts to good work. I am thankful to the members of my Thesis committee for showing great interest in this work and for being extremely co-operative at all times.

This work has been possible due to ground-breaking work done by others at CASSI, Ohio University. Most prominent among them is Lizhong Huang, the developer of EQL which has been the role model for CQL. Viswanath Majeti helped me during the early days with my struggle with the UG API. Chandrasekhar Subramanian saved me hours of mindless work by convincing me to use \LaTeX. The staff at the Alden Library, Ohio University were extremely co-operative during the course of the literature search.

Finally, for the people who have shaped my life at various stages to bring me where I am – my teachers, I will always hold the utmost respect.

Asato maa sad gamaya
Tamaso maa jyotir gamaya
Mrtyor maa 'mrtam gamaya
-Brhadaaranyaka Upanishad

(Lead me from the unreal to the real,
Lead me from darkness to light,
Lead me from death to immortality)
Contents

1 Introduction ......................................................... 1
  1.1 Introduction .................................................. 1
  1.2 Design-Time Cost Estimation ................................. 3
    1.2.1 The Conceptual Design Stage ............................ 3
    1.2.2 The Preliminary Design Stage ............................ 4
    1.2.3 The Detailed Design Stage ............................... 5
  1.3 Computer Aided Estimation .................................... 6
  1.4 Feature-Based Estimation ..................................... 8
    1.4.1 Features .................................................. 8
    1.4.2 Feature-Based Estimating Approaches .................... 9
  1.5 The Challenges Ahead ......................................... 10
    1.5.1 Disparate CAD System Views of Features ............... 10
    1.5.2 Heterogeneity in CAD System Interfaces for Data Exchange 11
    1.5.3 CAD Part Complexity .................................. 11
    1.5.4 Information Loss ..................................... 12
    1.5.5 Ad-Hoc Information Extraction ......................... 12
  1.6 A Survey on Estimation Practices ............................ 13
  1.7 Problem Statement ............................................. 15
1.8 Problem Description ................................................. 15

2 The CAD System Abstraction ........................................... 17
2.1 Introduction ......................................................... 17
2.2 The Feature Mapping Problem ..................................... 17
2.3 Confronting Heterogeneity ......................................... 18
2.4 Ad-hoc Information Extraction ..................................... 19
2.5 The CAD Object Model .............................................. 20
2.5.1 What is the CAD Object Model? ............................... 20
2.5.2 Model Elements .................................................. 23
2.5.3 Model Example ................................................... 24

3 CQL: Syntax and Use ................................................... 29
3.1 Introduction ........................................................ 29
3.2 Query Elements ..................................................... 30
3.3 Important Constructs ............................................... 32
3.4 The From Clause ................................................... 33
3.5 The Where Clause ................................................... 35
3.5.1 The Null Condition ............................................. 37
3.5.2 The ISA Condition .............................................. 37
3.5.3 The Empty Condition .......................................... 38
3.5.4 The IN Condition .............................................. 39
3.5.5 The Contains Condition ...................................... 40
3.5.6 The Comparison Condition ................................... 40
3.5.7 The For Condition ............................................. 41
3.5.8 The Match Condition ........................................... 42
4.4 Organization of CQL Code ........................................... 91
4.5 Important Global Variables ......................................... 92
4.6 Important Routines ..................................................... 94
  4.6.1 Program Control Functions ..................................... 95
  4.6.2 Node Manipulation Functions .................................. 95
  4.6.3 Node Execution Functions ...................................... 98
  4.6.4 UG API Support Functions ..................................... 101
  4.6.5 Error Handlers ................................................... 102

5 Results and Discussion .................................................. 105
  5.1 Introduction .......................................................... 105
  5.2 Design Objectives .................................................... 105
  5.3 Examples with Sample Parts ....................................... 106
  5.4 Summary of Examples ............................................... 122
  5.5 Important Contributions ............................................ 123
  5.6 Difficult Questions .................................................. 123
  5.7 Conclusions and Recommendations ............................... 124

A EXPRESS Schema .......................................................... 127

B Entities and Associated Methods ...................................... 135

C CQL Methods ............................................................... 143
  C.1 Single-Valued Methods ............................................. 143
  C.2 Multi-Valued Methods .............................................. 150

D Types in CQL ............................................................... 155
  D.1 Entity Types ........................................................ 155
D.2 Face Types .................................................. 155
D.3 Feature Types ............................................... 157
D.4 Color Codes .................................................. 157

E The Unigraphics API .......................................... 160
E.1 Preliminary Tasks ........................................... 160
E.2 Single-Valued Methods ..................................... 161
E.3 Multi-Valued Methods ...................................... 165
E.4 Support Routines ........................................... 170
E.5 De-allocating Dynamic Memory ......................... 173
E.6 System Functions ........................................... 173
E.7 User Defined Objects ....................................... 174

F TNR Demonstration Programs ................................ 177
F.1 casing.tnr .................................................. 177
F.2 ge.tnr ....................................................... 182
F.3 parker2.tnr .................................................. 183
List of Tables

3.1 Legal Data Types For Comparison .......................... 41

D.1 Entity Types .................................................... 156
D.2 Face Types ...................................................... 157
D.3 Feature Types .................................................. 158
D.4 Color Codes ................................................... 159
List of Figures

2.1 The CAD Object Model ........................................ 22
2.2 Methods for the Feature Entity ................................. 25
2.3 EXPRESS Schema Definition of Feature ......................... 26

3.1 Grammar Rules for Constants ................................. 32
3.2 Grammar Rules for the From and Implied-From Clauses .... 34
3.3 Grammar Rules for the Where Clause ......................... 36
3.4 Types of Atomic Conditions .................................. 37
3.5 The Null Condition ............................................. 38
3.6 The ISA Condition .............................................. 38
3.7 The Empty Condition ............................................ 39
3.8 The In Condition ............................................... 39
3.9 The Contains Condition ....................................... 40
3.10 The Comparison Condition ................................... 41
3.11 The For Condition ............................................. 42
3.12 The Match Condition .......................................... 43
3.13 Compound Conditions ......................................... 43
3.14 The Select Clause .............................................. 45
3.15 The Select Query ............................................... 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.16</td>
<td>The System Query</td>
<td>47</td>
</tr>
<tr>
<td>3.17</td>
<td>The Suppress and Unsuppress Queries</td>
<td>50</td>
</tr>
<tr>
<td>3.18</td>
<td>Example Part for Suppress</td>
<td>51</td>
</tr>
<tr>
<td>3.19</td>
<td>Suppress Query Example</td>
<td>52</td>
</tr>
<tr>
<td>3.20</td>
<td>features.prt</td>
<td>56</td>
</tr>
<tr>
<td>3.21</td>
<td>sheet_demo.prt</td>
<td>60</td>
</tr>
<tr>
<td>3.22</td>
<td>assy.prt</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Architecture of CQL</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Structure of a q_node</td>
<td>72</td>
</tr>
<tr>
<td>4.3</td>
<td>A q_node Chain</td>
<td>72</td>
</tr>
<tr>
<td>4.4</td>
<td>Structure of a v_node</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of an Aggregate v_node of Integers</td>
<td>74</td>
</tr>
<tr>
<td>4.6</td>
<td>Conceptual Structure of the Node Tree</td>
<td>75</td>
</tr>
<tr>
<td>4.7</td>
<td>Node Tree after Parsing</td>
<td>76</td>
</tr>
<tr>
<td>4.8</td>
<td>Node Tree after Binding</td>
<td>77</td>
</tr>
<tr>
<td>4.9</td>
<td>Node Tree without Optimization for Simple Query</td>
<td>78</td>
</tr>
<tr>
<td>4.10</td>
<td>Node Tree with Optimization for Simple Query</td>
<td>79</td>
</tr>
<tr>
<td>4.11</td>
<td>Optimized Tree</td>
<td>80</td>
</tr>
<tr>
<td>4.12</td>
<td>Conditional Operator Nodes</td>
<td>84</td>
</tr>
<tr>
<td>4.13</td>
<td>For Condition Node</td>
<td>85</td>
</tr>
<tr>
<td>4.14</td>
<td>Result Data Structure Containing Rows of Integers</td>
<td>87</td>
</tr>
<tr>
<td>4.15</td>
<td>Suppress Node Tree</td>
<td>89</td>
</tr>
<tr>
<td>4.16</td>
<td>System Query Structures</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>Suppression Phase 1</td>
<td>108</td>
</tr>
<tr>
<td>5.2</td>
<td>Suppression Phase 2</td>
<td>111</td>
</tr>
</tbody>
</table>
5.3 Suppression Phase 3 ........................................ 112
5.4 Sectional View of Turned Part ................................ 117
5.5 Tube Assembly ................................................... 119
5.6 Output Information from Tube Assy .......................... 121

E.1 Structure of a FaceLoop .......................................... 176
Chapter 1

Introduction

1.1 Introduction

The millennium is here. Sweeping economic and political changes, coupled with breakthroughs in critical technologies like communication and information processing, while opening up vast opportunities, have also made an enterprise or nation susceptible to competition from virtually any part of the world. This is evident from the development of a number of corporate strategies like re-engineering, right-sizing, TQM etc., all focussed on re-examination of the business process with a view to improving operating efficiencies, reducing product deployment time and enhancing product quality. Management strategies aside, one of the age old techniques for ensuring the survival of a product and the enterprise has been the use of sound cost estimation strategies for cost management. Sound cost management policies have, since the beginning of free enterprise, proved to be crucial to profitability. Cost estimation is widely used in several areas of manufacturing, by a wide range of persons for different purposes. This thesis focusses on facilitating the integration of cost estimation into the design process.
The product cost estimate is an estimate of the costs in making a part, subassembly or entire product [1]. Product cost estimates are primarily made to serve the following purposes [1, 2]:

1. establish the bid price of a product for a quotation or contract
2. enable the manufacturer to choose the most economic manufacturing process from alternatives
3. ascertain whether a proposed product can be manufactured profitably
4. provide data for make-buy decisions
5. provide a temporary standard for production efficiency and guide operating costs at the beginning of a project
6. help the manufacturer plan for procurement of tools, raw materials, equipment and predict labor and capital requirements

In recent times, there has been a growing realization for the need to control product costs right from the conceptual design stage. According to Wierda [3], the designer is in a unique position to effect cost reduction measures. He states that the design engineer, during the process of design, defines about 75% of final product cost. However, the designer rarely has the information necessary to make cost reducing design changes at his disposal. Even when this information is available, it is seldom in a form that can be related to design decisions. Several efforts have been made to develop interactive tools that keep the designer informed of the cost impact at every stage of the design process.
1.2 Design-Time Cost Estimation

Research in design-time cost estimation and the practical experiences of industry, have resulted in techniques that produce a variety of estimates ranging from the easily computable ball-park estimate to the accurate detailed cost estimate. The choice of estimating technique largely depends upon the stage of design at which estimation is performed or, equivalently, the quantum of information available about the product. The following sections outline the techniques currently available to the designer.

1.2.1 The Conceptual Design Stage

At this stage, the geometry and materials of the product are as yet unknown and estimating methods have accuracies ranging from -30% to +50%. The costs incurred during this stage of design are projected to be about 5-7% of the total cost whereas committed costs are in the order of 75–85% [2]. Consequently, significant cost savings can be effected at this stage. The following estimating methods find application at this stage.

**Market Surveys** Similar products in the market are examined to get an idea of the existing cost levels. This may lead to decisions regarding the acceptance or rejection of the product concept.

**Weight-Based Methods** Production costs and weights of existing products may be used to derive a relation between cost and product weight. This information may be used to compute the cost of a new product if its approximate weight is known [3]. However, this method is applicable only to products that are similar in design and manufacturing.
Material Cost Share Method This method finds application in manufacturing industries where the material cost forms a significant fraction of the final product cost. Based on the observed ratios between the material cost and total product cost for several product categories, the cost of a new product may be computed knowing its material cost. Creese [2] provides statistics on observed material cost shares for several common product categories.

Other Heuristics Apart from the above methods, expert opinion based on knowledge of the production environment or other heuristics developed from historical observations may be employed. Rondeau [4] proposes the '1-3-9 rule', whereby, for a progressive manufacturing company, the material cost, manufacturing cost and selling price are in the ratio 1:3:9. However, expert opinion is not generally preferred as it cannot be substantiated, is subject to bias and does not lead to an information base for future estimates [2].

1.2.2 The Preliminary Design Stage

At this time, typically, all of the materials and dimensions of the product are known and estimates are sufficiently accurate to determine project feasibility. Estimate accuracy ranges from -15% to +30%. The following are the primary estimating techniques used at this stage.

Cost Functions Cost functions express product cost as a function of product form features, product function or other design attributes. Considerable research in this area has resulted in the development of cost functions for machining, injection-molding, forging, die-casting, powder metallurgy and other processes. They are very useful to the designer in that they establish relationships between product attributes and costs. However, considerable research is needed
to establish a cost function, and a new cost function must be established for each group of products [3].

**Parametric Estimating** Parametric estimates are employed when the product considered is complex and consists of hundreds of components, like an aircraft. Simulation models, consisting of hundreds of cost estimating relationships (CERs), are constructed to determine the effect of product parameters on cost. Parametric estimates have a proven record for large projects, but are inapplicable to single components [5].

### 1.2.3 The Detailed Design Stage

All information about the product, such as dimensions, tolerances, finishes, scheduling, lot sizes etc... is known at this stage and estimates are accurate to within -5% to +10% of actual cost. All estimates at this stage are detailed to make the best use of available product information. Operations based estimating and activity based estimating find application at this stage.

**Operations Based Estimating** This is the most accurate method of estimation. All the operations required to be performed to produce the product are charted by the development of a process plan. Costs are then associated with each operation to arrive at the final product cost. This process requires considerable expertise and involves significant amounts of time and effort. Even though the estimates are highly representative of actual costs, they are not of much utility to the designer, as they do not correlate costs to design decisions.

**Activity Based Estimating** Indirect costs frequently account for a major portion of product costs, sometimes as high as 80% [6]. An activity is a chain of
transactions to achieve a result [6]. Activities consume resources during their execution. Activity based costing segregates expenses of direct and indirect support departments and allocates costs to activities based on the resources consumed. Product costs are established based on the usage of activities. Like operations based estimating, this is extremely expensive in terms of time, money and effort.

At this juncture, the utility of the above methods may be summed up by the following observations [3]:

- accurate estimates based on operations are far too time-consuming to be economical at the design stage;
- fast estimates from cost functions necessitate considerable research and are extremely specific to the estimating problem;
- very little use is made of the historical cost information gained from previous design experiences.

Simultaneous advancements in the areas of databases, process planning and computing has led to computer aided estimation, offering new insights into the above problems.

### 1.3 Computer Aided Estimation

Early efforts at computer aided estimation in academia resulted in the development of several prototype systems. Wierda describes the characteristics of several such systems [3]. Broadly speaking, all such systems may be classified as either variant-based systems or generative cost estimating systems, based on the estimating technique.
**Variant-Based Systems** Variant based estimation relies on the fact that parts that are similar in shape are manufactured by similar methods. By means of a search in a database of parts made earlier, a template part is retrieved that has a high degree of *similarity* to the part to be estimated. The cost of the retrieved part is then adjusted to account for the dissimilarities between the parts compared. Some of the major problems identified with such systems include the following:

- insufficient reliability of classification systems used for organizing the template part database;
- lack of integration with other systems;
- the establishment of a company specific parts database.

However, when properly implemented and used, variant based cost estimation is recognized to be a quick and relatively inexpensive process for estimating standard components.

**Generative Systems** Generative cost estimation arrives at an estimate by using detailed information regarding operation times and costs derived from a knowledge of part manufacturing processes. Some early systems machined a computer model of the part, step by step, based on user instructions, thus accumulating costs [3]. More recently, automated process plans for the part are used to estimate operation times. This process is generally used for high variety products. It is time-consuming and is resorted to only when no other methods are applicable. Moreover, it requires detailed information about the part to be estimated.
1.4 Feature-Based Estimation

Computer aided design software has significantly enhanced the capability of a designer to evaluate alternative designs. As a result, almost all product design is currently performed in an integrated CAD system environment facilitating FEM analyses, generation of NC programs, kinematic simulations, tolerance analysis etc. Several attempts have been made to integrate the cost estimation process into Computer Aided Design. Ongoing research is strongly centered around the concept of features.

1.4.1 Features

The ASME document on Dimensioning and Tolerancing defines a feature to be [7]

"the general term applied to a physical portion of a part such as a surface, tab, pin, hole or slot".

Although considerable ambiguity exists over the classification of features, there is general agreement that features are entities that are of a semantically higher level than the pure geometric elements used in solid modeling [8]. From the design perspective, features may be thought of as subsets of part geometry that fully and uniquely represent the designer’s intent in terms of functional characteristics and topological information [9]. Modern CAD systems support the process of feature-based design, wherein the designer constructs a part using pre-defined features from a feature library. This library may be extended by the addition of user-defined features. Thus, features are recognized to be highly pertinent product sub-units upon which estimates may be based.
1.4.2 Feature-Based Estimating Approaches

The first attempts at harnessing feature information were made before the appearance of feature based modelers. Consequently, the estimating system had to provide a means of selecting a feature by identifying its geometric elements and provide the storage structures to hold feature parameters. This was achieved in two common ways.

**Feature Identification** During this process, the user manually identified all the form features comprising the part. Feature parameters were either computed from key geometric elements comprising the feature or were entered by the user. Ehrlenspiel describes the XKIS system [10], which was based on the above principles and which could be used at the end of modeling. After all features were identified, an inference engine containing rules relating to feature manufacturability was used to perform machine selection. Subsequently, a rough process plan was developed and used to estimate part costs. Some of the major drawbacks of such a system were the following:

- in case of complex parts, the process of identification of feature geometry quickly became unwieldy;

- the system put the onus of correctly identifying all the features on the user.

**Feature Recognition** The alternative to manual feature identification has been to subject the CAD model to feature recognition. By comparison of the decomposed volumes with features from a pre-defined feature library, feature recognition is performed [10]. However, any non-geometric technological information which is part of the CAD model, such as tolerances, surface finish, density
etc... is lost. In addition, feature recognition algorithms are easily confounded by the presence of feature interactions and can thus only operate on CAD models of restricted complexity. The features obtained at the end of feature recognition may be used to generate a rough process plan to estimate part costs as before.

1.5 The Challenges Ahead

As a result of research, cost estimation techniques have continually become more robust and representative of the actual product cost. During the early years of research, the focus was on the design of a technique that would provide the best estimate with the given data. With refinement in estimating techniques, the need for representing product cost in terms of designer-controllable product parameters came to the forefront, eventually leading to feature-based estimation. In recent years, there has been considerable growth in the use of software tools for design, prototyping, simulation and manufacturing control [11]. As far as product development is concerned, the CAD system has emerged as the control center of all activity. Hence, the need for integrating the estimating function into the designer’s familiar CAD system environment is felt more than ever [10]. The emergence of the feature-based CAD system has played a pivotal role in this regard. The following sections discuss some of the key issues that must be resolved to facilitate such an integration.

1.5.1 Disparate CAD System Views of Features

The scientific and engineering community has still not reached a consensus on what clearly constitutes a feature and what does not. Consequently, different CAD systems
possess different feature libraries developed to suit the designer’s picturisation of a product better. As a result, any application engaging in feature data exchange with a CAD system, must, at some time or the other, contend with the issue of feature mapping to reconcile the different views of features.

1.5.2 Heterogeneity in CAD System Interfaces for Data Exchange

Any computer integration effort has to effectively address the issue of heterogeneity. Heterogeneity may be explained as the co-existing of several systems, possibly using diverse mechanisms to contribute to the overall activity of the enterprise, which are united by the need to share information or resources. Manufacturing systems, like large computer networks and the world wide web, are characterised by heterogeneity [12]. This arises from the fact that there are usually several solutions to similar engineering requirements, each affected by considerations such as cost, supplier reliability, legacy systems etc. Heterogeneity enables the use of the right kind of technology for different portions of the enterprise. Consequently, a cost estimation application may be required to work with several CAD systems and other manufacturing support applications, each of which possibly offers fundamentally different interfaces for data exchange.

1.5.3 CAD Part Complexity

CAD systems support the representation of extremely complex engineering products. A given product may be comprised of several sub-assemblies, each of which may have sub-assemblies of its own. A design engineer may arbitrarily establish groupings between individual parts and/or assemblies to facilitate design. Part features may have
hierarchical parent-child relationships with other features. Engineering attributes may be assigned to design objects. The navigation of such an intricate model structure to finally retrieve the information of interest is a non-trivial task, which feature dependent applications must perform. In addition, different CAD systems may have different model structures to represent the same product in their domain. Thus, the underlying CAD model structure to navigate may be CAD system dependent.

1.5.4 Information Loss

The problem of having to navigate through a CAD system's native model structure and the need to perform feature mapping may be eliminated by treating the CAD model as purely a collection of geometry and subjecting the geometry to feature recognition. However, this results in a loss of design intent and any user defined information that may be associated with the model. Moreover, feature recognition cannot capture vital inter-component relationships such as assemblies. Furthermore, even with single parts, feature recognition algorithms are unreliable in the presence of feature interactions [8]. No CAD data exchange standards currently exist that can capture all CAD model information. The closest is the AP203 STEP standard that specifies the structure of a neutral B-Rep format to store CAD model information. Hence, what is needed is a *lossless* mechanism to extract product information.

1.5.5 Ad-Hoc Information Extraction

When dealing with the issue of facilitating design-concurrent cost estimation, it must be borne in mind that enabling extraction of feature data alone may not be sufficient. Part and feature attributes are some of the other aspects of a CAD model that may provide useful estimation information. In all cases, the exact nature of
the information to be extracted is not known \textit{a priori}, as it depends on the cost estimation strategy. In such a scenario, attempting to provide access to only certain aspects of the CAD model, like features or solids, will most likely render the estimation process ineffective or severely limit the circumstances under which estimation can be performed. This was revealed during the examination of the product \textit{CostLink} which facilitates feature and part attribute data flow from CAD modelers to the cost modeler \textit{Cost Advantage}. By means of a map file specified by the user, CostLink extracts the specified data pertaining to the specified features from the CAD model. This mechanism is extremely restrictive in terms of selecting the specific information required to be extracted. In the absence of any knowledge about the kind of information to be extracted and to avoid imposing undue restrictions on the estimating strategy, a flexible mechanism is needed to extract information.

1.6 A Survey on Estimation Practices

In order to validate the above conclusions and gauge the status of estimating practices in industry, an informal survey was conducted among several aerospace, automotive and discrete parts industries. The information was collected on conditions of confidentiality and is summarised below.

1. Activity based costing has been attempted in several companies, but has not been successful.

2. Large aerospace industries primarily use parametric estimation. They provide poor per part estimates. However, when the number of parts in the product is in the order of 10,000–100,000 parts, accurate estimates are obtained.
3. Make/buy decisions on individual parts are made are on the basis of corporate policy or by operations based estimation and comparison of quotes.

4. Operations based estimation is viewed as the most accurate and prevalent method, but requires expertise and is costly.

5. Most large companies have developed custom costing software.

6. One large automotive company has developed a customized feature based estimation software. However, no direct links to CAD are in place and operation information still needs to be directly entered.

7. One large aerospace company has a custom costing system that obtains estimates using cost increase functions. This system produces large variations on single parts, but is accurate for products with multiple components. This system cannot be used with radically different designs or processes.

8. Feature based estimating is being pursued by R&D departments of several companies with a view to developing an engineering aid, rather than a corporate costing system. Some shortcomings of other methods in this regard are:
   
   • Parametric estimates do not give engineers the information needed to optimize the design of individual parts.
   
   • Operations based estimates are too expensive and difficult to produce and are of no use to product engineers.

9. As most of the above work is considered proprietary, it is not reaching the literature.

The results of the survey serve to bolster the findings of the literature search and confirm the absence of any flexible procedures for integrating the cost estimation
function into the design process. The survey also identifies the advantages offered by feature based estimation over conventional estimating procedures.

1.7 Problem Statement

This thesis aims to provide a direct, lossless means of extracting feature and cost driving data from CAD systems by the development of the CAD Query Language (henceforth CQL). At the completion of the proposed work, the major tasks that are expected to be implemented are:

1. Development of the syntax of CQL

2. Implementation of an interactive querying interface

3. Implementation of an embedded querying interface from the TNR (Templates and Rules) scripting language

4. Implementation of CAD data extraction methods for the Unigraphics CAD system

5. Implementation of conditional operators for filtering extracted data

6. Demonstrate examples using CQL

1.8 Problem Description

To successfully produce a cost estimate, a cost estimator needs CAD model data. In order for the estimate to be useful to the designer, it must relate costs to CAD model features. Past attempts at feature based estimation, in the absence of feature based modelers, had to devise their own mechanisms to aggregate geometric elements
into features. This either resulted in loss of information, as with feature recognition, or in significant discomfort to the designer, as with feature identification. In both cases, the complexity of the part to be estimated was restricted. The above problems may be overcome without restricting the complexity of the part to be estimated, by directly extracting feature data from CAD systems through a querying interface.
Chapter 2

The CAD System Abstraction

2.1 Introduction

Feature technology is still very much in the state of infancy. Several teething problems currently impede the widespread use of features [8]. Some of these issues confront CQL as well. In addition, a CAD model, in itself, is a veritable jungle of information. This chapter discusses the design approach to CQL and the development of the CAD Object Model, which is the basis of a multi-CAD CQL.

2.2 The Feature Mapping Problem

As discussed in the previous chapter, there is no uniform agreement over the concept of a feature. Different CAD systems possess different feature libraries. As a result, until a global standard is reached over the various possible features and their constitution, some form of mapping is required to bring the divergent views of features into conformance. Shah classifies the arising mapping situations as [8]:

**One-to-One** when no mapping is required
Variant Reparameterization different sets of dimensions are used to define the same feature in different domains

Discrete Aggregation two or more whole features combine to produce one feature

Discrete Decomposition one feature decomposes into two or more whole features

Conjugation a new feature is synthesized from two or more features

A conclusive solution to this problem is yet to be found. Gadh [13] suggests the use of pseudo-features called ‘C-loops’ during the process of design after which feature recognition is used to obtain the manufacturing features for process planning. Shah [14] describes mathematical techniques to transform the feature database of a modeling system to translate design features into machine and tool motions necessary for manufacturing. Most feature mapping efforts are currently centered around CAPP systems and the feature mapping problem is a major area of research in itself. Under the circumstances, a practical approach is to document the various features in use by different CAD systems and to design CQL’s feature library appropriately. In addition, variant reparameterization may be easily handled in CQL. However, for this implementation of CQL, the feature library is derived from the Unigraphics feature library.

2.3 Confronting Heterogeneity

Modern manufacturing systems are characterized by heterogeneity in terms of the nature and mode of operation of the various software and hardware systems in use. Apart from the CAD system, the estimating system may need to communicate with other manufacturing support systems like machining simulation systems,
process planning systems, feature recognition systems etc. These systems may employ several methodologies for data transfer ranging from simple file reads and writes to interprocess communication mechanisms. CQL is designed to be used in conjunction with the scripting language TNR (Templates and Rules) \(^1\). TNR possesses the following powerful mechanisms to facilitate software integration:

1. support for a variety of string handling routines to parse and create files of custom formats;

2. support for execution of SQL queries to store and manipulate data from relational databases like Oracle;

3. support for storage and manipulation of data in Part 21 files \(^2\) using the EXPRESS Query Language (henceforth EQL) \(^3\);

4. Interprocess communication using UNIX pipes and sockets.

By nature of this design, CAD system data may be accessed by any application via TNR and CQL.

### 2.4 Ad-hoc Information Extraction

From the investigation into the various cost estimating methods used, it is evident that estimating method inputs vary in nature and detail depending on the design stage. The development of CQL attempts to lay the framework for an ideal estimating system, wherein the designer may arrive at the best possible cost estimate with the

\(^1\)TNR was developed at the Center for Advanced Software System Integration (CASSI), Ohio University

\(^2\)Part 21 files contain text encoded in the object oriented EXPRESS data modeling format and are the format for STEP standards.

\(^3\)also developed at CASSI
available information, using the appropriate cost model. In this regard and for reasons discussed earlier, it is imperative to provide a flexible mechanism for data extraction. Query languages are well known to suit the above needs. Hence, CQL was modeled as a query language. At this point of time, CQL is envisioned as a means of data extraction alone and hence does not support the data definition function of a query language. In several cases, it may be necessary to select information from the CAD model based on certain conditions. For example, it may be desired to examine features from all solids with weight above a certain threshold. To facilitate such conditional selection of entities and data, CQL incorporates conditional operators. A detailed discussion of CQL operators is offered in the later chapters. At this stage, it suffices to say that by modeling CQL as a query language, almost any kind of CAD model information may be extracted painlessly.

2.5 The CAD Object Model

2.5.1 What is the CAD Object Model?

A query language is generally used in association with a schema to obtain the desired information from a database. The CAD Object Model, from the user’s point of view, serves the same function. In addition, it allows the user to treat all CAD systems as the same. All CAD systems share a fundamentally similar framework based on the need to manipulate the same kinds of geometry. The CAD object model clearly defines the various entities, geometric and non-geometric, that may be expected to be present in the CAD file of a generic CAD system, their inter-relationships and the cardinalities of these relationships. The following conventions have been used to represent the CAD Object model:
• solid, rectangular boxes represent concrete entities;

• solid arrowheaded lines connecting entities represent an inheritance relationship;

• dashed lines connecting entities represent a 'contains' relationship;

• the cardinality of the relationship is indicated by the use of the alphanumeric characters ‘1’,'n','z','p' at the ends of connecting lines where:

  1 indicates that for each instance of an entity at the other end of the relationship, there is exactly one instance of the entity at the end with the ‘1’;

  n indicates that for each instance of an entity at the other end of the relationship, there are zero or more instances of the entity at the end with the ‘n’;

  p indicates that for each instance of an entity at the other end of the relationship, there are one or more instances of the entity at the end with the ‘p’;

  z indicates that for each instance of an entity at the other end of the relationship, there are zero or one instances of the entity at the end with the ‘z’.

An EXPRESS schema for the CAD Object Model (figure 2.1) is provided in the appendix. Every instance of an entity in the CAD model is associated with a unique ID.
Figure 2.1: The CAD Object Model
2.5.2 Model Elements

At the highest level in the hierarchy of entities, is the Collection. The Collection is the equivalent of a CAD part file. A Collection contains Objects, Expressions and other miscellaneous entities. CQL provides several methods to access the entities in the Collection and associated information. A complete listing of these methods is provided in the appendix.

Objects CQL objects are a specific class of entities that encompass all geometric entities in the model like lines, curves, solids etc. and include some non-geometric entities like Notes and Dimensions. The term Object wherever mentioned is implied to be the CQL Object and is distinguished from the generic homonym used in Object Oriented Languages. Objects may have user defined attributes assigned to them. All instances of Object have an object-type which determines which of the various possible subclasses the particular instance belongs to.

Expressions Expressions represent the mechanisms used in CAD systems to establish parametric relationships. Expression instances have a name and a value. These comprise the left and right hand sides of an equation for the parametric relationship respectively. For example, a feature such as a block is characterized by three Expressions representing the length, breadth and height of the block. The value of the expression for length would represent the actual length of the block in the appropriate units. The name of an Expression is either a user specified string or a letter followed by integers. Expressions have unique names for a given Collection and like other entities have a unique ID.

Attributes Attributes are the means by which user defined properties may be assigned to a Collection and to each of its objects. At the Collection level, at-
tributes may be used to store product management data like the part number, organization, designer etc. At the Object level, attributes may be used to store information such as densities, material type, supplier information etc.

**PartsList** Some entities in the CAD model have been included because of their potential utility for cost analysis. *PartsLists* are some such entities. A *Collection* may contain a *PartsList* entity in case of an assembly. In such a case, the *PartsList* is a user-specified list of all the components of the assembly along with any user-specified information such as code number, supplier, quantity etc.

**FaceLoops** *FaceLoops* are another class of special entities that are seen to be potentially useful for tool-path length computation. A *FaceLoop* is an ordered set of edges that represents a boundary of a face. A single face may be associated with several such *FaceLoop* instances, which may either be internal *FaceLoops* or external *FaceLoops*. An important point to be noted is that *FaceLoop* instances are transient and vanish as soon as the corresponding part file is closed. *FaceLoops* are the only transient entities in the CAD model.

### 2.5.3 Model Example

To provide a concrete example of the interpretation of the CAD model, the case of the *feature* entity in the CAD model is considered here. In order to fully understand an entity's position in the CAD model, the CAD model should be examined in conjunction with the list of methods available for that entity and the EXPRESS schema provided in the appendix. In case of the *feature* entity, the corresponding methods are indicated in figure 2.2. The EXPRESS schema definition for the feature entity is shown in figure 2.3.
From the CAD model, it may be seen that a feature inherits the methods and attributes of the Object entity. Thus, a feature instance possesses an object-type and can have a name, in addition to its own attributes. By examination of the EXPRESS schema, in conjunction with the methods defined for the feature entity, it can be ascertained that a feature instance has the following attributes:

1. A feature-name which may be extracted by using the method featureName

2. A feature-type which determines the type of feature (boss, simple-hole, block etc.). The feature-type is represented by a string and may be extracted by using the featureType method.
3. A set of names of parameters that are required to define the feature (e.g., diameter and depth for a blind hole). These may be obtained by the method `paramNames`.

4. A set of values of parameters that the user specifies to define the feature during the process of modeling. These parameters may be obtained as an aggregate using the method `paramNames` or may be obtained individually, specifying the parameter name, with the method `paramValueByName`.

In addition to the above attributes of the feature, instance IDs of entities related to the feature may be obtained. From the CAD model, it may be seen that a feature has the following relationships:
1. A feature belongs to a single instance of a solid. This is represented by the 1 to many relationship between *solid* and *feature*. This also means that a solid may be associated with one or more features. The ID of the corresponding solid may be obtained with the method *getSolid*.

2. A feature entity contains one or more instances of *BrepElem* (Boundary Representation Elements, namely, faces and edges). To all first appearances, it seems that the exact minimum number of BrepElems necessary to compose a feature should be determined and represented in the CAD model. In other words, a feature should be shown as associated with x-p BrepElems instead of just p BrepElems, where x is the minimum. However, doing so would constrain the definition of a feature across CAD systems and is hence not attempted. These may be obtained using the *getBrepElems* method.

3. A feature may be produced by extruding a sketch. Thus, a feature is shown sharing a many-many relationship with sketch instances. It should be noted that by nature of an n-n relationship, a feature is not necessarily associated with any sketch and vice-versa. This permits this CAD model to remain valid in a CAD system wherein sketches cannot be used to construct features. The *getSketches* method obtains the IDs of the associated sketches.

4. A feature may be dependent on other features. For example, a boss constructed on a block is considered a child feature of the block. From the point of view of the boss, the block is its parent feature. This particular example is extremely trivial. In a real-world CAD model, feature hierarchies often span multiple levels. As the exact tree structure representing the relationship between any two features, separated by several levels, may be CAD system dependent, only
the general notion of parent and child is represented here. The desired members
are extracted using the `getParentFeatures` and `getChildFeatures` methods.

5. As mentioned earlier, Expressions are the mechanism to store parametric rela-
tionships in the CAD system. Thus an alternative to accessing the parameters
of a feature, say a block, by using the `paramValues` method would be to identify
the Expression instances associated with the feature and extract their names
and values. The `getExps` and `getExpByName` methods are useful in this regard.
This approach is of great value in the extraction of sketch parameters.

6. In order to represent the capability of a user to define a new kind of feature
according to his needs, the concept of a User Defined Feature (UDF) is shown
in the model. A UDF inherits all the methods of a feature and may have
additional methods of its own. In the present implementation of CQL, UDFs
are not functional as the on-site Unigraphics software bundle does not come
enabled with the UDF module, which is available at additional cost.
Chapter 3

CQL: Syntax and Use

3.1 Introduction

As a result of the discussions in the previous chapter, it is seen that a CAD part file may be likened to an object oriented database. CQL’s syntax therefore provides ways to specify the objects to be extracted and the methods to be executed on these objects to obtain CAD data. Every object in a CAD part file has a unique object ID, that it retains throughout the course of a querying session. Object IDs are also unique across concurrently open CAD files. A CQL query results in object IDs and/or data obtained from objects. This data could be in the form of a scalar (integers, real numbers, character strings, boolean values, null character) or aggregates of scalars.

Currently, CQL queries may be used only to extract data from the CAD part file. Though, the addition of new objects or modification of existing objects in the CAD file poses no major problems, this is not of interest at present. A CQL query may thus belong to one of the following categories:

Select Query Select queries serve to extract CQL objects/object data from the CAD part file.
System Query System queries provide the means to manipulate CAD part files (opening, closing) and to control a CQL session.

Suppress/Unsuppress Query These are special queries that lead to the suppression/unsuppression of selected CAD model features and help in feature volume computation.

The Select query is the most important type of query and consists of three distinct clauses, the Select clause, the From clause and the optional Where clause. In addition, implicit select queries do not possess the From clause. This chapter examines the basic building blocks of queries, discusses these clauses and ends with examples illustrating some important uses of CQL queries.

3.2 Query Elements

CQL queries, at the most basic syntactical level, use entities, methods, variables and constants to define the data to be extracted. These are, in turn, used to construct object expressions, method extensions and variable extensions, which in conjunction with CQL keywords and conditional operators form a CQL query. This section describes the basic elements of a query.

Entities CQL entities represent all possible CAD entities that may be present in the CAD part file. In the context of the CAD object model, they are comprised of objects, expressions, PartsLists and FaceLoops. They are indicated in the grammar by a set of eponymous keywords. Some examples of entities are Face, Edge, Object, Curve, Sketch and Feature. A given entity, as used in a query, represents an aggregate of the object IDs of all instances of that entity present in the CAD part file.
**Variables** Variables are user specified character strings that represent a single unit or aggregate data. Variable names must start with an alphabet or underscore and may be followed by alphanumerical characters and underscores. It is illegal for variables to have the same names as CQL keywords. Variables are used in all parts of the CQL query and their use is illustrated in the examples to follow.

**Constants** Constants in CQL may be integers, real numbers, character strings, boolean values (‘.T.’ representing TRUE and ‘.F.’ representing FALSE), the null character (‘$’) or an aggregate constant. Constants are typically used in conditional expressions to compare the result of a method with a desired value. The null character indicates that the attribute that it represents is undefined. Aggregate constants are represented in CQL as a comma-separated list of unit constants, surrounded by parentheses. For example, ‘(365,366,367,368)’ could represent an aggregate constant of object IDs. Aggregate constants form a handy mechanism to make a query operate on a group of selected object IDs. They could also be used in conditional expressions to verify if the result of a query belongs to a set of pre-defined values. One important thing to note is that CQL does not support nested aggregates. The grammar rules for constants are shown in figure 3.1.

**Methods** Methods are the means to extract entity information. All CQL methods operate on CAD entities only. Hence, a method cannot be applied on data. The execution of a method in a query can result in object IDs or data. CQL syntax allows methods to be concatenated, thus making it possible to chain several methods together. However, since methods can only be applied to entities, only those methods that produce object IDs as their result can have other methods following them. Since, the syntax does not differentiate between methods based
PrimitiveList : PrimitiveValue
  | PrimitiveList ',', PrimitiveValue
;
PrimitiveValue : INTEGER_TOKEN
  | REAL_TOKEN
  | STRING_TOKEN
  | TRUE_TOKEN
  | FALSE_TOKEN
  | DollarValue
;
Aggregate : '(' ')'
  | '(' PrimitiveList ')'`
;
DollarValue : '$'
;

Figure 3.1: Grammar Rules for Constants

on their output, this is enforced at run-time. CQL methods can be classified as either single valued or multi-valued, depending on whether they return a single scalar/object ID as their result, or an aggregate. This differentiation between methods arose in the syntax because of the need to enforce the correct use of conditional operators. Multi-valued methods may be distinguished from Single-valued methods by the presence of the ‘s’ suffix in their names, indicating multiplicity of output. Some examples are getSolid, volume, featureName (Single-valued), getObjects, getGroups and getDimensions (Multi-valued).

3.3 Important Constructs

CQL syntax allows variables and entities to be associated with methods to form a number of expressions. This is achieved through the use of two important operators:
• The dot operator (‘.’)

• The arrow operator (‘→’)

Both these operators indicate that the method following them is to be executed on the instances obtained from prior portions of the expression. The arrow operator is used when a method is to be executed on a variable, as in ‘x→getSolid’, while the dot operator is used when a method is to be executed on an entity or is chained to another method, as in ‘Feature.featureName’ and ‘Feature.getExps.expName’. The arrow operator is a convention adopted from the language EQL, after whose syntax CQL is closely modeled. While it is not necessary to devise the arrow operator to define an unambiguous grammar for CQL, its use makes it explicit that a method is being executed on a variable and not on an entity or method.

3.4 The From Clause

The from-clause forms the most important part of the select and suppress queries. It serves to clearly identify the set of entities or data to be extracted or on whom the required methods are to be applied. The grammar of the from-clause is shown in figure 3.2.

The from-clause is frequently used to construct a working set of data extracted from the CAD part file, so that it can be manipulated by the remaining portions of the query. A simple example would be:

CQL> select x from x in Feature;

This query results in the extraction of the object IDs of all instances of the Feature entity from the CAD part file. The from-clause, in this case, establishes the working
FromStat : FROM_TOKEN VariableDecl;
ImpliedFromStat : FROM_TOKEN ObjectExpression;
VariableDecl : VariableDecl | VariableDecl , VariableDecl;
VariableDecls : VariableDecl | VariableDecls , VariableDecl;
ObjectExpression : Object | Object , MethodExtension;
MethodExtension : SingleValMethodExt | MultiValMethodExt;

Figure 3.2: Grammar Rules for the From and Implied-From Clauses
set of data as the set of all object IDs of features in the CAD part file. To be more specific, it establishes ‘x’ as a variable that iterates over every feature instance ID in the part file. The from-clause may also be used to establish a variable as an alias for a complex expression, after which the variable may be substituted in place of the expression in the query. For example, in the query

CQL> select x, y from x in Solid, y in x->getFeatures.name;

the from-clause establishes the variable ‘y’ as an alias for the expression ‘x->getFeatures.name’ in the query. A variant of the from-clause is the implied-from-clause, devised to allow simple queries involving a single variable, to be framed without the use of any variables whatsoever. This allows the query to be intuitively simple. For example,

CQL> select x->featureName from x in Feature;

may be replaced equivalently by the query

CQL> select featureName from Feature;

wherein the implied-from-statement is used. The grammar rules for the implied-from-clause are shown in figure 3.2.

3.5 The Where Clause

Very often, the working set of data constructed by the from-clause needs to be filtered based on some user-defined conditions. This is achieved by the use of the
where-clause. The grammar for the where-clause is shown in figure 3.3. To facilitate this filtering process, a variety of conditional operators are provided in CQL. Atomic conditions, with the exception of the for-condition, operate on either one or two operands. Compound conditions, resulting from the combined effect of several atomic conditions, may also be constructed. The following sections discuss the various conditions that may be tested for using CQL. Figure 3.4 shows the various kinds of atomic conditions. All the conditions described below, with the exception of the For condition have ‘implied’ counterparts that behave identically.

From figure 3.3, it may be observed that the where statement may assume one of two forms, the regular-where statement and the explicit-where statement. The use of the regular-where statement results in faster query execution as compared to the explicit-where. However, a side-effect is that the order in which parts of a compound condition execute may not be that intended by the user. For this reason, the explicit-where statement is provided whereby parts of a compound condition execute exactly as specified by the user. The only difference in the syntax is that, in case of the explicit-where statement, the keyword ‘Where’ is replaced by ‘ExpWhere’. Details about the differences in these two queries are provided in section 4.3.3.
3.5.1 The Null Condition

The Null condition takes a single operand and tests if the operand is defined. Figure 3.5 shows the grammar of the Null condition. Operands with undefined values are of the type null and are represented internally in CQL as a separate data type with value ‘$’. In this regard, they are different from characters or strings with value ‘$’. Thus a Null condition checks if the data type of the operand is of the dollar data type. Depending on whether the IS-NULL or the IS-NOT-NULL operator is used, the appropriate boolean result is returned. This construct is generally used to check if optional attributes of objects are defined. For example, objects may be assigned user-defined names. The presence of an object name may be tested by this condition.

3.5.2 The ISA Condition

Every entity instance in the CAD part file has an integral ‘type’ which identifies its class in the CAD Object Model. Since the CAD object model incorporates inheritance relationships, this ‘type’ refers to the most specific class to which the particular
NullCondition : Operand IS_NULL_TOKEN
   | Operand IS_NOT_NULL_TOKEN
   ;

Operand : VariablePrefix
   | VariablePrefix ARROW_TOKEN
   SingleValMethodExt
   | PrimitiveValue
   ;

Figure 3.5: The Null Condition

ISACCondition : Operand IS_A_TOKEN Object
   | Operand IS_NOT_A_TOKEN Object
   ;

Figure 3.6: The ISA Condition

instance belongs. The ISA/IS-NOT-A operators return a boolean result depending
on whether the particular instance is a member of the family of the specified class. 
Figure 3.6 shows the grammar of the ISA condition. For example, an instance of
the LineEdge object would be considered to be an Object, a BrepElem, an Edge and
a LineEdge, all at the same time, because of the inheritance relationships involved.

3.5.3 The Empty Condition

The Empty operator is used to determine if an aggregate contains non-zero members. 
Figure 3.7 shows the grammar of the Empty condition. A boolean result is returned
depending on whether the aggregate contains any members. Thus the aggregate ‘( )’
would be deemed empty. It should be mentioned that while attempting to check if an
aggregate is empty, no attempt is made to check if its members are defined. Thus an
aggregate, all of whose members are ‘$’ would be deemed non-empty. If an attempt
is made to test if an undefined aggregate is empty, a runtime error results.

3.5.4 The IN Condition

The In operator takes an operand and checks to see if the operand is present in a
specified aggregate. Since the left hand side of the condition cannot be an aggregate
from the grammar shown in figure 3.8, the presence of an aggregate inside an ag-
gregate of aggregates cannot be checked with this construct. However, this situation
is not expected to arise in the context of data extracted from a CAD part file. An-
other important point to note is that, in the event of either the unit operand or the
aggregate being undefined (being of the dollar internal type), both the In and Not-In
operators return FALSE. All comparisons between the unit operand and the members
of the aggregate operand for equality conform to the rules discussed in section 3.5.6.
3.5.5 The Contains Condition

The Contains condition takes two aggregate operands and verifies if all the members of the aggregate on the right are present in the aggregate on the left. Figure 3.9 depicts the grammar of the Contains condition. It must be noted that in case of multiple occurrences of a member on the right, at least as many members must be present on the left for the Contains condition to be satisfied. To elaborate, consider three aggregates ‘a’=(1,2,3), ‘b’=(1,1,2,3) and ‘c’=(1,1). ‘c’ is contained in ‘b’, but is not contained in ‘a’. As with the In condition, if any of the members of either of the aggregates is undefined, or if either of the aggregates themselves is undefined, both the Contains and the Not-Contains conditions return FALSE.

3.5.6 The Comparison Condition

The comparison condition takes two operands and compares their values using the comparison operator indicated. Figure 3.10 shows the grammar of the Comparison condition. The comparison operators supported are the ==, !=, >, <, >= and <= operators. Operands of different types may be compared in cases where the outcome of the comparison is unambiguously defined. For example, an integer may be compared to a real number for equality. However, comparing, say, the integer 2
ComparisonCondition : Operand EQUAL_TOKEN Operand
| AggrOperand EQUAL_TOKEN AggrOperand
| Operand NE_TOKEN Operand
| AggrOperand NE_TOKEN AggrOperand
| Operand '<' Operand
| Operand '>' Operand
| Operand GE_TOKEN Operand
| Operand LE_TOKEN Operand
;

Figure 3.10: The Comparison Condition

Table 3.1: Legal Data Types For Comparison

<table>
<thead>
<tr>
<th></th>
<th>bool</th>
<th>int</th>
<th>real</th>
<th>str</th>
<th>$</th>
<th>aggr</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>str</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>aggr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

with the string "2" is not unambiguously defined. Hence, it is not supported. In order to allow values of optional attributes to be compared, comparison with the null value is supported and always results in FALSE. All legal comparisons are shown in Table 3.1. Aggregates may be compared with each other for equality/inequality. Two aggregates are equal if and only if they are identical to each other.

### 3.5.7 The For Condition

One of the most powerful conditional constructs available in CQL is the For condition. With the For condition, the members of an aggregate, can each be tested against
ForCondition : FOR_TOKEN VariableDecls ALL_TOKEN
                '(' Condition ')' |
                FOR_TOKEN VariableDecls ANY_TOKEN
                '(' Condition ')' |

;  

Figure 3.11: The For Condition

a condition, in turn, and a suitable action can be performed. The syntax of the
For condition is shown in figure 3.11. During the execution of the For condition,
variables are established to iterate over a set of values by means of the variable
declaration (VariableDecl). These are used to test for the occurrence of a specified
condition repetitively. For conditions are of two kinds, namely the For-all and For-
any conditions. The For-all condition returns TRUE only if the test condition is
satisfied for all members in the established data set. The For-any condition returns
TRUE as soon as the test condition is satisfied for a member of the data set.

3.5.8 The Match Condition

The Match condition is a powerful means of identifying patterns in character strings.
Figure 3.12 depicts the syntax of the Match condition. Regular expressions are ways
of specifying character strings. By using regular expression syntax to construct a
template pattern to be identified, the Match condition may be used to check for the
presence of the template pattern in the given string operand. The regular expression
corresponds to the right hand operand in the Match condition. The Match condition
may only be used on string operands. Use with an undefined operand returns FALSE.
3.5.9 Compound Conditions

All the atomic conditions above may be logically combined together using the ‘And’, ‘Or’ and ‘Not’ operators as shown in figure 3.13 to form compound conditions. One important point to note in case of compound conditions is the precedence of operators. The Not operator has the highest precedence and associates from right to left. This is followed in precedence by the And operator followed by the Or operator. Both the And and the Or operators associate from left to right. Precedence of conditional operators may be explicitly specified by grouping conditions using parentheses.

3.6 The Select Clause

After the required data has been extracted by the execution of the from clause and optionally filtered by the where clause, the select clause is used to specify the format and the portions of the working data set to be retrieved. The various forms of the
Select clause are shown in figure 3.14. The data to be selected is specified using select expressions. The syntax allows several such select expressions, separated by commas to be specified. In cases where several similar structured select expressions are employed, they may be grouped together using parentheses. For example,

CQL> select x->getFeatures.featureName, x->getFeatures.featureType
    from x in Solid;

could be equivalently abbreviated to

CQL> select x->getFeatures(featureName, featureType) from x in Solid;

The Implied-Select statement can be used in case of very simple queries where only one variable is needed. Such queries can be simplified further by the use of the No-From-Select which does away with the From clause.

3.7 The Select Query

The select query is the most frequently used query in CQL. From the discussion of the Select clause, it can be inferred that there are three variants of the Select query. Figure 3.15 depicts these variants. The most important form consists of the Select clause, the From clause and the optional Where clause. In case of very simple queries, the Implied and No-From forms of the select query can be used. To illustrate the use of the different variants of the select query, consider the following examples:
SelectStat : SELECT_TOKEN SelectExps
    ;
NoFromSelectStat : SELECT_TOKEN NoFromSelectExp
    ;
ImpliedSelectStat : SELECT_TOKEN MethodExtensions
    ;
SelectExps : SelectExp
            | SelectExps ',', SelectExp
    ;
SelectExp : VariableExtension
           | VariableExtension '(' MethodExtensions ')' ;
NoFromSelectExp : ObjectExpression
                 | ObjectExpression '(' MethodExtensions ')' ;

Figure 3.14: The Select Clause

SelectQuery : SelectStat FromStat OptionalWhereStat
             | NoFromSelectStat
             | ImpliedSelectStat ImpliedFromStat
    ;

Figure 3.15: The Select Query
Regular Select
CQL> select x->volume from x in Solid;
CQL> select x(name, volume) from x in Solid
   where x->volume > 2.0;

Implied Select
CQL> select volume from Solid;
CQL> select name, volume from Solid
   where volume > 2.0;

NoFrom Select
CQL> select Solid.volume;
CQL> select Solid(name, volume);

The above examples show how a select query involving one variable may be equivalently written in different forms. The NoFromSelect may only be used when there is no where-clause in the query. As a result, the NoFromSelect example shown above cannot check for the volume of the solid. Also note how multiple methods may be specified by the use of parentheses.

3.8 The System Query

System queries are not really queries, but in reality are commands used to perform administrative tasks. The main kinds of System queries are the Open, Close, Cad, Quit and SetSearchDirs queries as depicted in figure 3.16.

3.8.1 Cad

The Cad query is used to specify the CAD system whose files will be operated on. This query consists of the ‘cad’ keyword followed by the name of the CAD system to operate on, or a portion of the name. Currently the only CAD system on which CQL
is functional is Unigraphics. As CQL is extended to work with other CAD systems, this command will enable moving between CAD systems. Unless this query is first used to specify the CAD system to operate on, no files can be opened. The execution of this query causes the software license manager daemon for the corresponding CAD system to be contacted for a license. In the event of expiry of the license or exhaustion of all available licenses, this query fails.

3.8.2 Open

The Open query consists of the ‘Open’ keyword followed by the name of the CAD part file to open for querying. The name of the CAD part file can be in the form of an absolute path name, relative path name or just the name of the part file if the file is present in the current working directory. CQL allows several part files to be open at the same time. However, only one of them can be active at any given time. All non-system queries are directed at the most recently opened CAD part file. Files opened earlier can be reverted to at any time by issuing a fresh Open command. When an assembly part file is opened, the CAD part files containing the component parts are also implicitly opened. If the component part files reside in a directory
other than that of the assembly file, then the search paths must be specified with the setSearchDirs command, else the assembly file cannot be opened.

3.8.3 Close

The Close command results in the closing of all open part files. Since, CQL is currently functional only on Unigraphics, this implies the closure of all Unigraphics part files. This is primarily used to make memory available to load new part files. Closing a file does not result in any changes being saved. CQL does not provide means to permanently modify part files.

3.8.4 Quit

This command consists of either the ‘Quit’ or the ‘Exit’ keyword and results in the termination of the querying session. At this time, all software licenses obtained are returned.

3.8.5 SetSearchDirs

This command sets the directory paths to be searched for component part files when an assembly file is opened. From the syntax in figure 3.16, it can be seen that a comma separated list of path name strings is provided. When an assembly file is opened, these directories and all their sub-directories are recursively searched for component part files. Hence, the path name strings supplied must be as specific as possible to avoid time consuming searches for components. Use of this command overwrites any existing search paths with the new paths specified.
3.9 The Suppress Query

The Suppress query resulted from the need to devise a mechanism to compute the volumes of features. This is based on the concept that suppression of a feature causes its contribution to the volume of the parent solid to be ignored by the CAD system. Thus, knowing the volume of the parent solid before and after suppression of the feature of interest, the volume contributed to the solid by the feature can be determined. Both the suppress and unsuppress queries return the IDs of all features in the suppressed state in the part file at the end of the query. When a feature is suppressed, all its child features are also suppressed. For example, if a boss, with another boss on it, is suppressed, the CAD system ignores the volume contribution of both bosses to the solid volume. Hence, the contributions of the feature and all its children to the volume of the solid are ignored by the CAD system.

Interactions of features pose another problem to the computation of feature volumes. Consider the case of two intersecting holes in a block. The volume contributed to the block by a hole does not equal the volume of the hole, if determined by the above algorithm. Another point to note is, that if all the features on a solid are suppressed, the solid itself is suppressed and becomes invisible to queries. Any queries made using the object ID of the solid obtained before suppression result in errors. Thus, the above factors must be considered while interpreting the feature volumes computed by suppression.

The syntax of the suppress query is very similar to that of the select query, as shown in figure 3.17. The suppress query can only be applied to feature object IDs. Attempting to suppress non-features results in a run-time error.

The effect of a suppress can be undone with an Unsuppress query. The syntax is identical to that of the suppress query. In addition, a blanket unsuppression of
SuppressQuery : SuppressStat FromStat OptionalWhereStat | NoFromSuppressStat | ImpliedSuppressStat ImpliedFromStat


SuppressStat : SUPPRESS_TOKEN SelectExps

NoFromSuppressStat : SUPPRESS_TOKEN NoFromSelectExp

ImpliedSuppressStat : SUPPRESS_TOKEN MethodExtensions

UnsuppressStat : UNSUPPRESS_TOKEN SelectExps

NoFromUnsuppressStat : UNSUPPRESS_TOKEN NoFromSelectExp

ImpliedUnsuppressStat : UNSUPPRESS_TOKEN MethodExtensions

Figure 3.17: The Suppress and Unsuppress Queries

all suppressed features can be performed by just using the keyword ‘Unsuppress’ by itself. It must be noted that unsuppressing a parent feature does not unsuppress its children. On the other hand, unsuppressing a child forcibly unsuppresses its parent features.

The behaviour of the suppress query may be better understood with an example. Consider the sample part shown in figure 3.18. The part consists of a simple block with two rectangular pads, one on top of each other. The effect of a suppress query is visible as a change in volume of the suppressed solid. For the given part, the block (id: 650) has volume 4 cu in (2x2x1), the large pad (id: 386) has volume 0.5
Figure 3.18: Example Part for Suppress

cu in (1x1x0.5) and the small pad (id: 387) has volume 0.0625 cu in (0.5x0.5x0.25), resulting in a total volume for the solid of 4.5625 cu in. This can be seen using the volume method as shown in figure 3.19.

At first, the smaller pad is suppressed and the volume of the solid is checked. The new volume of 4.5 cu in indicates that the volume contribution of the smaller pad to the solid (0.0625 cu in) has been discounted. This is corroborated by the fact that the output of the suppress query indicates that feature 387 (the small pad) is in suppressed state. An unsuppress at this stage unsuppresses the small
CQL> select x, x->featureName from x in Solid.getFeatures;
650 BLOCK(0)
386 RECTANGULAR_PAD(7)
387 RECTANGULAR_PAD(8)
CQL> select Solid.volume;
4.5625
CQL> suppress x from x in (387);
387
CQL> select Solid.volume;
4.5
CQL> unsuppress;
CQL> suppress x from x in (386);
386
387
CQL> select Solid.volume;
4
CQL> unsuppress x from x in (386);
387
CQL> select Solid.volume;
4.5
CQL> unsuppress;
CQL> select Solid.volume;
4.5
CQL> unsuppress;
CQL> select Solid.volume;
4.5625
CQL> suppress x from x in (650);
650
386
387
CQL> select Solid;

Figure 3.19: Suppress Query Example
pad. Suppressing the large pad causes both the large pad and the small pad to be suppressed, as the small pad is a child feature of the large pad. This is reflected in the solid volume of 4.0 cu in, which is the volume of the block alone. Unsuppressing the large pad does not automatically unsuppress the small pad, as is reflected in the volume after unsuppressing the large pad (4.5 cu in). If the small pad had been unsuppressed instead of the large one, both pads would have been unsuppressed simultaneously.

If a global unsuppress is performed and the block is now suppressed, all features in the solid are suppressed. This is because the child of the block, the large pad is unsuppressed, suppressing the small pad in turn. Due to all features in the solid being suppressed, the solid vanishes. The last query to select all solids in the part file does not yield any results indicating that the solid has vanished. Any query performed at this stage using the solid id obtained earlier will result in a runtime error. An exception to the above behaviour may be observed in the case of boolean features (feature-type META), resulting from unites, substracts or intersections of features, when suppressing the boolean feature suppresses the parent features.

It must be noted that the suppress query is a time-consuming query as the solid model is required to be updated by the CAD system after each suppress/unsuppress. Hence, program performance will be greatly improved if all the features to be suppressed are aggregated and suppressed using the minimum number of queries. Care has been taken to ensure that the sequence in which a given set of features is suppressed does not affect the outcome of the suppress query.
3.10 Using Embedded CQL Queries

CQL is designed to allow queries to be embedded in the programming language TNR. TNR is a scripting language developed at the Center for Advanced Software Systems Integration (CASSI), Ohio University. It has been designed specifically for the purpose of enabling software integration and supports interfaces to several data resources [15]. This includes ASCII text files, relational databases, STEP files and computer networks. With the support of CQL, TNR programs can be used to extract CAD data by issuing embedded CQL queries. This data can then be exchanged with other software applications via one of TNR’s many interfaces. Three main TNR constructs enable the use of CQL queries:

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

**cad_part(partName)** This construct specifies the part to open for querying.

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

This section presents some examples of the most useful and frequently used queries in CQL. To do so, three CAD part files have been employed. The first, ‘features.prt’, consists of a block with 4 simple holes and a boss and is used to introduce the use of CQL queries. The part ‘sheet_demo.prt’ is a sheet body and is used to illustrate queries used to extract sketch information. To show some queries used on assemblies, the part ‘assy.prt’ is used. Detailed descriptions of all methods are provided in the appendix.

Consider the part ‘features.prt’ shown in figure 3.20. The following queries demonstrate some of the useful information available from this part.

Example 1 Specify the CAD system as Unigraphics.

CQL> cad "ug";

Example 2 Open the part file “features.prt”.

CQL> open "/home/research/lex/ug/features";

Example 3 Select all Solids in the part file and display their IDs and volume.

CQL> select x, x->volume from x in Solid;
307 3.8232854
Figure 3.20: features.prt
Object IDs of the solids selected are returned. In this case, there is only one solid in the part file with ID 307 with volume 3.8232854 cubic inches.

**Example 4** Select all features from the block and display their object ID, feature-name and feature-type.

```cql
CQL> select x, x->featureName, x->featureType
    from x in Solid.getFeatures;

514 BLOCK(0) BLOCK
384 BOSS(3) BOSS
385 INSTANCE[0](4)/SIMPLE HOLE(4) INSTANCE
517 RECTANGULAR_ARRAY(5) LINEAR_ISET
518 INSTANCE[0,1](5)/SIMPLE HOLE(4) INSTANCE
519 INSTANCE[1,0](5)/SIMPLE HOLE(4) INSTANCE
520 INSTANCE[1,1](5)/SIMPLE HOLE(4) INSTANCE
```

Nine features are obtained in this case. The LINEAR_ISET is a feature produced when features are generated by instancing. In this case, a simple-hole was first modeled and was used to generate the remaining three simple-holes by constructing an array of simple-holes. Thus the name of the instance, contains the name of the simple-hole from which it was generated.

**Example 5** Select all the child features from features of type "BLOCK".

```cql
CQL> select y, y->featureType from x in Solid.getFeatures,
    y in x->getChildFeatures where x->featureType == "BLOCK";

384 BOSS
516 SIMPLE HOLE
385 INSTANCE
As a result of execution of the from-clause, the variable ‘x’ iterates over all feature IDs obtained from the Solid. For each such feature ID, the variable ‘y’ iterates over the IDs of child features obtained from ‘x’. The where-clause ensures that the selection is performed only when the feature-type equals the string "BLOCK". In reality, the execution is slightly different because of the optimization of the where-clause. However, the effect on the selection results is the same.

**Example 6** Select the names and values of the parameters of all bosses in the part.

```
CQL> select x(paramNames, paramValues) from x in Feature
   where x->featureType == "BOSS";
(DIAMETER,HEIGHT,TAPER) (0.5,0.1,0)
```

**Example 7** Select the IDs and value of the lengths of all blocks.

```
CQL> select x, x->paramValueByName(Length) from x in Feature
   where x->featureType == "BLOCK";
514 2
```

Only one feature of type BLOCK is present with length 2 inches.

**Example 8** Determine the volume of the boss by suppression

```
CQL> select x->getChildFeatures from x in Solid.getFeatures
   where x->featureType == "BOSS";
()
CQL> suppress x from x in Solid.getFeatures
```
where x->featureType == "BOSS";
CQL> select Solid.volume;
3.8036505
CQL> unsuppress;

Firstly, it is verified that the boss has no children whose volumes will contribute to the solid. Now it is suppressed and the solid volume is extracted. The difference between the volume before suppression and the suppressed volume yields the volume of the boss.

Example 9  Open the file "sheet_demo.prt" shown in figure 3.21 and select the IDs and names of all sketches present.

CQL> open "/home/research/lex/ug/sheet_demo";
CQL> select x, x->name from x in Sketch;
2316 SKT000
2320 SKT001
2506 SKT002
2507 SKT003
3300 SKT004
3247 SKT005
3068 SKT006
3069 SKT007
3176 SKT009
2696 SKT010
2697 SKT011
2797 SKT012
2798 SKT014
2827 SKT015
2828 SKT016
2874 SKT017
2875 SKT018
Figure 3.21: sheet_demo.prt
Sheet bodies can be made by extruding sketches. The part shown consists of a solid made up of several sweep features (extrusions). Each of these extrusions has a cross-section corresponding to a sketch. Thus the cross-section dimensions can be obtained by extracting the sketch parameters.

**Example 10** Select all solids from the part file.

```cql
CQL> select Solid;
3837
```

**Example 11** Select all the features and their types comprising the solid.

```cql
QL> select x, x->featureType from x in Solid.getFeatures;
7298 BLOCK
7299 SWP104
7300 SWP104
7313 SWP104
7314 SWP104
7304 SWP104
7315 SWP104
7301 SWP104
7302 SWP104
7312 SWP104
7303 SWP104
7311 SWP104
7310 SWP104
7309 SWP104
7308 SWP104
7307 SWP104
7306 SWP104
7305 SWP104
```

The feature-type ‘SWP104’ obtained corresponds to an extrusion.
Example 12 Get the sketches corresponding to each extrusion and display the ID of the extrusion, ID of the corresponding sketch and the names and values of sketch parameters.

CQL> select x, y, y->getExps(name, value)
from x in Solid.getFeatures, y in x->getSketches
where x->featureType == "SWP104";

7299 2316 (Cut_Length1_1, Cut_Length2_1, Rad1_1, Cut_Length3_1,
Cut_Length4_1, Cut_Length5_1, Cut_Length6_1,
Angle1_1) (0.01, 0.007, 0.005, 0.007, 0.01, 0.022,
0.022, 90)

7300 2506 (Cut_Length1_3, Cut_Length2_3, Cut_Length3_3,
Cut_Length4_3, Rad1_3, Rad2_3, Cut_Length5_3,
Cut_Length6_3, Cut_Length7_3, p302, Cut_Length8_3,
Cut_Length9_3, Angle1_3, Angle2_3) (0.01, 0.0175,
0.02, 0.022092807, 0.0025, 0.0025, 0.049336501,
0.018, 0.01, 136.01892, 0.090093831, 0.087260292,
90, 90)

7313 2507 (Cut_Length1_4, Rad1_4, Cut_Length2_4, Angle1_4,
Angle2_4, Cut_Length5_4, Cut_Length6_4, Rad2_4,
Cut_Length4_4, Cut_Length5_4, Cut_Length7_4, Cut_Length3_4,
Cut_Length9_4, Cut_Length8_4) (0.01, 0.003, 0.025,
45, 45, 0.029757359, 0.025, 0.003, 0.25, 0.01,
0.029757359, 0.21901934, 0.21901934)

7314 2320 (Rad1_2, Cut_Length1_2, Cut_Length2_2, Cut_Length3_2,
Cut_Length4_2, Cut_Length5_2, Cut_Length6_2,
Angle1_2) (0.005, 0.01, 0.01, 0.007, 0.007, 0.022,
0.022, 90)

7304 3247 (p446, p447, p448, p450, p452, VWidth_2, BendAngle_2)
(0.001, 0.007, 0.007, 0.003, 0.002, 0.01, 90)

7315 3069 (p408, p409, p410, p412, HWidth_2, p414, BendAngle_4)
(0.001, 0.007, 0.007, 0.003, 0.01, 0.002, 90)

7301 3068 (p393, p394, p395, p397, HWidth_1, p399, BendAngle_3)
(0.001, 0.007, 0.007, 0.003, 0.01, 0.002, 90)

7302 3300 (p485, p486, p487, p489, VWidth_1, p491, BendAngle_1)
(0.001, 0.007, 0.007, 0.003, 0.01, 0.002, 90)

7312 3176 (p427, p428, p429, p431, Width, p433, BendAngle_5)
(0.001, 0.007, 0.007, 0.003, 0.01, 0.002, 90)
The above output has been formatted manually for purposes of clarity, without disturbing content. Each section of the output above beginning with an integer corresponds to a row of the actual query output. As is seen here, in case of sketches, by assigning meaningful names to the expressions that store sketch parameters, they can be interpreted easily from the query output. Each row in the query output can be interpreted as the ID of the extrusion, followed by the ID of the associated sketch, the names of the sketch expressions and their corresponding values. The value of each sketch expression represents the length of a curve on the perimeter of the sketch.

**Example 13** Select all sketches that have a parameter `Cut.Length5` and display the sketch name, parameter name and the parameter value.

```cql
CQL> select x->name, y(name, value) from x in Sketch,
    y in x->getExps
    where for z in y->name any (z match "Cut.Length5.*");
SKT000 Cut_Length5_1 0.022
SKT001 Cut_Length5_2 0.022
SKT002 Cut_Length5_3 0.049336501
SKT003 Cut_Length5_4 0.029757359
```
This example demonstrates the use of the for-comparison-condition. In this case, all the names of the expressions for a given sketch are cycled and checked for a match with the specified string. When a match is found, the select clause is executed. This also demonstrates the usefulness of regular expressions.

**Example 14** Open the file "assy.prt" and determine if it is an assembly part. The part is shown in figure 3.22.

```cql
CQL> open "assy";
CQL> select Collection.getRoot;
8002
```

All assembly parts have an entity called the 'Root-Instance' which is at the head of the component hierarchy. The presence of the 'root instance' implies that a part is an assembly.

**Example 15** Select the components of the assembly.

```cql
CQL> select x, x->instanceName from x in Instance;
8000 BASE
8001 SLIDER
```

The part "assy.prt" is a simple assembly of a base and slider to form part of a bench vice assembly. Assembly components share parent-child relationships in the component hierarchy. The getChildren and getParent methods help to identify these relationships.
Example 16 Determine the parent and children of the base and slider instances.

CQL> select x->instanceName, x->getParent, x->getChildren
   from x in Instance;
BASE 8002 ()
SLIDER 8002 ()

As is apparent, both base and slider instances have the root instance as their
parent and have no children (synonymous with no sub-components). This im­
plies that the assembly tree in this case just spans a single level with an assembly
consisting of two components. An assembly part file usually does not contain
detailed information about its components as it does not contain a copy of
component geometry, but merely points to component part files. So, to ob­
tain detailed information about component geometry, it is necessary to open
up component part files.

Example 17 Determine the names of the component part files of the components
of the assembly.

CQL> select x->instanceName, x->collectionName
   from x in Instance;
BASE /home/research/lex/ug/base.prt
SLIDER /home/research/lex/ug/slider.prt

Example 18 Open the part "slider.prt" and display its volume, surface area, mate­
rial density and extents in 3D space.
CQL> open "/home/research/lex/ug/slider.prt";
CQL> select Solid(volume, density, surfArea, extent);
8.8827908 0.2829 39.379634 (2.5625,3.25,2.8128352)

Once a component part file has been opened, information about its solids is available and all the queries used above may be applied.

**Example 19** Display the parameter names and values of all counter-bored holes in the slider.

CQL> select x, x(paramNames, paramValues)
    from x in Solid.getFeatures
    where x->featureType == "CBORE_HOLE";
423 (CBORE_DIAMETER,HOLE_DIAMETER,CBORE_DEPTH,HOLE_DEPTH, TIP_ANGLE,THRU_FLAG) (0.21875,0.1495,0.375,1.0625,0,0)
402 (CBORE_DIAMETER,HOLE_DIAMETER,CBORE_DEPTH,HOLE_DEPTH, TIP_ANGLE,THRU_FLAG) (0.21875,0.1495,0.375,1.0625,0,0)
Chapter 4

CQL: Implementation

4.1 Introduction

Having explored the syntax of CQL and the use of queries to access CAD part file information, it is time to look into the inner workings of CQL and understand how a query actually retrieves data. CQL has currently been implemented on the SGI-IRIX and HP-UX platforms using the C language, for the Unigraphics CAD system. The architecture of CQL and its interaction with the CAD system and external applications is best explained by figure 4.1. CQL is designed for use through two interfaces, namely an interactive interface, where the user interactively types in CQL queries and through the TNR programming language. Irrespective of the interface used, the steps involved in the processing of a query are the same. If the query is grammatically correct, then CQL extracts the required data from the CAD system via its Application Program Interface (API). Currently, this corresponds to the Unigraphics API. The API is a set of C and FORTRAN routines available to external applications, to access CAD system data. As the uni-directional arrows indicate, information flow occurs only out of the CAD system as CQL does not
Figure 4.1: Architecture of CQL
provide ways to write to CAD part files. This data is then passed on as a character string to TNR or displayed in the form of rows and columns on the user screen, as the case may be.

When a query is issued, it must first be checked for conformance to CQL syntax and grammar rules. This is accomplished by a parser generated using the software tools ‘lex’ and ‘yacc’. A lexical analyser, generated with lex, breaks up the query into tokens (character strings) pre-defined in CQL. These tokens correspond to CQL keywords, legal variable names and operators. These tokens are then passed on to the parser, generated by ‘yacc’, which checks the stream of tokens for conformance to pre-defined grammar rules. Failure to comply with the grammar is reported as a syntax error. In general, all errors occurring at this stage are termed parse errors. During this process, the parser also constructs a tree of ‘nodes’ for execution.

After successful parsing, bindings must be made in the node tree to associate variable references to their corresponding variables. Some syntax errors, such as using undefined variables, are detected at this time and reported as runtime errors. After binding, portions of the node tree are rearranged to eliminate unnecessary looping during query execution, if the optimized-where-clause is used. Finally, the node tree is executed by executing the leading node, after which query results are accumulated and returned. Once this cycle is complete, the node tree, along with other temporary data structures created, is destroyed in preparation for the next query. The following sections illustrate this process in detail and provide a description of the major procedures instrumental.
4.2 Data Structures

The result of parsing a query is a complex data structure, that in essence is a software machine, whose execution yields the query results. Every node in this data structure possesses some information necessary for executing the query and knows how to execute itself. These nodes are termed ‘q.node’s.

4.2.1 The q.node

The structure of a q.node is shown in figure 4.2. Every q.node has a type and subtype that determine its function in the node tree. Depending on function, q.nodes may be classified into the following node types:

**Operator Nodes** These nodes serve as control centers for query execution and accumulation of query results.

**Class Nodes** These nodes are responsible for extracting all instances of the specified entity from the CAD part file.

**Class Reference Nodes** These nodes contain class information to be used by other nodes.

**Method/Multi-Method Nodes** These two node types result in the execution of single and multi-valued methods respectively.

**System Nodes** These nodes are used in the execution of system queries.

Figure 4.3 shows the links that exist between q.nodes in the node tree. Every q.node has a link to the next q.node at its level (next) and the q.node that it is immediately below it in the tree (contents). The ‘last’ pointer links a q.node to the last q.node in the chain at its level. This is used for efficiency while appending nodes
typedef struct q_node {
    struct q_node* delete;
    struct q_node* next;
    struct q_node* contents;
    struct q_node* last; /* Pointer to last q_node in q_node chain */
    struct q_node* ref;  /* Pointer to node that this node refers to */
    int type;
    int subtype;
    v_node* v;
    status_type status;  /* Status of node */
} q_node;

Figure 4.2: Structure of a q_node

Figure 4.3: A q_node Chain
typedef struct v_node {
  struct v_node* next;
  v_node_type type;
  union {
    int b;
    long i;
    double d;
    char *s;
    struct v_node *c[3]; /* c[0] points to the first element,
                           c[1] points to the last element.
                           c[2] is used as an iterator. */
  } val;
} v_node;

Figure 4.4: Structure of a v_node

to the chain. The ‘delete’ pointer links a q_node to the one created immediately before it. By means of the ‘delete’ pointer, all q_nodes are linked together so that they may be deleted at the end of the query. Some q_nodes (var_ref, select_var_ref, suppress_var_ref) have a ‘ref’ pointer that links them to the variable that they reference. During the execution of a query, certain q_nodes exist in several states, as reflected by the ‘status’ field. To store intermediate information used to process a query, a q_node uses a structure called the ‘v_node’.

4.2.2 The v_node

v_nodes are used to hold data. They do not play any part in controlling query execution. The structure of a v_node is shown in figure 4.4.

v_nodes may be used to store integer, boolean, real, string or aggregate data. All integers are internally represented as long integers and reals are represented as double-precision floating point numbers. The ‘type’ of a v_node indicates the type
val.e\[I\]
val.e\[O\]
next next next

Figure 4.5: Example of an Aggregate v_node of Integers

of data that it contains. Thus a v_node may be of a boolean, integer, double, string, dollar, aggregate, row or column type. The dollar type is used to indicate that the data held is undefined. Query results are typically held in the forms of rows and columns using the row and column type v_nodes. The value of the v_node is held in the variable ‘val’. v_nodes have a link to the next v_node in the chain to facilitate the building of aggregates. Aggregate, row and column v_nodes have additional links to the chain of v_nodes that they hold to facilitate their traversal, as shown in figure 4.5.

4.3 Query Processing

The successful completion of a query involves the parsing, binding, optimizing, execution and clean-up phases. The structure of the node tree built-up in each case depends on the kind of syntactical constructs used. However, in general, the node tree can be visualized as shown in figure 4.6. The node tree can be conceptually represented as the from clause, followed by the where clause and the select clause. As discussed in the previous chapter, this results in the construction of a working data set by the from clause, filtration of this data by the where clause and selection of the required information by the select clause. To provide an overview of all the processes
involved from the input of the query to the generation of results, the following query will be examined.

```cql
CQL> select y, y->featureName from x in Solid,
    y in x->getFeatures
    where x->volume > 4.0;
```

### 4.3.1 Parsing

The query is first checked for grammar by the lexical analyser and parser. If there are no syntax errors, a node tree containing information about the query is generated. The node tree created as a result of parsing is shown in figure 4.7. The sections of the tree corresponding to the from, where and select clauses have been shown enclosed by discontinuous lines. All oval-shaped boxes represent q.nodes and all square boxes represent v.nodes. With respect to any given q.node, the arrow directed towards the right-hand side corresponds to the 'next' pointer and the arrow directed downward/leftward corresponds to the 'contents' pointer in the q.node data structure (figure 4.2). The q.node type is indicated inside the oval boxes along with the name of the method in case of single and multi-valued method nodes.
4.3.2 Binding

At this stage, the node tree is not yet ready for execution. All nodes that refer to a variable declaration (var.dcl) need to be bound to the appropriate var.dcl. In this case, the variable reference (var_ref) and select variable reference (select.var_ref) nodes need to be bound. This consists of setting the 'ref' pointers of these nodes to point to the appropriate var.dcl. During this process, syntax errors such as referring to undeclared variables or multiple declarations of the same variable are detected, producing runtime errors. The node tree after the binding process is shown in figure 4.8 with 'ref' indicating the new links in place. Dotted arrows represent the presence of other nodes not shown for clarity.

4.3.3 Optimization

The node tree, if executed at this stage, will produce perfectly good query results. However, execution may involve unnecessary steps, depending on the structure of
the where clause, thus increasing query execution time. This may be avoided by rearranging the data structure for optimum execution. To illustrate this problem better, consider the simple query shown below.

\[
\text{CQL} > \text{select } x \text{ from } x \text{ in } (1,2,3), y \text{ in } (4,5,6) \\
\text{where } x == 3 \text{ and } y == 5;
\]

3

The node tree after parsing and binding would look as in figure 4.9. As will be explained in detail, during the process of execution, each \texttt{var.dcl} node iterates over all possible values of its variable. Thus in this case, the variable \(x\) iterates over the values 1, 2 and 3. For each of these values of \(x\), the variable \(y\) iterates over its possible values (4,5 and 6). For each value of \(x\) and \(y\), the where clause is executed. If the conditions specified are satisfied, the selection occurs. Thus with the present node tree, the total number of executions of legs A and B amounts to 9, out of which one
Figure 4.9: Node Tree without Optimization for Simple Query

selection occurs \((x==3 \text{ and } y==5)\). Given the nature of the conditions specified, it is possible to avoid executions of leg B for unacceptable values of variable \(x\). This is achieved by re-arranging the node tree as shown in figure 4.10. As a result, the number of executions of legs A and B in total is reduced to 3. However, it may not always be necessary to perform an optimization.

In performing an optimization, an unfortunate side-effect is that the user has no control over the sequence in which the specified conditions are executed. Consider the following query:

CQL> select x from x in Group.getObjects, y in Face
   where x==y and x->faceType == 16;
The above query attempts to select all cylindrical faces from all Groups. Although this may be done in different ways, the given query is a valid way of effecting the same. Due to the optimization process, this query results in the condition ‘x→faceType == 16’ being checked before the condition ‘x==y’. As a result, if x is an entity other than a Face, then the execution of the faceType method on x causes the query to be halted with an error message, as the method is invalid for the given Object. In this case, it is essential to ensure that x is a Face before attempting to check if it is cylindrical. In these kinds of circumstances, the explicit-where clause is used. As a result, optimization does not occur and the order of execution of conditions is preserved.

Coming back to our example query under consideration, optimization results in the node tree shown in figure 4.11. This tree executes faster as the var.dcl node for variable y is executed only if the volume of the solid is greater than 4.0 cubic inches and not otherwise.
4.3.4 Execution

Every q.node has a specific responsibility during execution. The location of the leading node in the tree (in this case, the var.dcl node with x) is always stored in the global variable 'cql.Begin.query'. Execution of the tree is started by executing the node pointed to by this variable. This starts off a recursive process in which every node executes its contents and the node following it. The responsibilities of the major nodes involved are summarized below:

The var.dcl node The var.dcl node initiates the extraction of data from the CAD part file. It first executes the node below it. The nodes below it retrieve data and pass up the retrieved value(s) to the var.dcl which holds the value in its v.node. If the execution of the lower nodes yields a set of values, these values are passed up one at a time. For each such value passed up, the var.dcl executes the next q.node. This causes a chain of executions to cascade through the tree.
from left to right until control finally returns to the var.dcl, along with a signal
to continue or stop. On receipt of a continue signal, it executes its contents
again to obtain the next value in the set of values and executes the next node.
This process is repeated until the var.dcl exhausts all values in the set or until
a stop signal is received, at which time it passes execution control back to the
node that executed it. In case of the first var.dcl, the query is complete at this
stage. In all, this generates the effect of the declared variable iterating over a
set of values.

**The class node** The class node is responsible for collecting all instances of the speci-
fied entity from the CAD part file. The subtype of the class node determines
the type of entity that is to be collected. In the example shown, the class node
has a subtype corresponding to the Solid entity. The class node uses the ap-
propriate functions to interface with the CAD system and extract all instances
of the specified class. As a result, a set of object IDs is obtained. The class
node then executes the elements node below it and passes on this set of IDs to
it. Once the set of IDs is passed below, the role of the class node is reduced to
that of an intermediary node that, when executed, executes the node below and
passes the value obtained up the tree. This continues until all values resulting
from the set passed below are exhausted. This completes the life-cycle of the
class node.

**The elements node** The elements node (represented by ‘[ ]’) is found attached
below nodes that generate a set of data. Its responsibility is to accept a set
of values and iterate through them. For each such value, it executes the node
below it with the value and passes back the result obtained up the tree. If there
are no nodes below it, it simply passes the values in the set, one at a time, up
the tree. When all the values in the set have been exhausted, it signals this to
the upper portions of the tree.

**The where nodes** There are three kinds of where nodes discussed here: the
optimized-where node, the where node and the explicit-where node. A node
tree may either contain a number of optimized-where nodes and a where node
or just a single explicit-where node. The optimized-where node functions much
like a gatekeeper. When executed, it executes the node below it. If the result
of the executions of the nodes below yields ‘TRUE’ then it executes the next
node in the tree, else it re-initializes the remainder of the node tree to the right
and passes back control immediately to the previous node. The process and
need for re-initialization is explained in detail in case of the for condition.

The where node has a much different function. It is the last node before the
select-clause section of the tree begins. It is responsible for creating a new
row in the query results data structure and passing it on to the select section.
Selected data is appended under this row.

The explicit-where node combines the functions of the optimized-where and
the where nodes. The explicit-where node, unlike the where-node has condition
nodes under it that need to be executed. If the execution of these nodes returns
‘TRUE’, then a row is added to the results data structure and the next node is
executed. The failure of the condition results in control being transferred back
to the previous node. The node structure of an explicit-where node is identical
to that of a where node that has not been optimized. The example shown in
figure 4.9 depicts a node tree before optimization. The explicit-where clause,
if used, would result in such a node tree at the end of parsing.
The **conditional operator node** Conditional operator nodes may be of several kinds, the kind of condition being identified by the subtype of the node. There exist separate condition nodes for all the atomic and compound conditions discussed in the previous chapter. The contents node and the next node for a condition node correspond to the left and right hand sides of the imposed condition respectively. A condition node executes its contents and next nodes and checks for the condition using the values obtained. It returns the result of this check (‘TRUE’ or ‘FALSE’) to the node above it. This is the mode of execution of all condition nodes except for the for-condition. The node structures of the different condition nodes are shown in figure 4.12. The structures of all the negative operands(!=, is_not_a, not_in etc.) are similar to that of their positive counterparts.

The for-condition merits separate consideration. The structure of a for-condition is shown in figure 4.13. Both variants of the for condition, the for-all and the for-any conditions, have similar node structures. The nodes enclosed by discontinuous boxes are compact representations of larger structures. Considering the case of the for-all condition, the for-all node executes the var.dcl node and waits for a return value. The var.dcl executes as explained earlier and each value generated is checked against the condition by the for.all.where node in repetitive fashion. If any of the values checked fails the condition, a stop signal is sent to the var.dcl. This signal is passed on to the for.all node which then returns ‘FALSE’ to the node above it. If all values generated by the var.dcl satisfy the condition and it runs out of values, the for_all node returns ‘TRUE’ to the node above. The functioning of the for-any condition, though slightly different, uses similar principles.
Figure 4.12: Conditional Operator Nodes
An important difference exists in the nature of execution of the for condition. As mentioned previously, q_nodes may assume several states during the course of execution of a query. This is reflected in the value of the ‘status’ member in the q.node structure. At the beginning of a query, all q_nodes are in the un-initialized state. In general, whenever the primary var_dcl nodes for the query take on new values of the variables declared, the remainder of the node tree to the right must be in the un-initialized state. This occurs as a normal consequence of query execution as long as all the var_dcl nodes in the tree are allowed to iterate over all possible values. However, in case of the for condition, this is not the norm. Under such circumstances, it is necessary to ‘re-initialize’ the remainder of the node tree at the end of the for condition. This is accomplished by the ‘for_all’ or ‘for_any’ node.

The var_ref node The execution of the var_ref node is similar to that of the var_dcl node in many ways. It, however, does not have a next node and only passes values up and down the tree. A var_ref node executes the node below it with the value of the variable that it refers to. If this execution results in the generation of a set of values, these values are passed up the tree, one at a time, on each
subsequent execution. If a single value is generated, this is passed up and the life-cycle of the var.ref is complete. If there are no nodes below a var.ref, the value it refers to is passed up the tree.

**The method nodes** The method and multi-method nodes are discussed here. In this context, the word ‘method’ is used to refer to both types of nodes, unless stated otherwise. Both these types of nodes use their subtype to identify the kind of method that they represent. Their function is to execute the specified method on the object ID passed down to them. To do this, they call the UG API functions. The node below is then executed with the output of the method and the result is passed up the tree. In case of multi-valued methods, the aggregate obtained from their execution is passed down to the elements node below. Subsequent executions of the multi-method node do not execute the method itself but only result in values generated in lower portions of the tree to be passed up.

**The value node** These nodes are used to hold constants used in the query in their v.node. On execution, they simply pass the value held, up the tree. In case of aggregate constants, an elements node under the value node breaks up the aggregate and sends values up the tree.

**The select.var_ref node** The select.var_ref node is instrumental in selecting the required portions of the filtered data and constructing the resulting data structure. It executes the node below with the value of the variable that it references. The result of execution of any methods on this variable from lower portions of the tree are passed up one value at a time. The values passed to it from below are appended to the results data structure that represents the final output of
the query. After it has appended all passed up values, it executes its next node if present. The results data structure, for a query that results in rows of integers being selected is shown in figure 4.14. Each v_node of type v_column.t shown represents a row in the resultant query.

In the light of the above functions, the execution of the tree shown in figure 4.11 can be explained easily. Execution begins by execution of the node pointed to by the variable `.cql_Begin_query`. Thus the leading var_dcl is executed. The class node below collects object ID’s of all Solids from the part file and hands them to the elements node below which passes these up one by one. For each ID passed up, the optimized_where is executed. The volume method is executed for the current value of x (current object ID) and if it is greater than 4.0, a ‘TRUE’ is returned to the optimized_where which then executes the next var_dcl. This results in the the execution of the getFeatures method for the value of x pointed to by the var_ref (the current value of x), generating a set of feature IDs which are handed up one by one. Thus, for each Solid ID (x value), y cycles through all the feature IDs. For each (x,y) pair, the where is executed which creates a row in the result data structure to append results to. Following this, the select_var_refs execute adding the result of
their methods to the data structure. This continues until all values of x and y have been exhausted.

4.3.5 Cleanup

After the query has completed execution and the results have been displayed/passed on to TNR, the node tree and the results data structure must be destroyed in preparation for the next query. This is done by deleting the nodes pointed to by the '_cql_Nodes' and '_cql_Query_result' global variables when the tree and the results are destroyed by a recursive delete process.

4.3.6 The Suppress Mechanism

The suppress query undergoes the same steps as the other queries during processing. However, there are some important differences in the execution process that are explained here with an example. Consider the following query that suppresses all the features in all the Solids in the part file:

CQL> suppress x from x in Solid.getFeatures;

The resulting node tree after parsing, binding and optimization is shown in figure 4.15. Note the presence of the suppress and the suppress_var_ref nodes. The suppress node, unlike in the select query, is at the head of the node tree. At the start of execution, the suppress node simply executes the next node. This begins the iteration of all the declared variables. The suppress_var_ref nodes function exactly like the select_var_ref nodes except for the fact that instead of appending the values passed up to the results data structure, these are added to a global list of features to be
Figure 4.15: Suppress Node Tree

suppressed. When control returns to the suppress node, all the features in this list of feature IDs is suppressed and the list is destroyed. The suppress node then fills the results data structure with the IDs of all features currently in the suppressed state in the CAD system. The unsuppress query also executes by the same principle.

4.3.7 System Queries

System queries have very simple tree structures. All that a system node needs to do is invoke the appropriate API function and condition the data that is sent and retrieved from the function. Since all system queries involve CAD system specific operations, the system query is first routed through functions in the file ‘cql_cad.c’ that call the appropriate CAD specific functions. The node structures for all the system queries are shown in figure 4.16.

open The open node is the most important of all system nodes. Its node structure consists of the open node with a v_node containing the name of the file to be
opened attached to it. A check is first made to verify if a part specified is already open. If so, it is just made the display part, otherwise it is opened. In all cases, the notion of the active part is updated.

**close** The node structure just consists of the close node. The API function responsible for closing all open parts is called and the absence of any active part is marked. Upon closure of a file, no attempt is made to write back the file to disk.

**search** This query structure consists of the search node containing an aggregate v.node with the names of all the paths to be searched for component part files in case of assemblies. The paths specified, along with their sub-directories, are the only paths in which component files are looked for subsequently in the CQL session. These paths are only used for the current CQL session and a fresh session initiates default behaviour. Default behaviour constitutes searching for components in the same directory as the assembly file.
4.4 Organization of CQL Code

The sections to follow provide an overview of the C code used in the implementation of CQL. During the implementation of CQL, care has been taken to isolate CAD system specific modules from the rest of CQL. As a result, the present organization of code allows new modules to be added to the system with minimal changes to existing code. The following files comprise CQL:

**cql.main.c** This contains the function ‘main’ that starts off the execution of CQL and the signal handlers to handle operating system traps. This is only used by the interactive form of CQL.

**cql.1** This contains the token definitions for the lexical analyser.

**cql.y** This is the input file for the parser generator containing the grammar for CQL and the mode of construction of the node tree.

**cql.model.c** This file contains the high level routines that control parsing, binding, execution and output of results.

**cql.parse.c** This contains all the routines to manipulate q.nodes and v.nodes.

**cql.exec.c** The routines here manage the execution of any type of q.node. Calls to CAD system specific functions in case of method, class, class_ref and multi-method nodes are routed to the file ‘cql.cad.c’ from here.

**cql.cad.c** The routines here serve as an intermediate layer to handle function calls to CAD system specific functions. Depending on the CAD system in use, the appropriate functions are called.
cql.ug.exec.c These routines control the execution mechanism for class, class.reference, method and multi-method nodes for UG along with other functions which aid this process.

cql.print.c These functions aid in verifying the structure of the node tree by printing out the tree structure and are used only during debugging.

cql.ug.udo.c This file contains code for defining user defined objects for the Unigraphics CAD system.

cql.generic_CAD_exec.c This file is a template that provides the skeletal structure for functions that need to be implemented to extend CQL to other CAD systems.

Due to the large number of global variables in use and the need to avoid name conflicts with global variables in TNR, a uniform naming convention has been adopted. All global variables begin with the prefix '_cql_'. Those specific to the routines that make Unigraphics API function calls begin with 'cql.ug_'. The first character following this prefix is capitalized. Similarly, all CQL functions begin with the prefix 'cql_' to avoid name conflicts with functions in TNR and the CAD system libraries.

4.5 Important Global Variables

This section lists the important global variables used in the CQL implementation and describes their use. All global variables may be found defined either in the file cql.model.c, in which case they pertain to CQL as a whole, or in cql.ug.exec.c, when they are used for UG specific actions. All variables that have the prefix '_cql_ug_' are found in the file 'cql.ug.exec.c'. Others, with the exception of UFX.UDO.face.loop_class which occurs in 'cql.ug.udo.c', are found in 'cql.model.c'. 
.cql_Error_detected This is an integer used to indicate the occurrence of a parse
error.

.cql_Query_done This is an integer used to indicate that the query has been com-
pleted.

.cql_Query_type This is an integer used to indicate the type of query executed.
Possible values may be found defined in the file cql.h.

.cql_State This is an integer used to identify the most recent phase completed in
the execution of a query.

.cql_CAD_system This integer indicates the type of CAD system that is being
used.

.cql_Jmp_env This is a very important variable of type jmp_buf used to store the
program stack for error handling.

.cql_Statement This is the character array used to store the input CQL query.

.cql_Nodes This is a pointer that points to the last q_node created and is used to
delete the node tree.

.cql_Begin_query This is a pointer that points to the leading q_node in the node
tree.

.cql_Query_result This is a pointer that points to the query result data structure.

.cql_Suppress_list This is a pointer that points to the list of features IDs to be
suppressed/unsuppressed.

.cql.pg.Part.tag This is an integer used to keep track of the active part open for
querying.
.cql.ug.Status This integer indicates if a license has been obtained for the Unigraphics CAD system.

.cql.ug.Error.str This is a string used to hold error messages generated by Unigraphics.

.cql.ug.Check.method This is a 2D array that maps all entities in CQL to the methods that may be used on them.

.cql.ug.Feature.map This is a 1D array that maps a feature type to its name string.

.cql.ug.Parameter.map This is a 1D array that maps a feature parameter type to the name string of the parameter.

.cql.ug.Argument.map This is a 2D array that defines all the feature parameter types for a given feature type.

UFX_UDO_face_loop_class This is the Unigraphics recognized class name of the face-loop entity.

4.6 Important Routines

This section provides an overview of the major functions used in the implementation. A conscious effort has been made during implementation to isolate CAD system specific functions in separate files. As a result, in all cases where a CAD system API function needs to be invoked, the function call from the generic CQL interface functions (cql.exec.c) is routed to the file cql.cad.c. The functions here route the call to the functions for the appropriate CAD system based on the CAD system currently being used.
4.6.1 Program Control Functions

These functions perform high level tasks and set up the mechanism for input of queries, call the appropriate functions to parse, bind, optimize, execute and print query results.

\texttt{cql.start} This routine leads to the initialization of all the CAD systems supported by initializing global variables and is called when CQL is started.

\texttt{cql.shutdown} This routine is called when a CQL session is terminated by the 'exit' command. All CAD system licenses that have been obtained are returned as a result.

\texttt{cql.parse} This routine calls the parser with the input query. On a syntax error, the node tree is destroyed and an error message is returned.

\texttt{cql.process} This routine controls all phases of processing after a query is parsed. Importantly, a copy of the program stack is stored at this stage should a rollback be required on error. This routine calls the functions responsible for binding, optimizing and executing the query and destroys the node tree.

\texttt{cql.print.result} This function leads to the printing of the result data structure and its subsequent deletion.

4.6.2 Node Manipulation Functions

These routines help in the manipulation (creation, appending, insertion, initialization, assignment and deletion of nodes.

\texttt{cql.add.q.node} A q.node is created, given the contents, next, node type, subtype and v.node.
_cql_append_q_next This appends a q.node to the next of the last node encountered by traversing the next pointer in its node chain.

_cql_append_q_contents This appends a q.node to the contents of the last node encountered by traversing the contents pointer.

_cql_insert_q_next This inserts a q.node immediately after the specified q.node and makes the appropriate links.

_cql_delete_q_nodes This initiates a recursive delete process by which the entire node tree is destroyed. When called on a q.node, it deletes its v.node and deletes the node connected to its delete pointer.

_cql_dup_q_node Given a q.node chain, it produces a duplicate chain by recursively duplicating the contents, next and v.nodes of all nodes in the chain.

_cql_reinit_q_nodes Some q.nodes change state during the execution of the node tree. During some queries, they require to be re-initialised (restored to start state) by cleaning up all changes to them because of the query. This is accomplished for the specified q.node and all others following it, by this routine.

_cql_add_v_node This routine creates a v.node given the v.node type and a value to initialize the v.node with. When an aggregate v.node is created, it does not have any members.

_cql_add_v_node_integer This is one function in a family of functions that create a v.node and initialize it with an integer, double or string value.

_cql_append_to_v_aggr This is a very useful routine that enables appending a v.node to a column, row or aggregate type v.node.
_cql_delete_v_nodes_ This is a routine that deletes a v_node and all other v_nodes in its chain.

_cql_delete_v_node_at_itr_from_aggr_ This routine takes an aggregate v_node and deletes the v_node that its iterator points to. The iterator is then set to point to the first v_node in the aggregate’s contents.

_cql_assign_v_node_ This routine assigns one v_node to another. v_nodes of different types may be assigned to each other. This results in overwriting of one v_node by the other. Assignment of aggregates to other aggregates as well as to non-aggregates is supported.

_cql_assign_v_integer_ This is one in a family of v_nodes that set a v_node’s value to that specified and make the required changes to its type. Functions exist to assign integers, doubles, strings and boolean values to existin v_nodes.

_cql_init_v_node_ This sets the v_node’s type to the dollar type.

_cql_dup_v_node_ This routine facilitates the duplication of a v_node chain or individual v_node like in the case of q_nodes.

_cql_clean_v_node_ This routine frees up all allocated memory within a v_node and sets its type to the dollar type.

_cql_print_v_node_ This routine prints a v_node structure to standard output after converting it into a character string with the appropriate delimiters between values.
4.6.3 Node Execution Functions

These functions facilitate the execution of q_nodes and make recursive calls to each other to execute adjoining nodes in the tree.

_cql_bind_q_node This function is responsible for making the bindings in the node tree. All nodes which refer to a variable need to be bound to the appropriate variable declaration. The bind process is initiated when this function is called with the leading node in the tree. When nodes that reference variables (var_ref, select_var_ref, suppress_var_ref, unsuppress_var_ref and their implied counterparts) are found, the corresponding var_dcl nodes are found by back-tracking along the node tree and a link is established. Undeclared and doubly-declared variables lead to runtime errors.

_cql_optimize This routine performs the optimization of the node tree by the use of the routines 'cql_relocate' and 'cql_find_max_dcl'. The where node is located and its contents, if present, are relocated. If the section of the tree to be relocated has an 'and' node at the top, the individual conditions on either side of the 'and' node may be relocated independently. For the tree to execute correctly, the relocated section must be placed to the right of all the var_dcl nodes referenced by the section. For optimization, the section must be placed as close to the left end of the tree as possible. Thus the 'cql_find_max_dcl' routine locates the rightmost of the referenced var_dcl, next to which the section is placed. This is performed for all relocated sections.

_cql_exec_q_node This is the generic interface for the execution of any node. Depending on the type of node, the appropriate function for the execution of the specific node is called from this routine.
_cql_exec_q_oper_ This function executes operator type q_nodes. Depending on the subtype of the operator node different actions are performed. In general, for every node a unique set of actions is performed and its adjacent nodes are executed. This set of actions is the same for all CAD systems.

_cql_exec_q_class_ This and the functions to execute the other kinds of q_nodes, all perform CAD system dependent actions. Hence, they are routed through another layer of functions that call the functions for the specific CAD system.

_cql_exit_cad_systems_ This routine calls the functions for each CAD system to ensure that licenses are returned.

_cql_init_cad_systems_ This routine calls the functions for each CAD system to initialize the appropriate global variables at the start of a CQL session.

_cql_make_suppress_list_ This is the generic interface to all CAD systems to build a list of feature IDs to be suppressed/unsuppressed. This routine calls the specific function for the CAD system in use.

_cql_perform_operator_ This function implements the atomic conditional operators. Given the nodes to be compared and the comparison operator to use, it checks for the applicability of the operator on the specified nodes and returns the boolean result of the comparison.

_cql_perform_suppress_ This is the generic interface to suppress a list of features. The function for the appropriate CAD system is called.

_cql_perform_unsuppress_ This generic function to unsuppress also calls the function for the specific CAD system.
_cql_perform_isa_ This function tests if the given q_node contains the ID of an object of the specified type. This is a generic interface.

_cql_exec_q_class_ug_ This function calls the appropriate Unigraphics API functions to collect all entity IDs of the specified class from the part file.

_cql_exec_q_class_ref_ug_ This function creates a data structure containing the object types and subtypes of all objects that may be considered to be descendants of the object type specified.

_cql_exec_q_method_ug_ This function executes single valued methods for the Unigraphics CAD system. Checks are first made to ensure that the object ID passed in is valid and that the specified method can be executed on the given object. The required API functions are called to execute the method and any chained methods. Chained methods can only be executed if object IDs comprise the result. A runtime error is generated otherwise.

_cql_exec_q_multi_method_ug_ This function executes multi-valued methods for Unigraphics.

_cql_exec_q_system_ug_ This function executes system nodes like ‘open’, ‘close’ and ‘setSearchDirs’ by calling the appropriate API functions.

_cql_exit_cad_system_ug_ This function is responsible for returning the Unigraphics license.

_cql_init_cad_system_ug_ This function initializes the ‘_cql_ug_Error_str’, ‘_cql_ug_Status’, ‘_cql_ug_Part_tag’ global variables and the global feature matrix maps at the start of CQL.
4.6.4 UG API Support Functions

_.cql.get_cql_type_ Given an object ID, this function returns an integer representing the entity type as defined in CQL. These definitions may be found in the file ‘cql.defs.h’.

_.cql.get_ug_type_ This function serves to identify the the entity type of a specified object ID. Due to inheritance relationships, a given object may be associated with different object types. Hence, instead of returning a single object type, this function returns a v.node structure containing the the Unigraphics defined types and subtypes of all valid descendants of the specified entity.

_.cql.get_objects_by_type_ Given a data structure containing the Unigraphics defined types and subtypes of the objects desired, this function retrieves instance IDs of all specified objects from the part file.

_.cql.get_objects_from_group_ This function extracts objects from a specified reference set either by the name of the object desired or by its type. Depending on whether the argument supplied corresponds to a data structure containing the UG defined types and subtypes of the objects desired or an object name, this function extracts objects from the specified reference set.

_.cql.get_objects_from_ug_group_ This function acts exactly like the above function, but extracts objects from a Group whose instance ID is supplied.

_.cql.get_feat_type_ Given the object ID of a feature object, this function returns a CQL defined integer representing its feature type.

_.cql.get_feat_params_ This function gets parameter related information for a specified feature. Depending on the arguments supplied, it can return a list of name
strings for all the feature parameters, a list of the values of all parameters for
the feature in a pre-defined order or the value of a particular feature parameter.

`.cql.perform.suppress.ug` This function uses the UG API function to suppress all
features present on the global feature list.

`.cql.perform.unsuppress.ug` If a list of feature IDs is given, it unsuppresses all
the features on the list, else unsuppresses all suppressed features.

`.cql.perform.isa.ug` This function checks if the object ID passed in belongs to an
object of the type specified. It does so by comparing the object type of the ID
specified against a map of all legal object types for a specific class.

**UFX.UDO.create** This function uses the concept of user defined objects to define
the face loop object class having class name ‘UFX.UDO.face.loop.class’.

**ufsta** At the startup of Unigraphics (obtaining of the license), a search is made for a
function of this name in the directory specified by the value of the environment
variable ‘UGII_SITE_DIR’. This function is used to specify any user defined
object class definitions. In this case, it calls the function ‘UFX.UDO.create’ to
define a face loop class.

### 4.6.5 Error Handlers

Errors, as meant here, indicate errors occurring due to bad formulation of queries.
Errors generated during processing of queries may be classified into two types:

**Parse Errors** generated during the parsing stage due to syntax and grammar vi­
olations.

**Runtime Errors** generated during the binding and execution stages.
When an error occurs, the cause of the error has to be echoed to the user as an error message and the system has to be brought back to a stable state for execution of subsequent queries. This involves freeing up dynamic memory allocated to any temporary data structures up to the point of error, as well as restoring the program stack to a stable state. To keep track of dynamic memory allocated to temporary data structures, a set of global variables has been used. Upon encountering an error, the structures pointed to by these variables are destroyed. Just after parsing is complete and before binding begins, the state of the program stack is stored in the variable \_.cql.Jmp.env\. Upon an error, the current program stack is discarded and replaced by the stored stable stack. In addition, in case of external traps from the operating system leading to program termination, all CAD licenses obtained must be returned. Apart from the above, an effort has been made to trap genuine programming errors and errors arising due to abnormal conditions. These errors are indicated by a suitable error message string, where possible, prefixed by the string "System Error". These errors must be reported to the system administrator. The main error handlers that accomplish these are described below.

\_.cql.error\ This is the main error handler that is invoked under all occasions, directly or indirectly. CQL uses the global variable \_.cql.State\ to identify the stage of execution at which an error occurred. If the error resulted during parsing, the function \yyerror\ is called which echoes a syntax error. In case of a runtime error, an error message is printed out if available and the program stack is reset. In both cases, any dynamic memory allocated is released by deletion of data structures pointed by all global variables.

\_.cql.ug.error\ Sometimes, errors are generated by UG API functions due to improper use of CQL methods in queries. These errors are handled by this rou-
tine. This routine constructs an error message string corresponding to the error and invokes the primary error handler `_cql_error`.

**Signal handlers** A CQL session can also terminate due to the receipt of external signals from the operating system. Signal handlers have been defined to return the Unigraphics license obtained in such an eventuality.
Chapter 5

Results and Discussion

5.1 Introduction

The past chapters have traced the development of CQL, from the need to facilitate lossless extraction of CAD model information to its present state, as a specialized query language. It is time to stand back and examine its suitability to its intended role as a cost estimation information extraction tool. This chapter reviews the original design goals of CQL and provides some scenarios to illustrate its utility to design-concurrent cost estimation.

5.2 Design Objectives

Throughout the development of CQL, the focus has been on providing ways and means to extract information relevant to cost estimation. In the presence of multiple strategies to arrive at estimates, this has resulted in the provision of access to all parts of a CAD model, from geometric and feature data to drafting data and user-defined
attributes. Looking back at the tasks undertaken at the beginning of this work, the following observations may be made:

1. A complete and flexible syntax is in place for CQL that allows complex queries to be formulated, yet keeping simple queries as brief as possible;

2. CQL can be currently used via the interactive interface as well as the programming interface from TNR with equal ease;

3. A comprehensive set of methods for the Unigraphics system is in place that may be expanded, if necessary, in the future;

4. A powerful mechanism for computation of feature volumes and user-designated regions is functional;

5. An optimization scheme has been implemented to improve the run-time performance of queries.

These capabilities make CQL a very useful tool for extracting any modeling information stored in the CAD system.

5.3 Examples with Sample Parts

To illustrate some typical scenarios where CQL could be put to use, three sample programs have been formulated. During the course of examination of different cost models and estimating approaches discussed in the literature, some of the product parameters used were:

1. Part weight;

2. Part material;
3. features present, their characteristic parameters and number of such features present;

4. machining volumes;

5. machined lengths and mean diameters for turned parts;

6. specifications for standard components.

The sample programs have been written to demonstrate how such information could be extracted. These can be found in Appendix F.

Example 1 The first part to be examined is a machine casing, shown in figure 5.1.

Part Description The casing consists of several hole-patterns, a rectangular pocket and some cavities to be milled. Unigraphics allows the creation of feature patterns either as circular patterns or as rectangular patterns. The casing contains both circular and rectangular patterns made with simple holes and cylinders. The large cavity in the casing with countoured faces has been constructed by the subtraction of cuboidal blocks from the base solid. Since this is not a standard feature in Unigraphics, the cavity is identified by assigning all the faces enclosing it with a common name. In this case, the faces that enclose the cavity were named “CAVITY1”.

Information Extracted Solid volume, extents, density, composition of all feature patterns, parameters of all features in the part, volumes of cavities to be machined.

Extraction Strategy Before a TNR program can be coded to extract the above information, the various ways in which this information may be organized in the part file must be considered.
Figure 5.1: Suppression Phase 1
- Features, such as holes and pockets, may have been modeled using the CAD system defined features or by boolean operations. In case of the former, feature parameters are stored and may be obtained easily. In case of the latter, alternative measures are necessary.

- Parameters of holes (diameter, depth), for example, if modeled using Simple Hole features are easily extracted. However, they may also be modeled by subtraction of Cylinders. In this case, the hole diameter is the same as that of the Cylinder, but the depth must be computed using the volume of the Cylinder obtained by suppression.

- In cases where a cavity/projection has been obtained by a sequence of boolean operations involving several features, the above procedure may not be applicable. This necessitates some guidance from the user indicating the region of interest.

- Once the region of interest is known, the various possible ways of modeling the given region of the part must be considered to arrive at a suitable algorithm to compute the desired characteristic. In the case of the cavity in the casing, all primitives that may have been used are considered.

Once the various modeling alternatives are understood, the task of the TNR program becomes fairly clear.

**Program Summary** The following is the summary of the actions performed in the TNR program listed in appendix F to obtain the output shown.

- Cycle through all the solids in the part (only one here)
- For each solid found, obtain object IDs of all the patterns
- For each pattern, obtain its members and their parameters
• For the given solid, look for all features of the specified type and extract their parameters. In this case, the features specified were "CYLINDER", "SIMPLE HOLE" and "RECTANGULAR POCKET"

• The volume of the cavity is found by a series of suppressions of the primitives that have been used to construct it. The term 'primitive' is used to refer to one of the five basic features (block, cone, sphere, cylinder, extrusion) that are used as the base feature for any solid. From the faces enclosing the cavity, the features used to model the cavity are identified and from these, the primitives are culled out (in this case, blocks). The volume of the solid is determined (figure 5.1). Now the primitives identified for the cavity are suppressed and the volume of the solid is obtained (figure 5.2). This is the solid volume without the primitives and all of their children (in this case, the concentric simple holes on the floor of the cavity). Now the primitives alone are unsuppressed leaving their children suppressed (figure 5.3). This volume obtained now is that of the casing without the children. The volume of the cavity alone may be computed using these three volumes obtained.

Output The output of the TNR program is shown below. The results may be passed on to other applications from TNR. The hole diameters and depths obtained for all holes in the part (simple holes and cylinders), along with feature pattern information, are inserted into a Part 21 file using EQL. The total number of holes for each diameter-depth pair in the part is then retrieved from this Part 21 file and printed out. Similarly after all pattern
Figure 5.2: Suppression Phase 2
information has been extracted and stored into the Part 21 file, this is retrieved and printed out.

---------------------Solid tag: 587-------------------
Volume: 101.959310   Density: 0.282900   Extents: (8,11.25,3)

CIRCULAR_ARRAY(32) found...
Inserting 5 unit(s) of CYLINDER(30) in database...

CIRCULAR_ARRAY(35) found...
Inserting 5 unit(s) of CYLINDER(33) in database...

RECTANGULAR_ARRAY(41) found...
Inserting 3 unit(s) of SIMPLE HOLE(40) in database...

RECTANGULAR_ARRAY(42) found...
Inserting 1 unit(s) of SIMPLE HOLE(40) in database...

RECTANGULAR_ARRAY(43) found...
Inserting 1 unit(s) of SIMPLE HOLE(40) in database...

RECTANGULAR_ARRAY(44) found...
Inserting 1 unit(s) of CYLINDER(33) in database...

RECTANGULAR_ARRAY(45) found...
Inserting 1 unit(s) of CYLINDER(33) in database...

RECTANGULAR_ARRAY(46) found...
Inserting 1 unit(s) of CYLINDER(33) in database...

RECTANGULAR_ARRAY(47) found...
Inserting 1 unit(s) of CYLINDER(33) in database...

RECTANGULAR_ARRAY(49) found...
Inserting 3 unit(s) of SIMPLE HOLE(48) in database...

RECTANGULAR_ARRAY(51) found...
Inserting 3 unit(s) of SIMPLE HOLE(50) in database...

Getting parameters for all SIMPLE HOLE s...
Inserting 1 unit(s) of SIMPLE HOLE(17) in database...

Inserting 1 unit(s) of SIMPLE HOLE(18) in database...

Inserting 1 unit(s) of SIMPLE HOLE(19) in database...

Inserting 1 unit(s) of SIMPLE HOLE(26) in database...
Inserting 1 unit(s) of SIMPLE HOLE(27) in database...
Inserting 1 unit(s) of SIMPLE HOLE(28) in database...
Inserting 1 unit(s) of SIMPLE HOLE(29) in database...
Inserting 1 unit(s) of SIMPLE HOLE(40) in database...
Inserting 1 unit(s) of SIMPLE HOLE(48) in database...
Inserting 1 unit(s) of SIMPLE HOLE(50) in database...

Getting parameters for all CYLINDER s...
Inserting 1 unit(s) of CYLINDER(30) in database...
Inserting 1 unit(s) of CYLINDER(33) in database...

--------------------------Summary of Holes in Part--------------------------

<table>
<thead>
<tr>
<th>Diameter</th>
<th>Depth</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.250000</td>
<td>0.468750</td>
<td>30</td>
</tr>
<tr>
<td>3.750000</td>
<td>0.125000</td>
<td>1</td>
</tr>
<tr>
<td>3.000000</td>
<td>0.625000</td>
<td>1</td>
</tr>
<tr>
<td>2.000000</td>
<td>0.250000</td>
<td>1</td>
</tr>
<tr>
<td>2.500000</td>
<td>0.781250</td>
<td>2</td>
</tr>
<tr>
<td>0.937500</td>
<td>0.375000</td>
<td>2</td>
</tr>
</tbody>
</table>

--------------------------Summary of Patterns in Part--------------------------

Rectangular Patterns

Pattern Name: RECTANGULAR_ARRAY(41)  # of holes: 4 x 1
Hole Diameter: 0.250000  Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(42)  # of holes: 1 x 2
Hole Diameter: 0.250000  Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(43)  # of holes: 1 x 2
Hole Diameter: 0.250000  Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(44)  # of holes: 1 x 2
Hole Diameter: 0.250000  Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(45)  # of holes: 1 x 2
Hole Diameter: 0.250000  Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(46)  # of holes: 1 x 2
Hole Diameter: 0.250000  Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(47)  # of holes: 1 x 2
Hole Diameter: 0.250000 Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(49)  # of holes: 2 x 2
Hole Diameter: 0.250000 Hole Depth: 0.468750

Pattern Name: RECTANGULAR_ARRAY(51)  # of holes: 2 x 2
Hole Diameter: 0.250000 Hole Depth: 0.468750

-------------------Circular Patterns-------------------

Pattern Name: CIRCULAR_ARRAY(32)  PatternRadius: 1.687500
PatternAngle: 60.000000  # of holes: 6
Hole Diameter: 0.250000 Hole Depth: 0.468750

Pattern Name: CIRCULAR_ARRAY(35)  PatternRadius: 3.687500
PatternAngle: 36.000000  # of holes: 6
Hole Diameter: 0.250000 Hole Depth: 0.468750

Getting parameters for all RECT_POCKET s...
RECTANGULAR_POCKET(20)(LENGTH, CORNER_RADIUS, FLOOR_RADIUS, ANGLE)
(5, 0.375, 0, 0)

--Computing volumes of interacting cavities CAVITY1 and CAVITY2--
CAVITY2 is older....
Suppressing ("BLOCK(9)", "BLOCK(10")
Suppressing ("BLOCK(3")
Solid volume without cavities is 239.275480
Unsuppressing ("BLOCK(3")
Solid volume with old cavity is 229.527260
Unsuppressing ("BLOCK(9)", "BLOCK(10")
Solid volume with old & new cavities is 120.321470
Volume of old cavity is 9.748220
Volume of new cavity is 109.205790

Example 2 This example demonstrates the extraction of the mean diameter and the total machined length for a turned part. These two parameters, along with others, are used to construct a fast cost estimate using a cost model for turned parts devised by Mahmoud and Pugh [16]. The total machined length is defined as the sum total of all lengths (internal and external) machined, as measured on the part after turning. The mean diameter is the average of all diameters (internal and external) on the part.
Part Description The view shown in figure 5.4 is the sectional view of a turned component, split in the middle to reveal the cross-section for tool-path generation. Unfortunately, this operation results in the loss of all feature information, converting the model from a feature-based model to a Boundary Representation model. This is the most adverse circumstance under which CQL may need to function, an environment without any feature information. However, even in such cases, with some user interaction, any information that may be deduced from the faces and edges of the model may be extracted. In this case, the user has named the face constituting the upper half of the cross-section as “SECTION”.

Information Extracted Mean diameter of the part, total machined length.

Extraction Strategy Applying the same considerations as mentioned before, the following modeling situations arise.

- If the turned part has been modeled using features (bosses, cylinders), then those contributing to the part diameter and length need to be identified and used for computations.
- If boolean operations have been used or if a feature-based model is unavailable, then alternative measures are needed. In this case, this consists of exposing the cross section of the part and naming the cross-sectional face.

The mean diameter is purely related to part geometry. Computation of the total machined length on the other hand, requires knowledge about which faces on the part are machined. This is required to be indicated by the user, possibly by assigning an attribute to the machined faces or by
Figure 5.4: Sectional View of Turned Part
naming them. In this case, faces of all edges constituting the cross section have been machined.

**Program Summary** The TNR program `parker2.tnr` in appendix F accomplishes the following:

- Gets all the edges of the face with the name “SECTION”
- Computes the lengths of all the edges and sums them up to arrive at the total machined length. In this case, all edges of the cross section are machined. In a scenario where this is not so, an attribute assigned to the edge can indicate this.
- Gets all the faces that are bounded by the edges of the cross-section and culls out the faces that are axial and non-planar.
- Extracts the diameters for all these faces and averages them. The diameter of a conical face is taken as its diameter at half its height.

**Output** As a result, the mean diameter and the total machined length are computed as shown.

```
Total machined length = 4.087532
Mean diameter: 0.626500
Shutting down UG...
```

**Example 3** This example illustrates the power of CQL in extracting information that may be present in areas other than the solid model itself, namely drafting information.

**Part Description** The part considered is an assembly of tubes attached to flanges and inter-connected by T-joints, shown in figure 5.5. Drawings
such as this typically contain the ‘PartsList’ entity, which has details about all the components used in the assembly. All drafting information indicated on the drawing is present as ‘Note’ entities. Several such ‘Note’ entities may be logically grouped together to form a ‘Group’ entity. After associating a particular part with an identification number obtained from the PartsList, this number or some similar identifier may be used to search for useful information available as drawing notes.
Information Extracted  A list of all parts comprising the tube assembly with their identification numbers and quantities, material, developed length and specifications for all tubes in the assembly.

Extraction Strategy  Extraction of drafting information is relatively free from the considerations discussed in the previous cases. All that is required is a character pattern that may be matched with the text obtained from Notes to see if a Note is of interest.

Program Summary  The program ‘ge.tnr’ shown in appendix F uses the following logic:

- Selects all the lines from the PartsList entity.
- Uses Unigraphics conventions to translate formatting characters into the right number of blank spaces so that output may appear under the right column headers.
- Looks for a Group entity that contains Note members that contain the substring ‘WALL THK’ indicating that wall thickness information is present.
- Extracts the tube number, wall thickness and material information for the tubes from the identified Group along with an identifier regarding the drawing sections the tube spans.
- This identifier is used to search through other Notes in the drawing for any Notes containing ‘LENGTH’ information for the tube between the given sections. This is the developed length.

Output  A list of all parts is output in tabular form. Tube identification numbers along with material, developed length and specifications are shown in figure 5.6.
---List of Parts---

<table>
<thead>
<tr>
<th>Item</th>
<th>Dwg</th>
<th>Dwg Identification</th>
<th>Nomenclature</th>
<th>CAGE</th>
<th>G01</th>
<th>Standard component</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1898M20G01</td>
<td>ASSY</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1898M20P01</td>
<td>TUBE</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1898M20P02</td>
<td>TUBE</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1898M20P03</td>
<td>TUBE</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1898M20P04</td>
<td>TUBE</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1898M20P05</td>
<td>TUBE</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1898M20P06</td>
<td>TUBE</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1898M20P07</td>
<td>TUBE</td>
<td>1</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9960M65P05</td>
<td>TEE,TUBE</td>
<td>2</td>
<td>NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1882M13P01</td>
<td>FLANGE,MANIFO</td>
<td>3</td>
<td>NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1882M14P01</td>
<td>FLANGE,MANIF</td>
<td>1</td>
<td>NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1882M42G01</td>
<td>MANIF,FUEL SP</td>
<td>1</td>
<td>NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>9375M51G07</td>
<td>CPLG,TUBE WLD</td>
<td>1</td>
<td>NO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---Summary Of Tubes Present---

<table>
<thead>
<tr>
<th>Tube#</th>
<th>Material</th>
<th>Dev.Length Specs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>AMS 5557 TYPEI</td>
<td>39.4376 .7500 OD x .0490 WALL THK</td>
</tr>
<tr>
<td>3</td>
<td>AMS 5557 TYPEI</td>
<td>10.9700 .6250 OD x .0490 WALL THK</td>
</tr>
<tr>
<td>4</td>
<td>AMS 5557 TYPEI</td>
<td>4.6224 .5000 OD x .0350 WALL THK</td>
</tr>
<tr>
<td>6</td>
<td>AMS 5557 TYPEI</td>
<td>6.9555 .6250 OD x .0490 WALL THK</td>
</tr>
<tr>
<td>7</td>
<td>AMS 5557 TYPEI</td>
<td>6.9127 .5000 OD x .0350 WALL THK</td>
</tr>
<tr>
<td>8</td>
<td>AMS 5557 TYPEI</td>
<td>5.7306 .5000 OD x .0350 WALL THK</td>
</tr>
<tr>
<td>5</td>
<td>AMS 5557 TYPEI</td>
<td>5.3797 .5000 OD x .0350 WALL THK</td>
</tr>
</tbody>
</table>

Shutting down UG...

Figure 5.6: Output Information from Tube Assy
5.4 Summary of Examples

After reviewing the examples shown above, the following observations may be made about the capability of CQL to deal with any arbitrary CAD part file:

- CAD models constructed using features represent the ideal case for CQL in terms of the amount of information available and the degree of associativity established by the CAD system.

- CQL is capable of performing satisfactorily with BRep models provided there is some guiding user interaction.

- The volume computation mechanism used is very flexible in terms of the ability to suppress selected features at will, however the heuristics used for volume computation need to be refined to compute the volume of any arbitrary region.

- CQL is extremely useful for extracting drafting information. However, to do so, the user must observe some basic drafting discipline. Connections must be present in the form of identifiers to relate information present in different portions of a drawing to a given part. CQL can exploit connections, but cannot forge them where they are non-existant.

- It has been observed that in many situations, it is possible to write very general TNR programs to extract specified information from arbitrary parts. The TNR program used for example 1, may be applied to any arbitrary part with features to extract feature information. Thus separate TNR modules may be coded for specified kinds of information and easily assembled together depending on the need.
• Due to the limited number of examples that can be explained and the unlimited complexity of parts that can be modeled, it is not possible to explore all aspects of the CAD model here. However, methods have been provided, as in the case of assemblies and sketches, to exploit relationships between CAD model entities.

5.5 Important Contributions

The following are some of the most important contributions of this work to the goal of integrating the CAD system with cost estimation and subsequently other manufacturing functions:

1. The development of the CAD Object Model, which clearly delineates the various areas of a CAD model that are of interest to an external application, irrespective of the specific CAD system used.

2. The identification and implementation of methods necessary to access this information.

3. Provision of methods to access feature parameters.

4. Establishment of a flexible and robust mechanism to compute part volumes.

5.6 Difficult Questions

As with all research, the accomplishments above unearth more questions, some new and some old. To name a few:

1. Is the architecture of CQL satisfactorily CAD system independent?

2. What will be the effect of a uniform agreement of features on CQL?
3. What is the general heuristic to compute the volume of an arbitrarily shaped cavity?

5.7 Conclusions and Recommendations

Despite the looming questions, CQL is expected to offer a new and flexible way to access hitherto unavailable product data. The goal of seamless design-concurrent cost estimation is yet to be demonstrably achieved. CQL has however laid out a framework to deal with low-level data extraction issues. The design process now needs to be understood, to identify the product information available at the various stages of design and how this may be used to arrive at a satisfactory cost estimate. The completion of this first phase towards design-concurrent cost estimation clears the way for other important work, namely:

1. The extension of CQL to encompass other CAD systems and the refinement of the CAD Object model.

2. The provision of a designer-usable interface from the CAD system to move data out to other applications.

3. Integrating a variety of cost estimating strategies into the estimation scheme depending on the nature and amount of information available.

4. Investigating the extension of CQL into the manufacturing modules of the CAD system to exploit tooling and tool-path information.

5. Investigating the use of incorporating methods to populate CAD part files with entities.
Bibliography


turing,” in Advances in Feature Based Manufacturing, pp. 1–11, Elsevier Science

in Advances in Feature Based Manufacturing, pp. 363–396, Elsevier Science B.V,
1994.


[12] S. Vinoski, “CORBA: Integrating diverse applications within distributed hetero-

eration,” in Advances in Feature Based Manufacturing, pp. 107–128, Elsevier

extensible sets of design features,” in Advances in Feature Based Manufacturing,

[15] L. Huang, “Eql and tnr: Two languages for advanced software system integra-

[16] M. A. M. Mahmoud and S. Pugh, “The costing of turned components at the
Appendix A

EXPRESS Schema

The EXPRESS Schema representation of the CAD Object Model is presented here.

SCHEMA objectModel;

ENTITY Collection;

  name:                STRING;
  desc:                STRING;
  attribs:            SET[0:?] OF STRING;
  assyRoot:           OPTIONAL Instance;
  objects:            SET[1:?] OF Object;
  exps:               SET[0:?] OF Expression;

UNIQUE:
  key:    name;

END_ENTITY;

ENTITY Object
  SUPERTYPE OF ONEOF (Solid, Feature, Sketch, Instance,
                      Group, ReferenceSet, BrepElem, Curve, Dimension, Note) ;

  tag:          INTEGER;
  name:         OPTIONAL STRING;
  type:         INTEGER;
  attribs:      SET[0:?] OF STRING;
INVERSE:
    myCollection: SET OF Object FOR Collection;
    myGroups: SET OF Group FOR SET OF Object;
    myReferenceSets: SET OF ReferenceSet FOR SET OF Object;

UNIQUE:
    key: tag;

END_ENTITY;

ENTITY Solid
    SUBTYPE OF Object;

    density: REAL;
    volume: REAL;
    color: INTEGER;
    layer: INTEGER;
    surfaceArea: REAL;
    extent: LIST[0:2] OF REAL;
    features: SET[1:?] OF Feature;
    brepElems: SET[1:?] OF BrepElem;

END_ENTITY;

ENTITY Feature
    SUBTYPE OF Object
    SUPERTYPE OF UDF;

    featureName: STRING;
    featureType: STRING;
    paramNames: ARRAY[0:?] OF STRING;
    paramValues: ARRAY[0:?] OF OPTIONAL REAL;
    childFeatures: SET[0:?] OF Feature;
    parentFeatures: SET[0:?] OF Feature;
    brepElems: SET OF BrepElem;
    exps: SET OF Expression;
    sketches: SET[0:?] OF Sketch;

INVERSE:
    mySolid: Solid FOR SET OF Feature;
    myChildren: childFeatures FOR SET OF Feature;
    myParents: parentFeatures FOR SET OF Feature;

END_ENTITY;

ENTITY UDF
SUBTYPE OF Feature;

END_ENTITY;

ENTITY Sketch
SUBTYPE OF Object;

color: INTEGER;
layer: INTEGER;
sketchElems: SET[0:?] OF Curve;
dims: SET[0:?] OF Dimension;

INVERSE:
  myFeatures: SET OF Feature FOR SET OF Sketch;
  myFace: Face FOR SET OF Sketch;

END_ENTITY;

ENTITY Instance
SUBTYPE OF Object;

instanceName: STRING;
collectionName: STRING;
referenceSetName: STRING;
color: INTEGER;
layer: INTEGER;
parentInstance: OPTIONAL Instance;
childInstances: SET[0:?] OF Instance;

INVERSE:
  amRootOf: Collection FOR OPTIONAL Instance;
  myParent: parentInstance FOR SET OF Instance;
  myChildren: childInstances FOR Instance;

END_ENTITY;

ENTITY Group
SUBTYPE OF Object;

color: INTEGER;
layer: INTEGER;
objects: SET[1:?] OF Object;

END_ENTITY;

ENTITY ReferenceSet
SUBTYPE OF Object;

    color: INTEGER;
layer: INTEGER;
objects: SET[0:?] OF Object;

END_ENTITY;
-------------------------------------------------------------

ENTITY BrepElem
SUBTYPE OF Object
SUPERTYPE OF ONEOF (Face, Edge);

INVERSE:
    mySolid: Solid FOR SET OF BrepElems;
    myFeatures: SET OF Feature FOR SET OF BrepElem;

END_ENTITY;
-------------------------------------------------------------

ENTITY Edge
SUBTYPE OF Object
SUPERTYPE OF ONEOF(LineEdge, ArcEdge, EllipseEdge, SplineEdge);

length: REAL;
edgeType: INTEGER;
color: INTEGER;
layer: INTEGER;

INVERSE:
    myFaces: SET OF Face FOR SET OF Edge;
    myLoops: SET OF FaceLoop FOR SET OF Edge;

END_ENTITY;
-------------------------------------------------------------

ENTITY LineEdge
SUBTYPE OF Edge;

END_ENTITY;
-------------------------------------------------------------

ENTITY ArcEdge
SUBTYPE OF Edge;

center: LIST[0:2] OF REAL;
radius: REAL;
startAngle: REAL;
endAngle: REAL;
ENTITY EllipseEdge
SUBTYPE OF Edge;

center: LIST[0:2] OF REAL;
startAngle: REAL;
endAngle: REAL;
majorAxis: REAL;
minorAxis: REAL;

END_ENTITY;

ENTITY SplineEdge
SUBTYPE OF Edge;

END_ENTITY;

ENTITY Face
SUBTYPE OF Object;

faceType: INTEGER;
area: REAL;
direction: LIST[0:2] OF REAL;
point: LIST[0:2] OF REAL;
extent: LIST[0:2] OF REAL;
color: INTEGER;
layer: INTEGER;
majorRadius: REAL;
minorRadius: REAL;
edges: SET OF Edge;
sketches: SET[0:?] OF Sketch;
loops: SET[0:?] OF FaceLoop;

END_ENTITY;

ENTITY Curve
SUBTYPE OF Object
SUPERTYPE OF ONEOF(Point, Line, Arc, Ellipse, Parabola,
Hyperbola, Spline);

length: REAL;

INVERSE:
mySketch: OPTIONAL Sketch FOR SET OF Curve;
ENTITY Point
SUBTYPE OF Curve;
    coords: LIST[0:2] OF REAL;
END_ENTITY;

ENTITY Line
SUBTYPE OF Curve;
    color: INTEGER;
    layer: INTEGER;
    endPoints: LIST[0:5] OF REAL;
END_ENTITY;

ENTITY Arc
SUBTYPE OF Curve;
    center: LIST[0:2] OF REAL;
    radius: REAL;
    startAngle: REAL;
    endAngle: REAL;
    color: INTEGER;
    layer: INTEGER;
    isClosed: INTEGER;
END_ENTITY;

ENTITY Ellipse
SUBTYPE OF Curve;
    center: LIST[0:2] OF REAL;
    startAngle: REAL;
    endAngle: REAL;
    majorAxis: REAL;
    minorAxis: REAL;
    isClosed: INTEGER;
END_ENTITY;

ENTITY Parabola
SUBTYPE OF Curve;

    center:    LIST[0:2] OF REAL;
startParameter:    REAL;
endParameter:    REAL;
shapeParameters:    LIST[0:1] OF REAL;
definingAxes:    LIST[0:5] OF REAL;

END_ENTITY;

ENTITY Hyperbola
SUBTYPE OF Curve;

center:    LIST[0:2] OF REAL;
startParameter:    REAL;
endParameter:    REAL;
shapeParameters:    LIST[0:1] OF REAL;
definingAxes:    LIST[0:5] OF REAL;

END_ENTITY;

ENTITY Spline
SUBTYPE OF Curve;

END_ENTITY;

ENTITY Dimension
SUBTYPE OF Object;

color:    INTEGER;
layer:    INTEGER;
exp:    Expression;

INVERSE:
    mySketch:    Sketch FOR SET OF Dimension;

END_ENTITY;

ENTITY Note
SUBTYPE OF Object;

textLines:    ARRAY OF STRING;

END_ENTITY;
ENTITY Expression;
  tag: INTEGER;
  name: STRING;
  value: REAL;

INVERSE:
  myCollection: Collection FOR SET OF Expression;
  myFeature: Feature FOR SET OF Expression;
  myDimension: Dimension FOR Expression;

UNIQUE:
  key: tag;
END_ENTITY;
---------------------------------------------------------------------

ENTITY PartsList;
  tag: INTEGER;
  textLines: ARRAY OF STRING;

UNIQUE:
  key: tag;
END_ENTITY;
---------------------------------------------------------------------

ENTITY FaceLoop;
  tag: INTEGER;
  loopType: INTEGER;
  edges: LIST[0:?] OF Edge;

INVERSE:
  myFace: Face FOR SET OF FaceLoop;

UNIQUE:
  key: tag;
END_ENTITY;
---------------------------------------------------------------------

END_SCHEMA;
Appendix B

Entities and Associated Methods

A list of all methods applicable to each entity type is described here. Details of a particular method may be found in C.

OBJECT Collection
METHODS:
  name
desc
getObjectsByName(name)
getObjects
getObjectsByType(type)
getGroupsByName(name)
getExps
getExpByName(name)
getRoot
getAttrs

OBJECT Object
METHODS:
type
name
getAttrs
groups
getReferenceSets

OBJECT solid
METHODS:
type
name
getAttrs
groups
getReferenceSets
density
volume
color
layer
surfArea
extent
getFeatures
getBrepElems

OBJECT feature
METHODS:
type
name
getAttrs
groups
generateReferenceSets
featureName
featureType
paramValues
paramNames
paramValueByName(name)
generateParentFeatures
generateChildFeatures
generateBrepElems
generateExps
generateExpByName(name)
generateSketches
generateSolid

OBJECT udf
METHODS:

OBJECT sketch
METHODS:
type
name
getAttrs
groups
generateReferenceSets
generateSketchElems
generateDims
generateExps
getExpByName(name)
color
layer
getFeatures
getFace

OBJECT instance
METHODS:
type
name
getAttrs
groups
getAddress
referenceSetName
color
layer
isRoot
getChildren
getParents

OBJECT referenceSet
METHODS:
type
name
getAttrs
groups
getAddress
color
layer
getObjects
getObjectsByName(name)
getObjectsByType(type)

OBJECT group
METHODS:
type
name
getAttrs
groups
getAddress
color
layer
getObjects
getObjectsByName(name)
getObjectsByType(type)
OBJECT brepElem
METHODS:
    type
    name
    getAttrs
    getGroups
    getReferenceSets
    getSolid
    getFeatures

OBJECT edge
METHODS:
    type
    name
    getAttrs
    getGroups
    getReferenceSets
    getSolid
    getFeatures
    getLength
    edgeType
    color
    layer
    getVerts
    getFaces

OBJECT lineEdge
METHODS:
    type
    name
    getAttrs
    getGroups
    getReferenceSets
    getSolid
    getFeatures
    getLength
    edgeType
    color
    layer
    getVerts
    getFaces

OBJECT arcEdge
METHODS:
    type
name
getAttrs
getGroups
getReferenceSets
getSolid
getFeatures
getLength
edgeType
color
layer
getVerts
getFaces
center
getRadius
startAngle
endAngle

OBJECT ellipseEdge
METHODS:
type
name
getAttrs
getGroups
getReferenceSets
getSolid
getFeatures
getLength
edgeType
color
layer
getVerts
getFaces
startAngle
endAngle
definingAxes
center
majorAxis
minorAxis

OBJECT splineEdge
METHODS:
type
name
getAttrs
getGroups
getReferenceSets
getSolid
getFeatures
getObject face
METHODS:
  type
  name
  getAttrs
  getGroups
  getReferenceSets
  getSolid
  getFeatures
  faceType
  area
  direction
  point
  extent
  getRadius
  minorRadius
  color
  layer
  getEdges
  getSketches
  getLoops

getObject loop
METHODS:
  getEdges
  loopType
  getFace

getObject curve
METHODS:
  getLength
  getSketch

getObject line
METHODS:
  getLength
  getSketch
  color
  layer
getEndPoints

OBJECT arc
METHODS:
    getLength
    getSketch
    isClosed
    center
    getRadius
    startAngle
    endAngle
    color
    layer
    paramValues

OBJECT point
METHODS:
    getLength
    getSketch
    coords

OBJECT ellipse
METHODS:
    getLength
    getSketch
    isClosed
    startAngle
    endAngle
    definingAxes
    center
    majorAxis
    minorAxis

OBJECT parabola
METHODS:
    getLength
    getSketch
    center
    definingAxes
    startParameter
    endParameter
    shapeParameters

OBJECT hyperbola
METHODS:
getLength
getSketch
center
definingAxes
startParameter
deParameter
shapeParameters

OBJECT spline
METHODS:
    getLength
    getSketch

OBJECT dimension
METHODS:
    type
    name
    getAttrs
    getGroups
    getReferenceSets
    getExp
    color
    layer
    getSketch

OBJECT note
METHODS:
    type
    name
    getAttrs
    getGroups
    getReferenceSets
    getLines

OBJECT partsList
METHODS:
    getLines

OBJECT expression
METHODS:
    name
    value
    getDims
Appendix C

CQL Methods

All the methods provided in CQL are described here.

C.1 Single-Valued Methods

Single-valued methods are those, that yield a scalar value upon execution. If the result of a method is undefined, then the dollar value is returned.

**center** Returns a triad of real numbers indicating the center point of the specified curve.

**Applicable Entities:** Arc, ArcEdge, Ellipse, EllipseEdge, Parabola, Hyperbola.

**collectionName** Returns the name of the part file that the instance refers to. This is frequently used to find the name of the file to be opened to get component information.

**Applicable Entities:** Instance.
**color** Returns an integer defining the display color of the entity. Applicable only to displayable entities.

**Applicable Entities:** Solid, Sketch, Instance, ReferenceSet, Group, Edge, Face, Line, Arc, Dimension, Ellipse, Parabola, Hyperbola, Note.

**definingDirection** Returns a triad of comma-separated real numbers representing the direction of the normal in case of a plane face and axis direction in case of cylindrical, cone and torus faces.

**Applicable Entities:** Face.

**definingPoint** Returns a triad of comma-separated real numbers representing a point on the plane face, a point on the axis for cylindrical and cone faces and the center in case of sphere and torus faces.

**Applicable Entities:** Face.

**density** Returns the density of the specified Solid in metric units.

**Applicable Entities:** Solid.

**desc** Returns a description of the part file if present.

**Applicable Entities:** Collection.

**edgeType** Returns an integer representing the type of edge. Possible types are Linear edge (3001), Circular edge (3002), Elliptical Edge (3003), Spline Edge (3005).

**Applicable Entities:** Edge.

**endAngle** Returns the end angle in degrees of the specified curve.
**Applicable Entities:** Arc, ArcEdge, Ellipse, EllipseEdge, Parabola, Hyperbola.

**endParameter** Real number representing the end parameter.

**Applicable Entities:** Parabola, Hyperbola.

**extent** Returns a triad of comma-separated real numbers indicating the spread of the solid/face in the x, y and z directions respectively.

**Applicable Entities:** Solid, Face.

**faceType** Returns an integer representing the type of the face. The possible types are cylinder(16), cone(17), sphere(18), surface of revolution(19), tabulated cylinder(20), bounded plane(22), fillet (blend)(23), b-surface(43), offset surface(65), foreign surface(66).

**Applicable Entities:** Face.

**featureName** Returns a character string representing the name of the feature. The name contains an integer time-stamp indicating the chronological order of creation of the feature relative to other features in the CAD system.

**Applicable Entities:** Feature.

**featureType** Returns a character string representing the type of the feature. The name contains an integer time-stamp indicating the chronological order of creation of the feature relative to other features in the CAD system.

**Applicable Entities:** Feature.

**getCollection** Returns the tag of the collection that an Instance refers to. In other words it returns the tag of the part file that has been opened to access instance information.

**Applicable Entities:** Instance.
**getExp** Returns the ID of the expression associated with a dimension entity.

**Applicable Entities:** Dimension.

**getExpByName(name)** Returns the ID of the expression with the specified name.

**Applicable Entities:** Collection.

**getFace** Returns the ID of the face that a sketch is located on if any or the face the face-loop belongs to.

**Applicable Entities:** Sketch, FaceLoop.

**getLength** Returns the length of the curve or edge. The length of the ellipse, hyperbola and spline edges are computed by approximating them by a set of closely spaced points and adding up the distances between these points.

**Applicable Entities:** Line, Arc, Ellipse, Parabola, Hyperbola, LineEdge, ArcEdge, EllipseEdge, SplineEdge.

**getParent** Returns the parent Instance of the specified Instance.

**Applicable Entities:** Instance.

**getRadius** Returns the radius of the specified curve/edge.

**Applicable Entities:** Arc, ArcEdge, Face.

**getReferenceSetByName(name)** Returns the object ID of the reference set with the specified name.

**Applicable Entities:** Collection.

**getRoot** Every assembly part file has an Instance entity called the ‘root’ instance which is at the top of the assembly structure. All components of the assembly
are child Instances of the root Instance. The presence of the root Instance can be used to check if the part file contains an assembly. Returns the root Instance of the part.

**Applicable Entities:** Collection.

**getSketch** Returns the ID of the sketch that the point, line, arc or dimension belongs to if any.

**Applicable Entities:** Point, Line, Arc, Dimension.

**getSolid** Returns the ID of the solid that the entity belongs to.

**Applicable Entities:** Edge, LineEdge, ArcEdge, EllipseEdge, SplineEdge, Face, Feature.

**instanceName** Returns the name of the specified instance.

**Applicable Entities:** Instance.

**isClosed** Returns a boolean value indicating whether the specified curve is closed.

**Applicable Entities:** Arc, Ellipse.

**isRoot** Returns a boolean value (.T. or .F.) depending on whether the specified Instance ID is a root instance in the active collection.

**Applicable Entities:** Instance.

**layer** Returns an integer indicating the layer of the drawing that the entity resides on.

**Applicable Entities:** Solid, Sketch, Instance, ReferenceSet, Group, Edge, Face, Line, Arc, Dimension, Ellipse, Parabola, Hyperbola, Note.
**loopType** Returns an integer denoting whether the Face Loop is an internal(2) or external(1) loop.

**Applicable Entities:** FaceLoop.

**majorAxis** Returns a triad of real numbers representing the major axis of the specified curve.

**Applicable Entities:** EllipseEdge, Ellipse.

**minorAxis** Returns a triad of real numbers representing the minor axis of the specified curve.

**Applicable Entities:** EllipseEdge, Ellipse.

**minorRadius** Represents the half angle in radians for a cone and the radius of the cross section of a torus.

**Applicable Entities:** Cone and Toroidal faces.

**name** Returns the name assigned to an object if present, file name of the part in case of a collection and the left hand side of the expression in case of an expression.

**Applicable Entities:** Object, Collection, Expression.

**paramValueByName(name)** Returns the parameter specified from a feature entity.

**Applicable Entities:** Feature.

**referenceSetName** Returns the name of the reference set in the instance’s part file that it references.

**Applicable Entities:** Instance.
**startAngle** Returns the start angle in degrees of the specified curve.

**Applicable Entities:** Arc, ArcEdge, Ellipse, EllipseEdge, Parabola, Hyperbola.

**startParameter** Real number representing the start parameter.

**Applicable Entities:** Parabola, Hyperbola.

**surfArea** Returns the surface area of the specified Solid in metric units.

**Applicable Entities:** Solid.

**type** Returns an integer denoting the entity type as defined in CQL C.2.

**Applicable Entities:** All CQL entities.

**value** Evaluates the right hand side of the specified expression and returns the resulting value.

**Applicable Entities:** Expression.

**volume** Returns the volume of the specified Solid in metric units.

**Applicable Entities:** Solid.

**xCoord** Returns the X co-ordinate of a point entity.

**Applicable Entities:** Point.

**yCoord** Returns the Y co-ordinate of a point entity.

**Applicable Entities:** Point.

**zCoord** Returns the Z co-ordinate of a point entity.

**Applicable Entities:** Point.
C.2 Multi-Valued Methods

All the following methods return an aggregate of scalar data types. An empty set is returned if no values result from the execution of a multi-method.

(coords) Returns a triad of real numbers representing the x, y and z coordinates of a point.

**Applicable Entities**: Point.

definingAxes Returns a set of six real numbers representing the x and y axes of the construction plane for the specified curve.

**Applicable Entities**: Arc, ArcEdge, Ellipse, EllipseEdge, Parabola, Hyperbola.

defAttrs Returns an aggregate of all the attributes specified for the given Collection or object. Each member of the aggregate is a character string of the form 'attribute_name=attribute_value'. It is possible that in some cases an attribute does not have a right-hand side. In such cases, the attribute appears as 'attribute_name='.

**Applicable Entities**: Collection, Object.

defBrepElems Returns an aggregate of all the faces and edges comprising the specified object.

**Applicable Entities**: Solid, Feature.

defChildFeatures Returns an aggregate of all features that the given feature interacts with that have been created after it.

**Applicable Entities**: Feature.
getChildren Gets all the child Instances of a specified Instance.

**Applicable Entities:** Instance.

getDims Returns all the dimension objects that are associated with a given Sketch or all the dimensions that a specified Expression is associated with.

**Applicable Entities:** Sketch, Expression.

getEdges Gets all the edges that comprise a Face or all the edges that comprise a given FaceLoop in the correct sequence.

**Applicable Entities:** Face, FaceLoop.

getEndPoints Returns a set of six real numbers representing the x, y and z coordinates for the endpoints of a line.

**Applicable Entities:** Line.

getExps Returns an aggregate of all the expression IDs associated with the given Collection, Feature or Sketch.

**Applicable Entities:** Collection, Group, Reference Set.

getFaces Gets all the Faces that a given Edge is associated with.

**Applicable Entities:** Edge.

getFeatures Returns an aggregate of all the features that the specified object is present in. In case of a Solid, this aggregate is comprised of all features present in the Solid.

**Applicable Entities:** Solid, Sketch, Edge, LineEdge, ArcEdge, EllipseEdge, SplineEdge, Face.
**getGroupsByName(name)** Returns an aggregate of all groups in the Collection, Group or Reference Set with the specified name.

**Applicable Entities:** Collection, Group, Reference Set.

**getLines** Returns an aggregate of all the lines of text comprising a Parts List or Note object.

**Applicable Entities:** Parts List, Note.

**getLoops** Creates Face Loop objects for the specified face and returns an aggregate of all Face Loop objects for the given face. Face Loop objects, internally, are transient across sessions. They are created during a session when requested and lost at the end of a CQL session.

**Applicable Entities:** Face.

**getObjects** Returns an aggregate of all CQL defined objects from Collections, Groups and Reference Sets.

**Applicable Entities:** Collection, Group, Reference Set.

**getObjectsByName(name)** Returns an aggregate of all objects with specified name from Collections, Groups and Reference Sets.

**Applicable Entities:** Collection, Group, Reference Set.
**getObjectsByType(type)** Returns an aggregate of all entities in the Collection, Group or Reference Set with the specified type. Entity types may be found defined in Appendix C.2.

**Applicable Entities:** Collection, Group, Reference Set.

**getParentFeatures** Returns an aggregate of all features that the given feature interacts with that have been created before the specified feature.

**Applicable Entities:** Feature.

**getReferenceSets** Returns an aggregate of all the Reference Sets that an object is present in.

**Applicable Entities:** Object.

**getSketchElems** Returns an aggregate of all the geometric elements (curves) used in the construction of a sketch. This does not include the graphical elements that make up dimensions and notes.

**Applicable Entities:** Sketch.

**getSketches** Returns all the sketches that a feature has been created with or all the sketches that are located on a face.

**Applicable Entities:** Sketch, Face.

**getVerts** Returns all the endpoints of an edge. An edge may have zero, one or two endpoints depending on its geometry. For each endpoint, a triad of real numbers representing the x, y and z coordinates of the vertex, is added to the set.

**Applicable Entities:** Edge.
**paramNames** Returns an aggregate of the names of all parameters for the feature specified.

**Applicable Entities:** Feature.

**paramValues** Returns an aggregate of all parameter values for the specified feature.

The order of parameters may be found by using the paramNames method.

**Applicable Entities:** Feature.

**shapeParameters** Returns a set of two real constants that govern the shape of a parabola or hyperbola.

**Applicable Entities:** Parabola, Hyperbola.
Appendix D

Types in CQL

Several entities in CQL have specialized ‘types’ associated to them based on their geometry. These ‘types’ are internally represented in CQL as positive integers. It may be necessary to use these integer types in conditional operators while framing queries on several occasions. For this reason, all the typed entities along with their integral type numbers are listed here.

D.1 Entity Types

The entity type (Table D.1) distinguishes one class of entities from another.

D.2 Face Types

A solid model face may have one of several possible geometries. The integral face types listed in Table D.2 are accessible through the ‘faceType’ method. Details of specific face geometries are available in the CAD/Modeling section of the Unigraphics online manual.
Table D.1: Entity Types

<table>
<thead>
<tr>
<th>Entity</th>
<th>Integer-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection</td>
<td>0</td>
</tr>
<tr>
<td>Object</td>
<td>1</td>
</tr>
<tr>
<td>Solid</td>
<td>2</td>
</tr>
<tr>
<td>Feature</td>
<td>3</td>
</tr>
<tr>
<td>Udf</td>
<td>4</td>
</tr>
<tr>
<td>Sketch</td>
<td>5</td>
</tr>
<tr>
<td>Instance</td>
<td>6</td>
</tr>
<tr>
<td>Group</td>
<td>7</td>
</tr>
<tr>
<td>ReferenceSet</td>
<td>8</td>
</tr>
<tr>
<td>BrepElem</td>
<td>9</td>
</tr>
<tr>
<td>Edge</td>
<td>10</td>
</tr>
<tr>
<td>LineEdge</td>
<td>11</td>
</tr>
<tr>
<td>ArcEdge</td>
<td>12</td>
</tr>
<tr>
<td>EllipseEdge</td>
<td>13</td>
</tr>
<tr>
<td>SplineEdge</td>
<td>14</td>
</tr>
<tr>
<td>Face</td>
<td>15</td>
</tr>
<tr>
<td>Curve</td>
<td>16</td>
</tr>
<tr>
<td>Point</td>
<td>17</td>
</tr>
<tr>
<td>Line</td>
<td>18</td>
</tr>
<tr>
<td>Arc</td>
<td>19</td>
</tr>
<tr>
<td>Ellipse</td>
<td>20</td>
</tr>
<tr>
<td>Parabola</td>
<td>21</td>
</tr>
<tr>
<td>Hyperbola</td>
<td>22</td>
</tr>
<tr>
<td>Spline</td>
<td>23</td>
</tr>
<tr>
<td>Dimension</td>
<td>24</td>
</tr>
<tr>
<td>Note</td>
<td>25</td>
</tr>
<tr>
<td>Expression</td>
<td>26</td>
</tr>
<tr>
<td>PartsList</td>
<td>27</td>
</tr>
<tr>
<td>FaceLoop</td>
<td>28</td>
</tr>
</tbody>
</table>
Table D.2: Face Types

<table>
<thead>
<tr>
<th>Face Geometry</th>
<th>Integer-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cylinder</td>
<td>16</td>
</tr>
<tr>
<td>cone</td>
<td>17</td>
</tr>
<tr>
<td>sphere</td>
<td>18</td>
</tr>
<tr>
<td>surface of revolution</td>
<td>19</td>
</tr>
<tr>
<td>tabulated cylinder</td>
<td>20</td>
</tr>
<tr>
<td>bounded plane</td>
<td>22</td>
</tr>
<tr>
<td>fillet (blend)</td>
<td>23</td>
</tr>
<tr>
<td>b-surface</td>
<td>43</td>
</tr>
<tr>
<td>offset surface</td>
<td>65</td>
</tr>
<tr>
<td>foreign surface</td>
<td>66</td>
</tr>
</tbody>
</table>

D.3 Feature Types

Every feature has a CAD system defined feature type that is represented as a character string in CQL. The feature types recognized in CQL are listed in Table D.3.

D.4 Color Codes

Displayable entities in the CAD system have a color associated with them that is obtained by the ‘color’ method. The color corresponding to the integral values obtained are listed in Table D.4.
Table D.3: Feature Types

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BallEndGroove</td>
<td>BALL.END.GROOVE</td>
</tr>
<tr>
<td>BallEndSlot</td>
<td>BALL.END.SLOT</td>
</tr>
<tr>
<td>Blend</td>
<td>BLEND</td>
</tr>
<tr>
<td>Block</td>
<td>BLOCK</td>
</tr>
<tr>
<td>Boss</td>
<td>BOSS</td>
</tr>
<tr>
<td>CounterBoredHole</td>
<td>CBORE.HOLE</td>
</tr>
<tr>
<td>Chamfer</td>
<td>CHAMFER</td>
</tr>
<tr>
<td>CircularFeaturePattern</td>
<td>CIRCULAR.ISET</td>
</tr>
<tr>
<td>Cone</td>
<td>CONE</td>
</tr>
<tr>
<td>CounterSunkHole</td>
<td>CSUNK.HOLE</td>
</tr>
<tr>
<td>Cylinder</td>
<td>CYLINDER</td>
</tr>
<tr>
<td>CylindricalPocket</td>
<td>CYL.POCKET</td>
</tr>
<tr>
<td>DovetailSlot</td>
<td>DOVE.TAIL.SLOT</td>
</tr>
<tr>
<td>Hollow</td>
<td>HOLLOW</td>
</tr>
<tr>
<td>LinearFeaturePattern</td>
<td>LINEAR.ISET</td>
</tr>
<tr>
<td>BooleanFeature</td>
<td>META</td>
</tr>
<tr>
<td>Offset</td>
<td>OFFSET</td>
</tr>
<tr>
<td>RectangularGroove</td>
<td>RECT.GROOVE</td>
</tr>
<tr>
<td>RectangularPad</td>
<td>RECT.PAD</td>
</tr>
<tr>
<td>RectangularPocket</td>
<td>RECT.POCKET</td>
</tr>
<tr>
<td>RectangularSlot</td>
<td>RECT.SLOT</td>
</tr>
<tr>
<td>SimpleHole</td>
<td>SIMPLE.HOLE</td>
</tr>
<tr>
<td>Sphere</td>
<td>SPHERE</td>
</tr>
<tr>
<td>SweptFeature</td>
<td>SWP104</td>
</tr>
<tr>
<td>Taper</td>
<td>TAPER</td>
</tr>
<tr>
<td>ThickenedSheet</td>
<td>THICKEN.SHEET</td>
</tr>
<tr>
<td>T-Slot</td>
<td>T_SLOT</td>
</tr>
<tr>
<td>U-Groove</td>
<td>U.GROOVE</td>
</tr>
<tr>
<td>U-Slot</td>
<td>U SLOT</td>
</tr>
</tbody>
</table>
Table D.4: Color Codes

<table>
<thead>
<tr>
<th>Color</th>
<th>Color-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLUE</td>
<td>1</td>
</tr>
<tr>
<td>GREEN</td>
<td>2</td>
</tr>
<tr>
<td>CYAN</td>
<td>3</td>
</tr>
<tr>
<td>RED</td>
<td>4</td>
</tr>
<tr>
<td>MAGENTA</td>
<td>5</td>
</tr>
<tr>
<td>YELLOW</td>
<td>6</td>
</tr>
<tr>
<td>WHITE</td>
<td>7</td>
</tr>
<tr>
<td>OLIVE</td>
<td>8</td>
</tr>
<tr>
<td>PINK</td>
<td>9</td>
</tr>
<tr>
<td>BROWN</td>
<td>10</td>
</tr>
<tr>
<td>ORANGE</td>
<td>11</td>
</tr>
<tr>
<td>PURPLE</td>
<td>12</td>
</tr>
<tr>
<td>DARK RED</td>
<td>13</td>
</tr>
<tr>
<td>AQUAMARINE</td>
<td>14</td>
</tr>
<tr>
<td>GRAY</td>
<td>15</td>
</tr>
</tbody>
</table>
Appendix E

The Unigraphics API

The Unigraphics API is a set of C and FORTRAN routines that provides access to part file data. Detailed information on the use of these routines can be found in the online Unigraphics documentation under the section ‘Languages’. Whenever class nodes, method nodes, multi-method nodes and system nodes are executed, API functions are called. In addition, operator nodes like the suppress and unsuppress nodes also need to invoke the API functions. All UG API functions names begin with the prefix ‘UF_’. The sections below list the API functions used in the implementation of the major tasks in CQL along with details where necessary.

E.1 Preliminary Tasks

Before any operation can be performed with part files, a license must be obtained from the license manager daemon. This license must be returned at the termination of a CQL session. This is accomplished with the following routines.

.cql.get_ug_license The ‘UF_initialize’ function is called to obtain a license. Failure, like in the case of most UG API functions, results in a non-zero return value.
After a license is obtained, this routine uses the ‘UF_ASSEM_ask_assem_options’ and ‘UF_ASSEM_set_assem_options’ functions to set options to load all assembly components fully when assembly part files are opened. Failure to do so may cause the system to crash on some types of queries on assemblies. An option to ensure that the work part and the displayed part are the same when a part is re-opened is also set. This is extremely important in case of assembly files.

_cql_exit_cad_system_ug The function ‘UF_terminate’ is called to return the license obtained.

### E.2 Single-Valued Methods

The API functions used in the implementation of single-valued methods are listed here. Wherever mere function calls based on the appropriate entity type are needed, the names of the API functions alone are provided.

- **name**  
  - UF_PART.ask_part.name,  
  - UF_OBJ.ask_name  
  - UF_MODL.ask.exp_tag.string are used to obtain the names of Collections, Objects and Expressions respectively.

- **desc**  
  - UF_PART.ask.description is used.

- **getReferenceSetByName(name)** _cql.get_objects_by_type is called which returns all reference set objects. The names of these objects is checked for the specified name using UF_OBJ.ask.name. If an object does not have a name assigned to it, this API function returns an error number (integer 1155001, defined in CQL as NO_NAME). All other error codes mean failure of the function.
**getExpByNam**e(name) **UF_MODL\_ask\_exp\_s\_of\_part** is used to get the tags of all expressions in the Collection. **_cql\_get\_exp\_name\_from\_tag** is used to obtain the name of a given expression and check for the specified name.

**getRoot** **UF\_ASSEM\_ask\_root\_part\_occ** is used.

**type** No API functions are used. The CQL defined entity type is returned.

**loopType** **UF\_UDOBJ\_ask\_udo\_data** is used to access data stored in the user-defined FaceLoop object regarding its type. The address of a UG defined data structure of type **UF\_UDOBJ\_all\_data\_t** is supplied to the function. **UF\_UDOBJ\_free\_udo\_data** is used to deallocate any dynamic memory used by this data structure after its use.

**density** **UF\_MODL\_ask\_mass\_props\_3d** is used with the appropriate options set. From the array of values for different mass properties, the value for density is returned.

**volume** **UF\_MODL\_ask\_mass\_props\_3d** is used.

**surfArea** **UF\_MODL\_ask\_mass\_props\_3d** is used.

**extent** **UF\_MODL\_ask\_bounding\_box** is used to obtain co-ordinates of the end-points of the solid diagonal of the bounding box. From these co-ordinates, the x, y and z extents are computed.

**featureName** **UF\_MODL\_ask\_feat\_name** is used.

**featureType** **UF\_MODL\_ask\_feat\_type** is used.

**paramValueByName(name)** **_cql\_get\_feat\_params** is called with the name of the parameter desired.
getSolid UF_MODL_ask_feat_body, UF_MODL_ask_edge_body or UF_MODL_ask_face_body is used depending on the entity whose parent solid is desired.

definingDirection UF_MODL_ask_face_data is used.

definingPoint UF_MODL_ask_face_data is used.

minorRadius UF_MODL_ask_face_data is used.

color UF_OBJ.ask.display.properties is used and is supplied with the address of the UG defined UF_OBJ.disp.props.t structure. The field corresponding to the color is returned from the data structure.

layer UF_OBJ.ask.display.properties is used.

getFace If this method is called on a sketch entity, a search is necessary. All face objects in the Collection are iterated over with the UF_OBJ.cycle_objs_in_part method. For each face object, all its associated sketches are obtained with the UF_SKET_ask_face.sketches method to see if the given sketch is present. When present, the corresponding face tag is returned. In case of FaceLoops, UF_UDOBJ.ask.udo.data is used to access FaceLoop information. The associated face is returned from the Loop information obtained.

collectionName UF_ASSEM_ask_component_data is used.

referenceSetName UF_ASSEM_ask_component_data is used.

instanceName UF_ASSEM_ask_component_data is used.

getParent UF_ASSEM_ask_part_occurrence is used.
**getCollection**  UF_ASEM.ask.prototype.of_occ is used.

**isRoot**  UF_ASEM.ask.root.part_occ is used to obtain the tag of the root instance and is checked with that of the given instance.

**getLength**  _cql.get.curve.data is used to obtain the required parameters from the curve/edge considered. For lines and LineEdges, the distance between the end-points is computed. For arcs and ArcEdges, the length of the circular arc is computed. The length of the curve is computed by integration for a parabola using the curve parameters. In case of curves and edges of type spline, ellipse or hyperbola, an approximation method is used. UF_MODL.ask.curve.points is used to obtain a number of points on the curve based on a chordal tolerance and angular tolerance specified. The distance between these points is accumulated to arrive at the length of the curve.

**faceType**  UF_MODL.ask.face.type is used.

**edgeType**  UF_MODL.ask.edge.type is used.

**getSketch**  The function _cql.get.my.sketch is used to obtain the sketch corresponding to the specified entity.

**isClosed**  UF_MODL.ask.curve.closed is used.

**center**  _cql.get.curve.data is used to obtain curve parameters. The value corresponding to the center is returned from the obtained curve parameters.

**startAngle**  _cql.get.curve.data is used.

**endAngle**  _cql.get.curve.data is used.
getRadius UF_CURVE.ask_arc_data is used for Arcs and ArcEdges while
UF_MODL.ask_face_data is used for Faces.

majorAxis .cql.get.curve.data is used.

minorAxis .cql.get.curve.data is used.

startParameter .cql.get.curve.data is used.

endParameter .cql.get.curve.data is used.

xCoord UF_CURVE.ask_point.data is used and the value corresponding to the x­
coordinate is returned.

yCoord UF_CURVE.ask_point.data is used.

zCoord UF_CURVE.ask_point.data is used.

getExp UF.SKET.ask_dim.status is used.

value UF.MODL.ask.exp_tag.value is used.

E.3 Multi-Valued Methods

getObjects Calls .cql.get.objects.by.type, .cql.get.objects.from.ug.group or
.cql.get.objects.from.group depending on whether the objects of a Collection,
Group or ReferenceSet are required.

getObjectsByName(name) In case of a Collection, gets all CQL defined objects
from the Collection using .cql.get.objects.by.type. Then the names of all these
objects is matched with the specified name using UF_OBJ.ask.name and objects
selected are accumulated and returned. In case of Groups and ReferenceSets,
_cql.get.objects.from.ug.group and _cql.get.objects.from.group are used to get all objects with the specified name.

**getGroupsByName(name)** Works exactly as above, except that instead of all objects, only Groups are extracted.

**getObjectsByType(type)** Uses _cql.get.objects.by.type, _cql.get.objects.from.ug.group, _cql.get.objects.from.group depending on whether objects are to be extracted from the Collection, a ReferenceSet or Group.

**getExps** Uses UF.MODL.ask.exps.of.part in case of a Collection. In case of a Feature, UF.MODL.ask.exps.of.feature is used. This function sometimes yields tags of non-existent objects. Hence each tag obtained is checked for presence in the part file by using UF.OBJ.ask.type.and.subtype and verifying if the error code obtained is that for an invalid object. In case of sketches, gets all the associated dimension tags using _cql.get.sketch.info and then getting the expression tag corresponding to each dimension using UF.SKET.ask.dim.status.

**getAttrs** Uses UF.ATTR.ask.part.attrs with a UG defined attribute data structure UF.ATTR.part.attr.p.t to get Collection attributes. Collection attributes are always strings and are constructed using the name and value of the attribute obtained from the attribute data structure obtained. In case of object attributes, uses UF.ATTR.cycle to cycle through all object attributes present. Object attributes may be of type integer, real, time, string, reference or null. Wherever required a conversion is made to string type and a string of the form ‘name=value’ is obtained for each attribute. A set of these strings is constructed.
**getReferenceSets**  
UF ASSEM count ref sets in is used to get all the ReferenceSets that an object belongs to.

**getGroups**  
First gets all the Groups in the part using .cql.get.objects.by.type. Now uses UF GROUP ask group data to get all objects in each Group and verifies if the given object is present. All Groups in which the object is present are accumulated.

**getFeatures**  
UF MODL ask body feats is used in case of a Solid and UF SKET ask sketch features is used in case of a Sketch. In case of Edges and Faces, UF MODL ask edge feats and UF MODL ask face feats are used respectively.

**getBrepElems**  
In case of a Solid, all associated edges and faces are obtained by using UF MODL ask body faces and UF MODL ask body edges respectively and accumulated. In case of features, UF MODL ask feat faces and UF MODL ask feat edges are used.

**paramValues**  
In case of features, .cql.get_feat.params is used. Arc parameters are obtained using UF CURVE ask arc.data with the data structure UF CURVE arc t and the values of the center, radius, start angle and end angle are accumulated.

**paramNames**  
.cql.get_feat.params is used with an argument indicating that only parameter names are desired.

**getParentFeatures**  
UF MODL ask feat relatives is used and the parent features are accumulated from the arrays of parents and children returned.
**getChildFeatures**  Works exactly as above except that the tags of the child features are accumulated.

**getSketches**  UF_SKET_ask_feature_sketches is used.

**getLoops**  This method generates the loop(s) of edges for a face if they don’t exist or merely retrieves them if they have already been generated. The global variable _cql.ug.Faces.w.loops contains a list of face tags for faces whose loops have been generated.

If the given face is not present in this list, then UF.MODL.ask.face.loops is used to construct a list of loops. For each loop, _cql_make.loop.array is called to construct an array with all the edges comprising the loop along with the tag of the given face and an integer that determines if the loop is an internal or external loop.

If the given face is present in the global list, then UF.UDOBJ_cycle.udos_by.class is used to cycle through all face loop objects. The data contained in each FaceLoop object is examined using UF.UDOBJ_ask.udo.data. All FaceLoops with parent face tag equal to that of the given face are accumulated and returned.

**getSketchElems**  _cql.get.sketch.info is used with option GEOM_INFO defined in CQL to obtain IDs of all Sketch geometry.

**getDims**  _cql.get.sketch.info is used with option DIM_INFO to obtain IDs of all Dimensions in case of a Sketch. In case of Expressions, all the Sketches in the part are cycled through using UF.OBJ_cycle.objs.in.part. For each Sketch, its Dimensions and the corresponding Expressions are obtained using
_cql_get_sketch_info and UF.SKET.ask.dim.status. If the given Expression is obtained, its corresponding Dimension is returned.

**getChildren**  UF.ASSEM_ask_part_occ.children is used.

**getVerts**  UF.MODL_ask_edge_verts is used.

**getFaces**  UF.MODL_ask_edge_faces is used.

**getEdges**  UF.MODL_ask_face_edges is used in case of a Face. In case of FaceLoops, UF.UDOBJ_ask_udo_data is used.

**getEndPoints**  UF.CURVE_ask_line_data is used.

**definingAxes**  _cql.get.curve.data is used to get all the parameters of the curve. The values corresponding to the axes of the plane in which the curve is constructed are returned.

**shapeParameters**  _cql.get.curve.data is used and the values corresponding to the shape parameters K1 and K2 are returned.

**coords**  UF.CURVE_ask_point_data is used.

**getLines**  In case of a PartsList, a Note object needs to be created to access the text. Hence a temporary Note object is created using UF.PLIST.create_note which is destroyed at the end of this method using UF.OBJ.delete_object. UF.PLIST_ask_tag is used to obtain the tag of the PartsList entity for the Collection and UF.PLIST_ask_tag_of_note is used to check if the PartsList object has any associated Note objects. UF.DRF_ask.draft.aid.text_info is used to obtain the text associated with the generated Note in case of the PartsList and the supplied Note in case of a
Note. After the temporary Note object is deleted in case of the PartsList, UF_OBJ_ask_status is used to check if the Note has indeed been deleted.

### E.4 Support Routines

Some of the important support routines involving further use of the API functions are described here.

- **cql.get.objects.by.type** This function takes in a data structure which is an aggregate v_node containing the types and subtypes of all objects required to be obtained. Each of these object types is extracted sequentially. All the objects in the part are cycled through using UF_OBJ_cycle_objs_in_part. The type and subtype of each object obtained is checked using UF_OBJ_ask_type_and_subtype. In case of a match, the object is selected and accumulated. Apart from the UG defined type and subtype, in case of Edge objects, the type of Edge needed may be specified in the data structure. This is checked using cql.get.curve.data. All Objects except Dimension objects can be obtained by the process described above. For Dimension objects, all Sketches in the part are extracted by a recursive call to cql.get.objects.by.type. Now, cql.get.sketch.info is used to obtain the Dimensions for all the Sketches in the part.

- **cql.get.objects.from.group** This routine gets all Objects with a specified type from a ReferenceSet. If a name is specified, only Objects with the specified name are extracted. All objects in the part are first accumulated using cql.get.objects.by.type. For each of these objects, all the ReferenceSets that they belong to are obtained using UF_ASSEM_count_ref_sets_in and UF_ASSEM_get_ref_set_inst. If the object is found to belong to the given Ref-
referenceSet, it is accumulated to a list. If a name is specified, only Objects with
the matching name are checked.

\_cqI\_get\_objects\_from\_ug\_group All objects from the specified Group are obtained
using UF\_GROUP\_ask\_group\_data. Each Group object is checked for the CQL
type specified using \_cql\_get\_cql\_type and for the name if specified. Matched
objects are accumulated.

\_cql\_get\_curve\_data The data structure containing the curve parameters first needs
to be obtained using UF\_CURVE\_ask\_curve\_struct. Now, UF\_CURVE\_ask\_curve\_struct\_data is used to access the data contained. This
data structure is deleted with UF\_CURVE\_free\_curve\_struct after use.

\_cql\_get\_my\_sketch All the Sketches in the part are cycled through. For each
Sketch, its Dimensions are extracted in case of Dimension objects and the tags
of the objects comprising its geometry are extracted in case of Points, Lines
and Arcs. If the given object is present in the list of Objects obtained from the
Sketch, the Sketch is returned.

\_cql\_get\_sketch\_info Before Sketch data can be obtained, the Sketch must be ini-
tialized using UF\_SKET\_initialize\_sketch. In order to obtain the name of the
Sketch to be initialized, UF\_OBJ\_ask\_name is used. After initialization,
UF\_SKET\_ask\_info can be used to extract tags of geometry or Dimensions de-
pending on the arguments supplied. After this is complete, the Sketch environ-
ment is un-initialized using UF\_SKET\_terminate\_sketch.

\_cql\_get\_feat\_params The number of API functions used are too many to list here.

Basically, for each type of feature, a separate API function needs to be used
to extract feature parameters. If only parameter names are required, these are
accumulated for the specified type of feature from a pre-defined array
and returned. Depending on the specified feature type, the appropriate API
function is called to obtain its parameters. If a named parameter is desired,
only that parameter is returned, else all parameters in the correct order are
returned.

_cql_perform_suppress_ug Everytime a suppress is performed, a list of all fea-
tures already suppressed and those required to be suppressed is appended using
_cql_append_lists and supplied to UF_MODL_suppress_feature. This is nec-
essary, otherwise sequentially suppressing objects one-by-one in a part file may
cause the UF_MODL_suppress_feature function to crash. Duplicate objects may
be present in the suppress list handed to UF_MODL_suppress_feature. After
suppression is complete, _cql_form_suppressed_list is used to obtain a list of all
suppressed objects to return as the result of the suppress query.

_cql_perform_unsuppress_ug If no list of features to unsuppress is provided, all
suppressed features are obtained using UF_MODL_ask_suppress_list and unsuppressed
using UF_MODL_unsuppress_feature. If a list of features to be unsuppressed is pro-
vided, each object in the list is first verified to be suppressed using
UF_MODL_ask_suppress_feature. Features in the list that are not suppressed
are removed.
E.5 De-allocating Dynamic Memory

The API provides functions to de-allocate memory allocated by UG API functions. These cannot be used to de-allocate any memory not directly allocated by an API function. If attempted, it may cause a program crash.

**UF _MODL_delete_list** Used to delete structures of type uf.list.p.t.

**UF_free** Used to de-allocate any UG allocated array.

**UF _UDOB OJ_free_u do _data** Used to de-allocate structures of type UF UDOB OJ_all.data.t.

**UF_free_string_array** Used to free an array of UG allocated strings.

**UF _CURVE_free_curve_struct** Used to delete structures of type UF_CURVE.struct.p.t.

E.6 System Functions

This section outlines the use of the API functions in the implementation of system functions.

**open** UF_PART.ask_part.tag is first used to check if a part tag exists for the named part, meaning that the part file has already been opened. If the part file is not open, it is opened using UF_PART.open. In case an assembly file is opened, all its component part files are fully loaded into memory too (because of option set by .cql.load.components_fully). If this is not done, queries on component files subsequently opened may cause a UG crash. If the part is found to have already been opened, it is merely set to be the ‘display part’ using
UF_PART_set_display_part. In case of an assembly, the part displayed and the part actually worked on may be different. The display part and the work part are always set to be the same while using CQL as this was found to be a safe way to switch between open parts. This is done using _cql_load_components_fully.

close UF_PART_close_all is used to close all open parts.

search To set search directories, the current assembly options data structure of type UF_ASEM_options_t is first retrieved using UF_ASEM_ask_assem_options. After setting the field (load_options) to load from search directories, these options are made effective by using UF_ASEM_set_assem_options. Now the search directories need to be defined. An array of strings containing user specified load paths is constructed. Another array of integers is constructed which specifies whether for each path, the subdirectories in the path should also be searched for component parts. This is set to be true. UF_ASEM_set_search_directories is then called with these arrays to set search directories.

E.7 User Defined Objects

User defined objects (UDO's) are a powerful way to define new kinds of entities with user specified behaviour in Unigraphics. FaceLoop entities have been implemented using UDO's. For a complete discussion of UDO's refer to the User Defined Objects section of the UG/Open API Reference. Class specifications and methods for UDO's are loaded at start time by UG. To load the shared library containing object code for defining UDO's, UG looks in a directory with a specified structure. This directory is indicated in this implementation by the environment variable 'UGI_SITE_DIR'.
The shared library name must have the extension `.so` for the SGI platform and `.sl` for the HP platform. Further all code that defines new classes must be placed in a function called `ufsta` and must not contain any code that attempts to access part files.

**ufsta** This obtains a license and invokes the `UFX_UDO_create` function that defines the FaceLoop class. After the class is defined, the license is returned.

**UFX_UDO_create** Uses `UF_UDOBJ_create_class` to define a FaceLoop class. At this time, no instances of this class are inserted into any part files. This only serves as a definition for the FaceLoop object.

The structure of the data held in the FaceLoop object is shown in figure E.1. The `getLoops` method instantiates FaceLoop objects with this structure. All the data in a FaceLoop is held as an array of integers. The first element in the array is the tag of the parent Face. The next entry indicates whether the loop is an internal or external loop. This is followed by the tags of the edges that comprise the loop in order.
Figure E.1: Structure of a FaceLoop
Appendix F

TNR Demonstration Programs

A listing of all the TNR programs used in the demo is provided below.

F.1 casing.tnr

template
main()
{
    cad_system("ug");
    cad_part("/usr/local_4/people/student/athreya/Thesis/Thesis_Test" | "parts/test_part.prt");

    process_feats();
    cavity("CAVITY1");
    cavity("CAVITY2");
}

// Gets solid data and pattern info
template
process_feats()
var result[]; s; i; sol_out[]; s_holes[]; pockets[]; patterns[];
    instances[]; vars[]; dmy[]; stamp; master[];
{
    cql_exec("select x, x(volume, density, extent) from x in Solid",
        result[]," ");

    for(s=0; s<$result[]; s++){
set_size(vars[], 0);
splitd(result[s], ",", vars[]);
printf("*************Solid tag: %d **************\n", vars[0]);
printf("Volume: %f  Density: %f  Extents: %s \n\n\n", 
    vars[1], vars[2], vars[3]);

cql_exec("select y, y(featureName) from x in Solid," |
        "  y in x->getFeatures " |
        "  where x == [vars[0]] " and y->featureType match ".*SET.*" |
        " patterns[]", " ");
for(i=0; i<#patterns[]; i++){
    set_size(dmy[], 0);
splitd(patterns[i], ",", dmy[]);
    printf("---------Feature Pattern '%s'--------\n", dmy[1]);
    set_size(instances[], 0);
    cql_exec("select y, y(featureName) from x in Solid," |
        "  y in x->getFeatures " |
        "  where x == [vars[0]] " and " |
        "  y->featureName == " | get_master_name(instances[0]) |
        "\n", "", master[], " " );
    printf("Pattern Feature: '%s'\n", get_master_name(instances[0]));
    printf("Feature parameters: %s\n", master[0]);
    printf("\n");
}

printf("------------------Simple Holes present ------------------\n");
get_feature_parms(vars[0], "SIMPLE HOLE");
printf("------------------Cylinders present ------------------\n");
get_feature_parms(vars[0], "CYLINDER");
get_volume(vars[0], "CYLINDER");
printf("\n-----------------Rectangular Pockets present -----------------\n");
get_feature_parms(vars[0], "RECT_POCKET");
get_volume(vars[0], "RECT_POCKET");

//Computes the volume of the specified cavity. The cavity is a void
//enclosed by faces with the specified name.

//Heuristic Used: Get the volume of the solid (say vol1). Determine all the
//features that share the faces of the cavity. From these, cull out the
//primitives (spheres, cones, blocks, cylinders). Suppress these and get
// the solid volume (say \text{vol}_2). This is the volume of the cavity with the
// contributing primitives and all children suppressed. Now unsuppress
// the primitives alone, children remain suppressed. Get the solid volume.
// The volume of the cavity is \((\text{vol}_2 - \text{vol}_1) - (\text{vol}_3 - \text{vol}_1)\).

template
cavity(name)

var results[]; i; dmy[]; suppress_str; vol[];
tot_vol[]; vol\text{-}wo\text{-}cav\text{-}and\text{-}child[]; vol\text{-}wo\text{-}child[]; cavity\_vol;
{
cql\text{-}exec("select y, y->featureType from x in Face," |
" y in x->getFeatures " |
" where x->name == \"\"|name|\" \", results[], " ");
if(#results[] == 0)
return;

printf("Processing \%s...\n",name);
suppress\_str = ";
for(i=0; i<#results[]; i++){
set\_size(dmy[], 0);
if(is\_primitive(results[i]) == 0){
split\_d(results[i], ",", dmy[]); 
merged(suppress\_str, ",", suppress\_str, dmy[0]);
}
}
sub("\[\[\]",", suppress\_str);
merged(suppress\_str, ",", suppress\_str, ");
}
cql\text{-}exec("select Solid.volume",tot\_vol[]," ");
printf("volume before suppression = \%f\n",tot\_vol[0]);
set\_size(dmy[], 0);
cql\text{-}exec("suppress x from x in " |suppress\_str| " ",dmy[]);
cql\text{-}exec("select Solid.volume",vol\text{-}wo\text{-}cav\text{-}and\text{-}child[]," ");
printf("volume after suppression of primitives and all children = \%f\n",vol\text{-}wo\text{-}cav\text{-}and\text{-}child[0]);
set\_size(dmy[], 0);
cql\text{-}exec("unsuppress x from x in " |suppress\_str| " ",dmy[]);
cql\text{-}exec("select Solid.volume",vol\text{-}wo\text{-}child[]," ");
printf("volume after suppression of children alone = \%f\n",vol\text{-}wo\text{-}child[0]);
cavity\_vol = (vol\text{-}wo\text{-}cav\text{-}and\text{-}child[0] - tot\_vol[0]) -
(vol\text{-}wo\text{-}child[0] - tot\_vol[0]);
printf("Cavity volume = \%f\n\n", cavity\_vol);
cql\text{-}exec("unsuppress",dmy[]);
}
// Used to check if a particular feature is a primitive feature
// The input argument is a string containing the feature type
template
is_primitive(feature_string)
  var ref[]; i;
  { ref[0] = "BLOCK";
    ref[1] = "CYLINDER";
    ref[2] = "CONE";
    ref[3] = "SPHERE";
    ref[4] = "SWP104";
    for(i=0; i<#ref[]; i++)
      if (match(feature_string, ref[i]) != 0)
        return 0;
  return -1;
}

// Gets the volume of features of specified type and all children by
// suppression
template
generate_volume(solid, feat_type)
  var flag=0; feats[]; dmy[]; voll[]; vol2[]; i; abs;
  {
    cql_exec("select y from x in Solid, y in x->getFeatures " |
      " where x =="|solid|" and y->featureType == "|feat_type|" \"",
    feats[], " ");
    if(#feats[] == 0)
      return;
    printf("Volume");
    for(i=0; i<#feats[]; i++){
      set_size(dmy[],0);
      cql_exec("select x->getChildFeatures from x in ("|feats[i]|") " ,
        dmy[], " ");
      if(strcmp(dmy[0],"()") == 0)
        flag = 1;
      cql_exec("select x->volume from x in ("| solid |") " , voll[], " ");
      cql_exec("suppress x from x in ("|feats[i]|")",dmy[], " ");
      cql_exec("select x->volume from x in ("|solid|"), vol2[], " ");
      cql_exec("unsuppress",dmy[]);
      if(flag != 1)
        printf(" of feature and all children: ");
      if((abs = (voll[0]-vol2[0])) < 0)
        abs = -abs;
      printf(": %f 
",abs);
    }
  }

// Extracts parameters from all features of the specified type
// from the given solid.
template
get_feature_parms(solid, feat_type)
    var dmy[], i, str; results[];
{
    set_size(dmy[0], 0);
    str = ".*INSTANCE\[[0-9]+].*";
    cql_exec("select y->featureName, y(paramNames, paramValues) " |
        " from x in Solid, y in x->getFeatures " |
        " where x =="|solid|" and y->featureType == "|feat_type|" " |
        ,dmy[0], " ");
    for(i=0; i<#dmy[0]; i++)
        printf("%s\n", dmy[i]);
}

// To extract parameters from features that are used as templates
// in an instance as a result of which their featureType has changed
set_size(dmy[0], 0);
    cql_exec("select y->featureName from x in Solid, y in x->getFeatures " |
        " where x =="|solid|" and y->featureName match \"|str|\" and " |
        " y->featureName match \".*|feat_type|.*\" |
        ,dmy[0], " ");
    for(i=0; i<#dmy[0]; i++)
        printf("%s ", dmy[i]);
    cql_exec("select x(paramNames, paramValues) from x in Feature " |
        " where x->featureName == \"|get_master_name(dmy[i])|\" ",
        results[0], " ");
    printf("%s \n", results[0]);
}

// Extracts the time stamp with surrounding braces from a featureName
template
get_time_stamp(s)
{ return extract(s, 1, "/[0-9]+/[0-9]+/\")
}

// Gets the featureName of the template feature for the pattern
template
get_master_name(s)
    var tmp;
{ tmp = extract(s, 1, "/\";gsub("/\", ",", tmp);
    return tmp;  
}
template
main()
var result[]; i; str; plist[]; dmy[];
{
str = ".*WALL THK.*\n";
cad_system("ug");
cad_part("/usr/local_4/people/student/athreya/research/ug/" |
"test_parts/1898m20_a_2.prt");
printf("-------------------List of Parts---------------------\n");
cql_exec("select y from x in PartsList, y in x->getLines",
    plist[], " ");
for(i=0; i<#plist[]; i++)
    printf(plist[i]);
printf("\n\n");
printf("---------------Summary Of Tubes Present----------------");
cql_exec("select y->getLines from x in Group, " |
    "y in x->getObjectsByType(Note) " | 
    " where for z in x->getObjectsByType(Note).getLines"|
    " any (z MATCH ".*" | str |")", result[], " ");
printf("Tube# Material Dev.Length Specs. \n");
printf("---------------------------------------------\n");
for(i=0; i<#result[]; i++)
    gsub("[()J", "", result[i]);
for(i=0; i<#result[]; i++){
    if(match(result[i], "-[0-9][0-9]*") != 0){
        print(result[i],"t",result[i+1],"t");
i++;
    }
    print_dev_length(result[i]);
i++;
    print("\t",result[i], " ");
    if(match(result[i], "WALL THK") != 0)
        print("\n");
}
printf("\n");

//Prints the developed length for a tube given the run section string
template
print_dev_length(run)
var dmy[]; i;
{ cql_exec("select x->getLines from x in Note " |
    "where for z in x->getLines any (z MATCH \".*\" | run |
".\*LENGTH.\*"", dmy[]);
    printf(" %s ", extract(dmy[0], 1, [.99] [.99] [.99] [.99] [.99] [.99]));
}

// Print a line of the parts list
template
print_line(str)
    var dmy[]; i;
    { if(match(str, "^[ ]*[0-9]"") != 0)
        printf(" \n");
        set_size(dmy[], 0);
        splitd(str, "[<>]", dmy[]);
        for(i=0; i<#dmy[]; i++){
            if(match(dmy[i], " ") != 0)
                printf(" ");
            if(strcmp(dmy[i], "N") == 0)
                continue;
            if(match(dmy[i], "^[ ]*[0-9]"") != 0)
                continue;
            if(match(dmy[i], "^[ ]*[B]"") != 0)
                print_blanks(dmy[i]);
            else
                printf("%s", dmy[i]);
        }
    }

// Print the correct number of blanks based on format specified
template
print_blanks(str)
    var tmp; i;
    { sub("^[ ]*[B]", ",", str);
        tmp = int(str + .5);
        for(i=0; i<tmp; i++)
            printf(" ");
    }

F.3 parker2.tnr

template
main()
{
    cad_system("ug");
cad_part("/usr/local_4/people/student/athreya/Thesis/Thesis_Test" | "/parts/parker2.prt");
get_turningParms("SECTION");

// Gets machining length and mean diameter of part using the cross
// section specified

template
get_turningParms(name)

var results[]; i; dmy[]; machined_length = 0; vars[]; num_dias=0;
mean_dia=0;
{ cql_exec("select x->getEdges from x in Face where x->name == \"
name\\" ",results[], ");
if(#results[] == 0)
  return;
if(#results[] != 1){
  error("Error: Multiple faces designated as cross-section\n")
  return;
}

// machined length
cql_exec("select x->getLength from x in results[0] ",dmy[], ");
for(i=0; i<#dmy[]; i++)
machined_length += dmy[i];
printf("Total machined length = %f\n",machined_length);

// mean diameter
set_size(dmy[], 0);
cql_exec("select y(faceType, getRadius, definingDirection)" | 
" from x in Face," | 
" y in Face " | 
" where x->name == \"name\" and for z in y->getEdges " | 
" any (z in x->getEdges)\", dmy[], ");
for(i=0; i<#dmy[]; i++){
  split(dmy[i], " ", vars[]);
  if(vars[0] != 22 && is_axial(vars[2]) == 0){
    // printf("Selected radii %f\n",vars[1]);
    num_dias++;
    mean_dia+=vars[1]*2;
  }
}
if(num_dias != 0){
  mean_dia/=num_dias;
  printf("Mean diameter: %f\n",mean_dia);
}
}

// determines if a direction is axial (along the x-axis)
template
is_axial(direction)
var i; j; k;
{
    gsub("[]","", direction);
    splitd(direction,",", i, j, k);
    if((i == 1.0 || i == -1.0) && is_zero(j) && is_zero(k))
        return 0;
    return -1;
}

// checks if a float in exponential notation is close to zero.
template
    is_zero(f)
    var exponent;
    { return f < 1e-10; }