Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2008-04-17

Exploring the Common Design Space of Dissimilar Assembly Parameterizations for Interdisciplinary Design

Brady M. Larson
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Mechanical Engineering Commons

BYU ScholarsArchive Citation

Larson, Brady M., "Exploring the Common Design Space of Dissimilar Assembly Parameterizations for Interdisciplinary Design" (2008). *All Theses and Dissertations*. 1696.
https://scholarsarchive.byu.edu/etd/1696

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

EXPLORING THE COMMON DESIGN SPACE OF DISSIMILAR ASSEMBLY PARAMETERIZATIONS FOR INTERDISCIPLINARY DESIGN

by

Brady Marc Larson

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering

Brigham Young University

August 2008

Copyright © 2008 Brady Marc Larson

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Brady Marc Larson

This thesis has been read by each member of the following graduate committee and by majority vote has been found satisfactory.

Date	C. Greg Jensen, Chair
Date	Jordan J. Cox
Date	Spencer P. Magleby

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Brady Marc Larson in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative material including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

______________________ ______________________________

Date

C. Greg Jensen
Chair, Graduate Committee

Accepted for the Department

____

Matthew R. Jones
Graduate Coordinator

Accepted for the College

____

Alan R. Parkinson
Dean, Ira A. Fulton College of Engineering and Technology

ABSTRACT

EXPLORING THE COMMON DESIGN SPACE OF DISSIMILAR ASSEMBLY PARAMETERIZATIONS FOR INTERDISCIPLINARY DESIGN

Brady Marc Larson

Department of Mechanical Engineering

Master of Science

The use of parametric CAD models in engineering, analysis, and optimization has greatly enhanced the effectiveness and efficiency of the product development process. Parametric models provide an attractive avenue for expansive design exploration. There still exists, however, a great challenge for products requiring design input from multiple disciplines.

The collaboration of engineering disciplines can be hampered by many factors including: competing design objectives, communication of design changes, the use of different design and analysis software, and different geometry definitions. These obstacles become compounded when developing products at the assembly level. The use of solid parametric assembly models is not readily employed for products developed by

groups from differing engineering disciplines. This is due to the huge cooperative effort required to create, analyze, and iterate on the geometry of the assembly model.

The objective of this thesis is to present a method for separate disciplines to be able to analyze multiple parameterizations of the same CAD assembly to help develop a master parametric assembly, and to define the design space to be explored during analysis and optimization. This is done through a custom application developed using the Application Programming Interface of Siemens' NX CAD software. The custom application allows the user to monitor the affects of manipulating the driving parameters of an assembly by observing user specified geometry, features, or parametric expressions. The application also allows switching from one set of parametric design rules controlling the assembly to another in a matter of seconds. Manipulating and observing key geometry from different parameterizations allows engineering teams to discover the impact of each discipline's driving equations and geometry on another discipline. This will have a profound impact on multidisciplinary design teams in developing a robust parametric assembly, while still taking consideration of the requirements of each discipline.

The collaborative efforts in the development of parametric assembly models used by multidisciplinary design teams are vastly improved through the method and application developed herein. This research will also show both the enhancements that could be made to existing CAD software, as well as the benefits of custom design tool development within the CAD environment.

ACKNOWLEDGMENTS

I would like to express my appreciation for all of those who helped me to pull this thesis together. I'm particularly grateful for my wife Jill for encouraging and supporting me as I worked to finish. I also thank my advisor Dr. Jensen for giving me the motivation and inspiration to explore new territory and grasp onto some of my potential. I will always look back on my experiences as a graduate student as the springboard that boosted me into the real world. I would also like to thank the other students in the ParaCAD research group at BYU for being so great to work with and for helping me formulate my research and computer code. This thesis could not have taken shape without the groundwork and tutoring that I received at their hands. I also need to thank Steve Kramer and many others at Pratt & Whitney who helped me to formulate my thesis topic as well as give valuable feedback regarding my test case CAD geometry. Pratt & Whitney also provided funding to help pay for my costs as a graduate student. Lastly I would like to thank my parents and my in-laws for giving me their support in my desire to seek a graduate degree. I hope that my education will help motivate my children to excel in whatever they choose to study as well.

TABLE OF CONTENTS

LIST OF FIGURES xiii
LIST OF TABLES xvii
CHAPTER 1: Introduction 1
1.1 Problem Statement 3
1.2 Thesis Objective 4
1.3 Delimitation of the Problem 5
1.4 Naming Conventions and Definitions 5
CHAPTER 2: Literature Review 7
2.1 CAD/Assembly Parametrics 8
2.2 Use of the CAD API 15
2.3 Multidisciplinary Optimization and CAD 18
CHAPTER 3: Background 23
3.1 API Programming in NX 23
3.2 Set Theory and Notation 27
CHAPTER 4: Method 31
4.1 Classification of Problems Addressed 32
4.2 Parametric Assemblies 35
4.2.1 Test Case 1: Simple Assembly 38
4.2.2 Test Case 2: Exit Nozzle Assembly 39
4.4 Parametric Data and Set Theory 40
4.3.1 Definition of Data Sets 41
4.3.2 Conflicts, Overlap, and Other Issues 43
4.4 Using the CAD API for the Custom Application 47

4.5 Verification of Valid Geometry 54
CHAPTER 5: Development 59
5.1 Classification and Exploration of Typical Problems 60
5.1.1 Tracking Downstream Effects and Sensitivity 60
5.1.2 Tracking Effects from One Parameterization to Another 61
5.1.3 Finding Common or Conflicting Design Space 64
5.2 Swapping the Assembly Parameterizations 66
5.3 How to Track, Observe, and Output Parametric Manipulation 79
5.4 Uses and Drawbacks of this Method 82
CHAPTER 6: Results 85
6.1 Creation and Validation of Parametric Assembly Models 85
6.2 Successful Swapping of Assembly Parameterizations 86
6.3 The Application Interface Design 89
6.4 How the Effects of Parametric Manipulation Were Observed 92
6.5 Presenting the Observation Data for Review 95
CHAPTER 7: Conclusions / Recommendations 99
7.1 Future Work 102
Appendix A: Example Expression Files 105
Appendix B: Component.cpp source file 109
Appendix C: dSpace.cpp source file 115

LIST OF FIGURES

Figure 2-1 Radius centered parameterization 13
Figure 2-2 Clearance centered parameterization 13
Figure 2-3 Multiple parameterizations in CAD models 14
Figure 3-1 A set of CAD parameter expressions 28
Figure 3-2 Venn diagrams showing set operations 30
Figure 4-1 Model control from multiple disciplines 36
Figure 4-2 Modeling considerations for parts and assemblies 37
Figure 4-3 Exit nozzle subassembly from a Pratt & Whitney jet engine 40
Figure 4-4 Definition of sets of parametric data 41
Figure 4-5 Overlap of sets 42
Figure 4-6 Neutral parameter set used by each discipline to control the CAD geometry 44
Figure 4-7 Invalid union of sets with identical elements 46
Figure 4-8 Constructor callback code for initializing the custom application 49
Figure 4-9 Update failure menu 53
Figure 4-10 Switching an assembly component to the working model 54
Figure 4-11 Launching the system interference analysis 57
Figure 5-1 Example output from "Aero" parameterization 64
Figure 5-2 Code used to export expression files from the CAD assembly 69
Figure 5-3 Code used to import expression files into the CAD assembly 71
Figure 5-4 The custom function “DerefExp”, which neutralizes the expressions in the active model 72
Figure 5-5 The “RewriteExp” custom function which neutralizes the parametric expression 74
Figure 5-6 User interface with values specified 80
Figure 5-7 Simple output file comparing parameters 81
Figure 6-1 Example of geometry changes upon execution of the program 88

Figure 6-2 Assembly parameterization GUI 90
Figure 6-3 Second tab of the custom GUI 92
Figure 6-4 Code fragment for choosing an expression to observe 93
Figure 6-5 Code Fragment for the “get PartExps” Function 94
Figure 6-6 Example Output File 97

LIST OF TABLES

Table 3-1 Set theory symbols used in this thesis 29
Table 4-1 Example parametric expressions 43

CHAPTER 1: INTRODUCTION

Parametric CAD packages have greatly enhanced both the speed and efficiency of the design cycle for new products. These software packages have allowed engineers and designers to more effectively create models which reflect their design intent. One of the major benefits of carefully planned parametric models is the ability to quickly and reliably change their size, shape and topology for analysis and optimization scenarios. Research in this area has shown that the use of robust parametric CAD models in Multidisciplinary Optimization (MDO) loops can dramatically reduce the product design cycle. There are some challenges, however, in attempting to use parametric assembly models for interdisciplinary design situations. To date any research involving assembly level optimization has used very simplified representations of the assembly, often reduced to a two dimensional sketch. Thus the advantages of using a 3-D parametric assembly model for MDO have largely been unexplored due to model complexity, computing capacity, and the large collaborative effort involved.

One of the challenges with parametric CAD models lies in the planning of the rules which will control the way the geometry is configured. In assembly modeling this scheme should reflect the design intent and establish a hierarchy of components and parameters. The hierarchy places certain components and parameters as "drivers" of the system, while forcing any corresponding components and parameters to be dependent and

therefore less likely to need direct manipulation. This one way characteristic of parametric schemes is often in conflict with traditional methods for analyzing and optimizing an assembly.

Assembly level design of complex mechanical and other systems involves multiple and competing criteria. The point of MDO analysis is to drive the design toward a desired balance of the performance and design objectives. If each of the components is at its optimal configuration for its specific function it is not likely that the system will be optimal, or even physically plausible. The objectives determine how the parameter scheme will be set up to "flex" the models within the ranges to be explored. Problems arise when competing objectives would benefit from having a very different parametric schema to manage their behavior. It is also the case that the unidirectional nature of CAD parametric expressions and relationships do not allow for various parameter schemes to be easily represented or iterated on in a single assembly model.

This thesis will propose a method to easily and quickly evaluate the common geometry, features, or parameters of completely separate assembly parameterizations. The proposed method will enable engineers from multiple disciplines to see the affects that each parameterization will have on key geometry when manipulating the driving parameters. This is made possible by enabling the designer to manipulate the assembly at both high and low levels of the assembly tree, and by automated switching from one parameterization to a completely different one. This process could have a significant impact on the ability of interdisciplinary teams to collaboratively develop a master assembly model. The method will essentially enable the designers to discover the available design space which can be explored as the assembly design is fine tuned.

1.1 Problem Statement

The use of parametric CAD models in multidisciplinary design presents several challenges. The first challenge is to create an appropriate parametric model that will not become invalid or unrealistic during optimization. This is true at the part level, and even more so with assemblies. Many researchers have addressed the issues surrounding parametric modeling, model failure, and the complicated process of developing a robust model. It has been shown that the implementation of a well thought out parametric scheme can greatly enhance the effectiveness and efficiency of optimization using CAD models (Srinivasan et al. 2000).

Parametric assemblies are commonly created in industry, but are seldom used in detailed optimization and exploration of configurations for several reasons. One reason is the computing power and time necessary to perform such tasks. This is quickly becoming less of an issue with the increased capacity and availability of high-end supercomputing and clustering facilities. Another challenge to overcome is involving all of the different disciplines which have a say in the design configuration of the assembly and its components. Designers and engineers from each discipline want to explore the sensitivities and impact that changing the geometry will have on the performance benchmarks and requirements for their discipline. The rules that determine how the geometry flexes and which dimensions or parameters drive that change are likely to be different for each discipline. The common practices which attempt to address or avoid this problem include developing a master model containing a compromise on parameterization, or exploring the design space as separate disciplines and then compromising on a final configuration.

1.2 Thesis Objective

As discussed above, reparameterization of assemblies is virtually impossible through the use of current interactive CAD tools. The objective of this research is to develop a method to allow a complete change of parameterizations controlling a CAD model at the assembly level. The feasibility and implementation issues surrounding the creation and use of this method will be thoroughly discussed. The method will be demonstrated through a custom program using C/C++ as within the Application Programming Interface of NX. The functionality made available through this program would enable designers from different disciplines to control the same master geometry using different, even conflicting sets of rules and parameters. Some discussion will also be given regarding the limitations in the current CAD systems to be overcome and/or addressed by this research. Specifically, the questions to be answered by this research will include:

1. What are the types of parametric assembly problems that are typically encountered that will be addressed by this research?
2. How can multiple parametric schemes be seamlessly "swapped" to change the driving parameters of the system?
3. How can the effects of manipulating driving parameters from multiple parameterizations be observed by a concurrent engineering team?
4. How should these effects be presented for comparison and review?
5. What are the uses and drawbacks of this method in design exploration?

1.3 Delimitation of the Problem

This work will include the development and implementation of a method for exploration and observation of multiple parametric assemblies. This will be accomplished by means of a custom application. Parameterizations for test case assemblies will be created and tested. The application of this method in design exploration and collaboration between disciplines will be conceptually discussed. The specific strategies involved in determining a final master assembly model will not be discussed in detail. Also, the steps that would be needed to incorporate the operations performed by the custom application into an optimization loop will be left for future discussion.

1.4 Naming Conventions and Definitions

The purpose of this section is to explain the different naming conventions and fonts that are used in this thesis. Words in **`Bold Courier font`** are used to indicate a C++ class name, or code fragments and examples.

When discussing CAD data, the terms parameter, rule, expression, and variable are all used to describe the method of controlling and assigning the values associated with CAD geometry. These terms have the same meaning for the purposes of this thesis, but the term “parameter” will be used in place of the other terms listed for consistency.

CHAPTER 2: LITERATURE REVIEW

For those readers who are well read and comfortable with the topics of Parametric Assemblies, customization of CAD via API programming, or MDO using a CAD-centric approach, the author encourages you to proceed to Chapter 3. The following sections will provide the basis and frame the problems relating to the management of multiple parameter schemes for optimization of CAD assemblies. The following topics will be reviewed and discussed:

- CAD/Assembly Parametrics (Section 2.1)
    - Robust Parameterizations
    - Managing Parameters and Updating Part Models
    - Inter-part Relationships and Mating Conditions
    - Multiple Parameterizations
- Use of the CAD API (Section 2.2)
    - Extending the System Capability/Compatibility
    - Building Custom Engineering Applications
- Multidisciplinary Optimization and CAD (Section 2.3)
    - Multidisciplinary Design Optimization
    - CAD Centric Optimization

2.1 CAD/Assembly Parametrics

Parametric Technology Corporation changed the way solid modeling was approached with the introduction of their Pro/ENGINEER CAD system in the 1980's (Hoffman and Kim 2001). Parametric modeling, as exemplified through Pro/ENGINEER, allowed a more strategic method to be applied to the dimensioning and modification of geometry. Models could now be easily and quickly modified, enabling a more design and engineering friendly development platform. Shapiro and Vossler (1995) stated that one of the most important practical uses of parametric solid modeling today is creating and maintaining an "electronic master" of geometric data that can be quickly modified to accommodate required engineering changes.

A parametric master model is usually viewed as a repository of knowledge and data of which the CAD model is a subset, or client (Hoffman and Joan-Arinyo 1998). This structure provides a means for all pertinent models of the system, not just the CAD model, to be updated upon the change of any of the master parameters. Although the realm of parametrics in engineering and design has now taken shape at such a high level, the focus for this thesis will be placed on the rules and parameters which control the variation of the CAD model only. Parametric modeling involves both strategic planning and careful implementation to maintain a robust model. This thesis will introduce to the reader the challenge of involving multiple parameter schemes in the same assembly. This challenge will require a new perspective and a different approach to the planning and parameterization of the models in the assembly in order to maintain a robust assembly model.

2.1.1 Robust Parameterizations

In order for a parametric model to be deemed robust it should be able to maintain a feasible, valid, and physically meaningful configuration while having its controlling parameters varied within a desired range. Hoffman and Kim (2001) have recognized that proposing a general solution for generating valid CAD models is a hard problem. Making a model "unbreakable" can result in a very limited flexible range, while a lack of robust parametrics leads to unwanted or invalid geometry. Srinivasan et al. (2001) showed, however, that careful planning of a parametric scheme can significantly reduce generation of invalid models. Thus the planning of the governing parametrics is mainly left to the experience and skill of the designer.

According to Bidarra (2002) "an ideal product modeling system should support both part modeling and assembly modeling, instead of just either of them as is the case in most current CAD systems." The crossover from part modeling to assembly modeling requires the used of more high level parametrics, meaning that the planning of the parameter scheme must consider the relationships that exist between each part in the assembly. The creation of robust, parametric parts does not guarantee a robust assembly.

The quality of the parametric scheme, and therefore the robust nature of the model, is judged at the most basic level by the ability of the model to regenerate or update itself reliably upon modification of parameter values. The types of update failures are diverse, but are often related to the intrinsic nature of the chosen schema for parameterization of the model (Hoffmann and Kim 2001). A poorly planned parameter scheme does not account for possible intersection, overlap, or other situations which result in invalid geometry or an over or under-constrained model. Whether or not an

update is successful is determined by the CAD system through solving a system of equations which determine if the constraints and relationships governing the creation, configuration, assembly, etc. of the geometry are satisfied. Therefore, when planning and creating a parametric model, it is advantageous for the designer to have a working knowledge of the system of constraints to be evaluated by the software upon a model update.

2.1.2 Managing Parameters and Updating Part Models

The use of parametric modeling strategies at the assembly level requires an even more careful, well thought out strategy, to achieve a successful robust model (Srinivasen et al., 2002). The major considerations that arise are inter-part relationships and potential interference between parts or components. If features or dimensions from one component are to affect, or be reflected in another component, a relationship or rule is set up to control the dependency; this is deemed an inter-part relationship. The other main method for controlling part to part association in an assembly is through the use of mating conditions. Mating conditions are used when assembling components to each other to constrain the special relationships and orientation of each component relative to the assembly. Examples of mating conditions are aligning or mating two faces, assigning an offset distance between two components, and enforcing a tangency constraint between a curved surface and another feature. This association between part models and features is the basis for successful integrated product development (Xu, Wang 2002). The modeling methods that employ these types of associative practices are discussed below.

These types of situations also require a new approach to updating the assembly when changes are propagated. If a component is not fully loaded in the assembly, the changes in another component which affect the partially loaded component will not immediately be reflected. The CAD system keeps track of these changes, and upon loading any affected components the geometry is updated.

2.1.3 Assembly Modeling Methods

There are three approaches to assembly modeling: the “Bottom Up”, the “Top Down”, or a combination of the two. Each method has its advantages, and all three can still involve robust parametric control of the geometry (Zeid 2005). The Bottom Up approach is better suited to smaller assemblies and is considered the most traditional and logical approach to assembly design. This process begins with the creation of a blank assembly model into which each of the previously modeled components will be imported. The first part assembled is the base or host part on top of which the other parts will be mated. This process is less likely to involve an intricate set of inter-part relationships since the assembly considerations take place after the parts have been modeled.

Lee (1999) stated that probably the greatest use of assembly design capabilities are in the automotive and aerospace industries. Top Down assembly design is a good approach to take for such large assemblies. This process starts with a more abstract approach by creating a “sketch” of the entire assembly model. This sketch, or perhaps several sketches, will serve as a sort of “skeleton” for the detailed components of each subassembly or subsystem. The most important part of Top Down design is space planning (Zeid, 2005). Using this approach allows project managers to coordinate the

layout of the assembly throughout the process. Three dimensional components may even be parametrically tied to the skeleton sketch such that they update with any changes at the top level. Thus the top level sketch may contain the master parameters for determining the geometric or other properties for three dimensional details. Thus each of the components of the assembly is built in context or concurrently with the surrounding components such that the inter-part relationships are developed as the parts are being constructed.

Kim Youngjun made an important observation regarding CAD assembly structures as it relates to this thesis. He stated that when attempting to remove a part from the assembly hierarchy, the user needs to unlink all the relationships associated with it. This process is done manually and could take a significant amount of time (Youngjun, 2003). These same relationships come into consideration when attempting the types of data and model manipulation discussed in this thesis.

The assembly to be used as a test case for this research will be created using the more traditional bottom up design method. The method developed for the exchange of parameterizations by this research however could be applied to an assembly created by any assembly method. This is possible because the method developed here deals only with the underlying parameter sets and rules used by the model. The types of rules, or methods used to create them, does not affect the data transfer associated with this research. The constraint systems involved in the data transfer to be performed by the custom application are solved internally by the CAD system irregardless to the structure of the parametric expressions.

2.1.4 Multiple Parameterizations

Bettig and Shah (2001) have illustrated that parametric models can have different dimensional schemes depending on the intent of the design. The figures below show two ways of parameterizing the same geometry. Figure 2-1 shows design intent centered on the importance of the radius of the cutout. Figure 2-2 shows a design intent emphasizing the clearance to be maintained between the edge of the part and the cutout. Although this illustrates the likelihood of different parameterizations being needed for the same model, Bettig and Shah made no attempt to integrate the parameterizations into a single model.

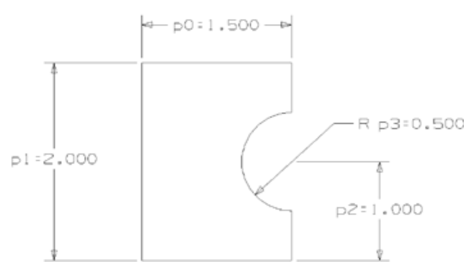


Figure 2-1 Radius centered parameterization

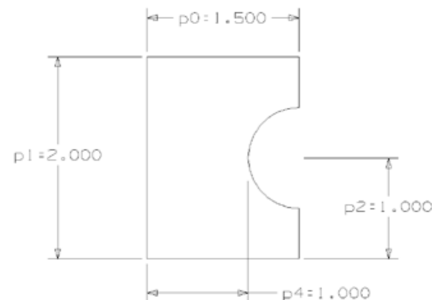


Figure 2-2 Clearance centered parameterization

Many others have discussed issues such as the existence of different viewpoints of a model, but such studies usually involve the representation of the same geometry using differing design and software packages.

Delap (2006) is apparently the first person to investigate the idea of including multiple parameterizations in the same CAD model. In his research, Delap illustrates a method for multiple users to be able to input different parameterizations to control the

same model as illustrated in Figure 2-3. A two-dimensional model of a gas turbine engine flowpath is used to show the usefulness of employing multiple parameterizations. The parameterizations include a module-based and a row-based set. These titles refer to the manner in which the splines used to represent the walls of the engine flowpath are drawn and controlled. The module-based parameterization allows for a more high level control of engine geometry by letting the user adjust the entry and exit geometry of a high pressure turbine module, for example. The intermediate spline geometry is controlled using interpolating methods. The row-based parameterization allows the user to input adjustments for each row of blades or vanes in the engine module while the entry and exit locations remain fixed.

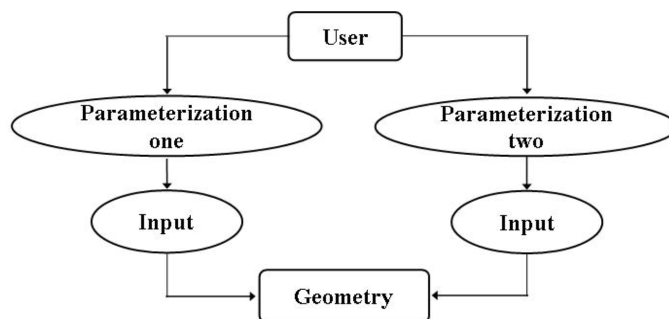


Figure 2-3 Multiple parameterizations in CAD models

The use of multiple parameterizations in Delap's (2003) graduate research allows designers to quickly develop concept geometry for a jet engine flowpath while maintaining both high and low level control. The use of multiple parameterizations was made possible through the CAD API in conjunction with an object oriented programming

structure. The calculations for placing and interpolating spline control points were performed through the C++ code based upon user inputs to a custom user interface.

Although the research in this thesis also involves using multiple parameterizations to control one model, the methods and application discussed follow a different approach, and can be applied in a much broader sense. Delap's research was applied to a single part file on two-dimensional geometry while this research operates on three-dimensional solid components of an assembly model. Further discussion of the differences in these efforts will follow in the method and development chapters of this thesis.

2.2 Use of the CAD API

One of the most important parts of this research will be in the use and implementation of custom menus and operations created within the CAD Application Programming Interface (API). The use of the API of the CAD system has been found to greatly enhance the ability of the engineer to quickly and repeatedly perform custom design tasks (Tucker 2000). Tucker's research involved the development of an API translator which was able to convert API functions and code from one CAD system into functions and code for another system. His research showed a partial solution to a pervasive problem in CAD which is the collaboration and file sharing between groups or companies which use different CAD packages.

In this section, a few examples of using the CAD API to enhance and customize the software to perform design/engineering tasks will be shown. The nature and scale of these examples varies, but the common thread supports the need for the ability to create custom applications such as the one developed for this thesis. Software developers can

learn a great deal about how to enhance CAD programs from these examples. Just as important to learn, is the need to design platforms that can be built upon when needed in an adaptable and user friendly way.

2.2.1 Extending the System Capability/Compatibility

Kim and Han (2004) used the API to access and manipulate data stored in a STEP model that was exported from the CAD system. Their research showed the ability to use API's from both NX and Solidworks to access the STEP file's geometric data to reproduce the geometry in a native CAD model. Similarly this thesis will use the API to perform assembly and data retrieval functions not readily available in the CAD system.

Ardalan (2000) used the API of Solidworks, and Visual Basic to create design program which enabled less advanced CAD users to create complex spacecraft assemblies. This was accomplished by means of a custom GUI where inputs were used to change key parameters on standard parametric components of the spacecraft. The user was guided through the creation and assembly of each component until an entire sophisticated system was complete. This program illustrates the usefulness of custom applications in conjunction with parametric modeling as effective means for aiding in preliminary as well as detailed design and engineering.

2.2.2 Building Custom Engineering Applications

Sometimes the CAD API is used to perform functions or combinations of functions not available in the interactive modeling environment. Much of my research involved combining system functions and customizing functions to perform the desired

tasks. The need for this type of approach is illustrated here, with examples of custom applications used to overcome discovered limitations of commercial CAD software. Myung and Han (2001) recognized that commercial CAD systems cannot handle the parametric design of assemblies. They developed a custom application utilizing the API of the system to aid in the design of a machine tool. This enabled the implementation of parametric design strategies at the assembly level which were unavailable through conventional use of the CAD system.

Use of the CAD API has also been effectively utilized in efforts to enable a more collaborative design environment. Zhou et al. (2003) used the APIs of multiple CAD systems in order to develop a CAD neutral module which will be able to interact with the different commercial CAD packages enabling more collaborative design efforts to take place. Kao and Lin (1998) used the CAD API to pass model information over a LAN as well as the internet as part of a collaborative CAD/CAM system in development.

Rangel and Shah (2002) created an internal integration of CAD and CAM operations through the use of the API creating a single continuous application. Their work is a good illustration of the power of using the API in linking or combining engineering tools to help centralize and unify the design process.

The ParaCAD research group at BYU, under the supervision and direction of Dr. Jensen, has explored the utilization of API programming in many different applications. This research has produced several theses and published articles showing the impact that parametric modeling, coupled with programmatic linking of CAx tools, can have in both industrial and academic efforts. Some of the work to come out of the ParaCAD research group includes preprocessing of rapid prototyping models (Wilson 2004), CAD-centric

CFD analysis with discrete features (King, 2004), and integration of commercial CAD and analysis software for multidisciplinary design optimization (Hogge, 2002). This work continues to be funded by aerospace and automotive industry leaders who are interested in exploring and implementing advanced CAx applications which can give them a competitive edge in engineering and time-to-market efforts.

As illustrated, the API available in some commercial CAD systems has been used in several ways, all of which enable engineers and designers to enhance their ability to design, analyze, and manufacture better products. The use of the API also allows for opportunities to customize products through a combination of parametric design and automated model generation. As parametric CAD systems become more and more embedded with design and engineering knowledge, the use of custom applications will help make a fully integrated design and analysis software more of a reality.

2.3 Multidisciplinary Optimization and CAD

The previous sections have shown the key role that parametric CAD plays in the development and analysis of new designs in both research and industry. This section will discuss the optimization of a design using modern methods and technologies. Through the use of popular optimization strategies, applied to parametric CAD models, engineers are beginning to be able to explore for potential designs in a very meaningful and effective manner.

2.3.1 Multidisciplinary Design Optimization

Alexandrov and Lewis (1999) define Multidisciplinary Design Optimization (MDO) to mean the systematic approach to optimization of complex, coupled engineering systems, where "multidisciplinary" refers to the different aspects that must be included in a design problem. There are basically two approaches to MDO (Cheng and Li 1999). The first is to try to find the optimal solution directly by combining each objective into one weighted objective. This approach limits the choice of the designer since only one optimal solution is produced, and is determined by the weighting of the objective. The second approach produces several optimal designs which are not dominated by any other design. These are the Pareto designs, or the Pareto front of designs. This method allows the designer to choose the weight of the objectives after he has seen the possible optimal solutions.

2.3.2 CAD Centric Optimization

MDO of complex mechanical assemblies is made possible through the successful construction of robust parametric CAD assembly models (Srinivasan et al. 2001). Attempting true multi-objective optimization or distributed optimization of a separable objective is not widely done in MDO at present (Alexandrov and Lewis 1999). There has been some research, at the single part level, that has shown the feasibility of performing true multidisciplinary optimization with parametric CAD models and other CAE tools (Hogge 2002). This was done using a parametric model of a single turbine blade through the use of commercial engineering software by utilizing the APIs and automated functionality in modeling, analysis, and optimization. As mentioned, the focus of

Hogge's work was to show the feasibility of linking analysis and design software using a CAD-centric approach with a parametric solid part model. This thesis expands the CAD centered approach to explore assembly models and how to involve multiple disciplines, while maintaining robust parametric methods.

The point of view, design emphasis, and design approach of each discipline specialist can be quite different. Too often the practice has been for specialists to independently optimize each discipline with limited direct interaction or communication with others (Townsend et al. 1998). Ledermann states that it is very important to have clearly defined interfaces between different disciplines that allow the exchange of reciprocatively required data. It is also noted, however, that the use of common parametric-associative geometry as a basis for analysis is promising in helping to overcome the difficulties encountered in aircraft pre-design (Ledermann et. al. 2005).

Another approach to MDO presupposes that objective functions, rules, constraints, etc. are being posed in a collaborative concurrent engineering environment, where the structures, aerodynamics, manufacturing, etc. disciplines are all working together to achieve an optimal design (Srinivasan et al. 2001). This collaborative effort involves an attempt to develop one parametric CAD assembly model to be used for analysis and optimization across all disciplines involved. During this process there are many compromises or trade offs made early in the development of the parametric scheme of the assembly. Thus there is significant effort put into creating a scheme which will be able to properly represent the design as perceived by each of the disciplines. The resulting model from a process such as this can be fairly robust in searching for an

optimal design, although the search may take place within a limited space due to the compromises made in the parametric setup.

As mentioned above, developing a robust parametric CAD model is crucial to the successful execution of MDO. Failure to properly parameterize the model will most likely result in invalid or unusable geometry during iterations (Srinivasan et al., 2001). This can become a very wasteful problem causing the analysis routines to either produce unusable data, or to crash completely while trying to analyze the model. Since computing time is already a limiting factor when attempting MDO with 3D models, a poorly parameterized model further aggravates this situation.

Sometimes a situation may arise in which the competing objectives are actually more effectively explored using different parametric models. This thesis will demonstrate a method which allows designers from each discipline to create their own parametric schemes which maintain design intent regarding their discipline while also allowing a more expansive search of the design space. This method will avoid the limitations and compromises of using a single parametric model. Because the parametric schemes will be able to be quickly and automatically swapped, a broader range of potential designs covering the fully desired space from each discipline can be explored. Similar to evaluating the designs on the Pareto front, this method will allow for designers to choose the objectives and driving parameters for the model after much of the design space has been explored.

CHAPTER 3: BACKGROUND

This chapter will give the reader background and explanation of two of the more prevalent topics of this thesis: API programming in NX, and Set Theory and Notation. The reader will be guided through the structure and creation of a custom application using the NX API. This will illustrate how problems such as those addressed by this thesis can be more effectively approached through custom applications. A brief explanation of Set Theory and how it is used to help represent the problems of this thesis will also be given. This representation will help the reader understand the relationships and interactions of parametric data and the CAD models using that data.

3.1 API Programming in NX

Some of today's CAD packages have a great advantage in their ability to be customized for specific application in industry and research. One of the most powerful ways to customize the CAD software is through the use of the Application Programming Interface (API). Through the API a completely customized application can be developed to use any combination of functions within the CAD package, as well as the functionality of the programming language itself. The application to be developed for this thesis will utilize the NX API to automatically perform tasks that are unavailable through typical interactive use of the software. An explanation of the interface and capabilities of the NX

API, along with a brief explanation of how a custom application is developed, will be given here.

The NX API is written in the C programming language providing the programmer with the ability to easily implement any C/C++ functionality within the custom application. This allows for custom calculations, file input and output, data manipulation, and other operations to enhance the capability of the custom application. The API library itself contains the following:

- A large set of user callable functions/subroutines that access the NX Graphics Kernel, File Manager, and Database.
- Command procedures to link and run user programs.
- An interactive interface in NX to run those programs.

The programs written with the API can be run in two environments:

- External - these programs are stand alone programs that can run from the operating system, outside of NX, or as a child process spawned from within NX.
- Internal - these programs can only be run from inside of a NX session. These programs are loaded into main memory along side of NX kernel and access routines within NX. One advantage to this is that the executables are much smaller and link much faster. Once an Internal API Program is loaded into memory, it can stay resident for the remainder of the NX session. If you call this program again, it executes without reloading (provided it was not unloaded). Internal API programs work on the current part and automatically modify the part display (UG/Open documentation, Version NX2).

Through the use of the NX API all of the functionality of the interactive program is available. Therefore there is no limit to the type or capability of custom programs utilizing the API. A fully developed custom application is typically desired when there is a complex design process to be performed more than just a few times. Simple custom calculations or operations are better suited for use with macros or small executable files. The ParaCAD laboratory at BYU has been developing custom applications in cooperation with industrial leaders that use product specific design rules to build, manipulate, and optimize parametric models of their products. The use of the API allows both custom control of the models as well as the ability to incorporate or link with other applications such as spreadsheets, in-house code, analysis, and optimization software. These applications are being implemented in industry to speed up the design process and allow opportunities for greater design exploration.

The starting point for the development of a custom application using the NX API is the UI Styler interface of NX. As mentioned above, NX also supplies a customizable interface for creating any windows or menus to be used for the application. The interface created is typically referred to as the Graphical User Interface, or GUI. Whenever the programmer creates any buttons, entry fields, or other menu features, template code is generated to assist in linking the GUI to whatever functionality the programmer wishes to employ. The template functions created while designing the GUI are called “Callbacks”. Custom callbacks can also be added and linked to buttons or other features of the GUI. If specified, a button may launch a new dialog, allowing the program to be layered and structured into a very complex and complete application.

The UI Styler application within NX creates template code for program initialization and entry points. The programmer is then able to add custom code to execute the desired tasks. There are significant resources available in the API programmers guide included in the NX documentation files. The callbacks used to execute the code in the custom application developed for this thesis will use a combination of NX user functions and custom code to manipulate the assembly models, as well as the embedded parameterizations. A generic list of steps used to develop a custom design application via the NX API is shown below.

1. The programmer designs the interface for the application with the desired tasks in mind.
2. The programmer creates the program files and header files containing the code and functions that will be used to execute the program.
3. The programmer "connects" the custom code to the user interface by inserting the custom code into the callback functions related to the GUI.
4. The programmer compiles and tests the application by launching the newly created GUI from an NX session.

Developing custom applications via the use of an API is an endeavor which requires a large investment up front. Generally a program is written with the API if it is to be used repeatedly and contains operations which would be time consuming when performed interactively, or which are unavailable directly in the user interface. Another case in which using the API is preferred is when design specific calculations are used to

create a model, or if the designer is to be stepped through the model creation with the user interface.

The program written in conjunction with this thesis performs a combination of custom functions and API calls to perform operations not available in the interactive NX environment. In this case the program was developed specifically to explore the possibility of enhancing the CAD software's capabilities in an area where limitations became apparent. The layout and execution of these operations will be explained further in the method chapter.

3.2 Set Theory and Notation

Set theory was founded by Georg Cantor, and has been around since the late 1800s. The concept of a set is very simple on the surface. A set is any collection, group, or conglomerate (Hrbacek 1999). Another definition given by Levy (1980) states: A set is a collection of distinct objects, without repetition, and without ordering. The elements of a set are referred to as its members. In this thesis set theory will serve as the method for representing all of the containment, interaction, and conflict of the parameters used in a typical CAD assembly. An example of a set of CAD parameters is shown in Figure 3-1 below.

Set theory could be used in several ways to discuss the entities and operations used in a CAD environment. The geometry of a CAD model is typically represented as a group of features, bodies, and operations which constitute the "tree" or "history" of the model. The features and bodies are made up of sets of lines, curves, and surfaces, each of which has its own defining sets of data such as control points, curvature equations, and

dimensions. Also, whenever a model is modified and updated there is a large set of constraints that the system must check in order to determine if the model is still valid after modification. These constraints and their interaction would appropriately be represented using set theory and notation.

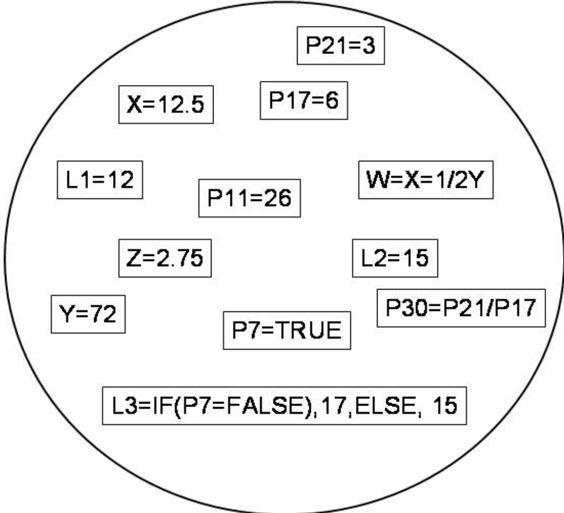


Figure 3-1 A set of CAD parameter expressions

There are certain requirements that an element of a set must satisfy. First, the elements must be well defined to determine unequivocally whether or not any object belongs to the set. Second, the elements of the set must be distinct and no element may appear twice. Third, the order of the elements within the set must be immaterial (Zeid 2005).

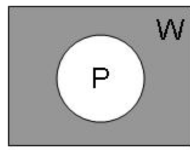
Table 3-1 contains a summary of the symbols from set theory used in this thesis. These symbols represent the operations performed on sets and are similar to mathematical and Boolean operations.

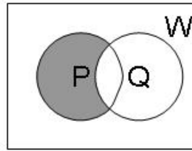
Table 3-1 Set theory symbols used in this thesis

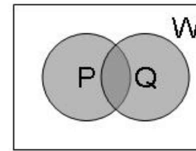
Symbol	Description
$\cup$:	This represents the union of two sets, as in $X \cup Y$
$\cap$:	This represents the intersection of two sets
$\in$:	This symbol is used to show membership in a set, as in $x \in X$
$\notin$:	This symbol excludes an element, declares it as a non-member
$\subset$:	This symbol shows that one set is a subset of another
$\not\subset$:	This symbol declares the set is not a subset of the other
c:	A lower case "c" represents the complement of a set, as in cX

Figure 3-3 shows some Venn diagrams illustrating the typical set operations to be used in this thesis. The grayed areas show the result of the operation stated below the Venn diagram. The diagrams only show the representation of the coexistence of the elements described by each operation. The effect that these operations may have when the elements of the sets are equations and expressions will merit further discussion in the following chapters.

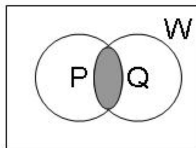
The figure below contains visual representations of the interaction of sets. The set "W" is basically a global set which contains the sets "P" and "Q" and all of their elements. These diagrams show the resulting configuration of elements and sets from each operation. Chapters four and five of this thesis will go into more depth discussing the interaction of the sets and the elements within the sets when such operations are performed.

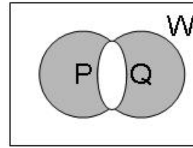






a) Complementation (cP) **b) Difference (P-Q)** **c) Union (P∪Q)**





d) Intersection (P∩Q) **e) Exclusive Union (P⊻Q)**

Figure 3-2 Venn diagrams showing set operations

Conflicts and constrained situations arise when the elements of the sets are mostly equations. The dependencies and references between expressions in a parametric CAD model will most definitely create these types of situations. For the purposes of this thesis, set theory is used to describe the representation and interaction of the parameters controlling the CAD geometry. Because the purpose of this research is to develop a method and an application which allows the exchange of parametric rules and data, the sets of rules and data from each separate part and assembly will need to be discussed both separately and as a group. The possible interactions, conflicts, and overlap of parameters from different parameterizations used for the same assembly model are best discussed in the context of set theory and notation.

CHAPTER 4: METHOD

This chapter contains the method that was initially proposed in the prospectus to this thesis. The method consists of an approach which would enable a designer or engineer to effectively observe the behavior, overlap, and potential conflict involved in incorporating design parameters from multiple disciplines into a master parametric assembly model. Part of this process allows for controlling the interchange of an entire set of parametric rules and expressions for a 3D CAD assembly. This method involves several steps and considerations including: (1) the classification and definition of the expected scenarios that would benefit from the strategies and application described in this thesis, (2) the creation of test assemblies parameterized according to different methods, and the utilization of set notation and theory to represent the interaction and transfer of data, (3) symbolic and mathematical representations of assembly model manipulation, (4) the use of the CAD API and C/C++ programming to enable functionality not available in the interactive CAD environment, (5) the gathering and export of the observed state of key features and geometry for review and comparison. These steps will be described in the sections of this chapter. It is intended that this method could be followed and implemented by other engineers/designers to approach this and other similar problems in parametric design. This method enables parametric design strategies to be considered and implemented in ways that are prohibited by the CAD systems themselves. Some of

the limitations of parametric rules and constraints used in modeling are addressed and overcome through the application of the method found here.

4.1 Classification of Problems Addressed

This section will give a definition for the types of problems encountered when developing parametric assembly models that will be addressed by this thesis. There are a number of specific problems that could be alleviated by applying the methods and utilizing the application developed here. These problems will be broken down into a few general classifications and given definition and explanation.

The problem classifications are as follows:

1. How to track "downstream" effects of complex parameterizations
2. How to discover the effects of changes in one parameterization on key geometry of another discipline's parameterization
3. How to explore the sensitivity of key geometry or key analysis parameters to high level changes or vice versa
4. How to find the common or conflicting design space of parameterizations from different disciplines

4.1.1 Tracking Downstream Effects

In complex assembly parameterizations there may be geometry or parameters that are several steps "downstream" from a driving parameter. This means that there is a series of dependencies between the driving parameter and the affected geometry. These items may still be of interest, or play a key part in the function of the system. The

application developed for this thesis allows the user to quickly review the effect that changing a driving parameter within a specified range will have on any downstream feature within any component of the assembly.

4.1.2 Discovering Change Effects Between Parameterizations

One of the main objectives in developing the custom application for this thesis was to be able to quickly change the parameterization being used to control an assembly model. This functionality coupled with the ability to observe any portion of the model that may be of interest allows for a new level of parametric exploration.

The ability to quickly change the entire scheme of parametric control for the assembly can dramatically enhance the awareness between disciplines during the development of a design. With the capabilities mentioned, the user could easily observe the effect that manipulating the driving parameters from his/her discipline will have on the key geometry for another discipline. Therefore, if communication is passed along between disciplines such as "we need the thickness of this member to remain between .125in. and .187in., the designers from other groups can easily track the effects of their manipulation on the thickness of the geometry of concern.

4.1.3 Observing Sensitivity of Parametric Changes

Observing the sensitivity of parametric changes is similar to the scenario described in section 4.1.1. However, when the user is interested in the sensitivity of certain geometry or expressions, he/she will likely want to plot the values of the affected

geometry while varying several of the driving parameters. In this way, the user can effectively realize the relationship that exists between these features.

This strategy may also be employed between different parameterizations as mentioned in the previous section. This becomes increasingly useful with the complexity of the assembly parameterization. If there are several parametric relationships within the assembly model, it can become very difficult to know the connection or sensitivity of some features when manipulating the driving parameters. The application presented here diminishes these difficulties.

4.1.4 Finding Common or Conflicting Design Space

One of the other key uses for the custom application presented is to more effectively explore the design space between disciplines. By observing the effects of parametric variation as described in section 4.1.2, the design team can begin compiling information allowing them to discover the common or conflicting design space of the parametric assembly.

This is done by compiling the range of variation for geometry of interest as it is affected by modifying the driving parameters from each of the disciplines involved in the design. The overlapping range, or lack thereof, will convey to the common design space that is being used during manipulation of each of the parameterizations. This process could be repeated for as many features as necessary. The designer can also know the range that each of the driving parameters was varied within that produced the common design space.

4.2 Parametric Assemblies

An integral part of the formulation of this thesis is the planning and use of parametric design strategies with assembly models. Test assemblies must be created which will be a representative candidate for the issues addressed by this research. There will be a very simple test assembly which will include parametric relationships that illustrate the concepts laid out here. There will also be an assembly with a system of components which require design considerations from multiple disciplines as shown in Figure 4-1. Each discipline would be able to input its own design rules into the parametric assembly.

The same geometry must be able to be used by designers from each discipline for concept generation, parametric variation, or optimization. Design rules will be implemented in a parametric fashion so as to be able to reliably change and update the model upon variation of the key driving parameters.

For example, if a simple nut-plate were assembled to a bracket, the holes in the nut-plate would need to maintain the appropriate size, position and spacing if any changes were made to the bracket. Similar considerations will be present in the assembly used in this thesis as the relative size and spacing of parts is crucial to the performance of the system.

The method developed here is meant to be used in conjunction with parametric assemblies. The benefits and usability of this method are more apparent when well planned parametric models are used. The scenarios that are best suited for the use of this method involve the interaction of design teams in a collaborative effort to explore design space while maintaining the representation of each team separately. This thesis also

surmises that such efforts could be aided by the use of a custom application such as the one created here. The parametric design strategies used by advanced design teams in industry today are not easily integrated into a single CAD model that could be used for analysis and optimization by these groups. In many large scale applications this would simply be impossible. It is proposed that by enabling multiple parametric schemes to be incorporated and observed in a single CAD model that the integration of analysis and exploration is greatly enhanced. Not only does this strategy bring the design teams into a communicative environment, but it also allows them to work with the exact same geometry. This means that a single parametric model can be adaptable for the uses of different, discrete disciplines.

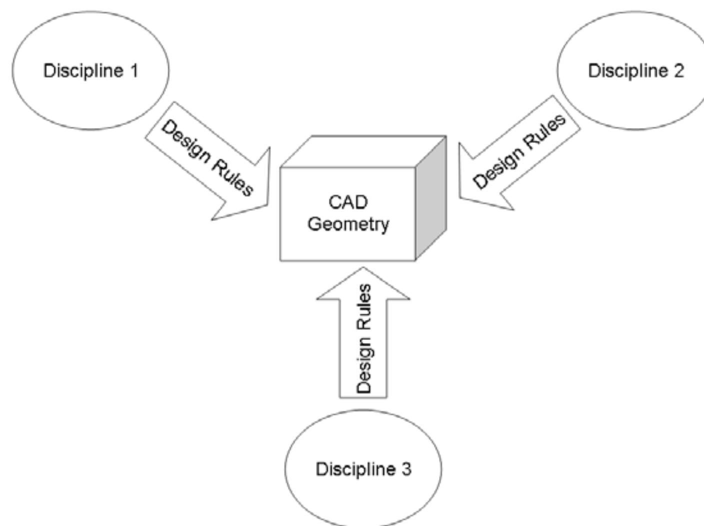


Figure 4-1 Model control from multiple disciplines

Modeling Considerations: Parts vs. Assemblies

Part Model	Assembly Model
• Parametric Rules	• Parametric Rules
• Dependent Features	• Dependent Features
• Geometric Constraints	• Geometric Constraints
• Design Intent	• Design Intent
	• Mating Conditions
	• Inter-part Relationships
	• Spatial Relationships
	• Interference Issues

Figure 4-2 Modeling considerations for parts and assemblies

Significant effort will be placed in the parameterization of the sample assemblies used for this thesis. Since each discipline will need to use the same base geometry, it will be created in such a way that the different disciplines will be able to easily customize the way their design rules affect the geometry. This means that the generic form of the model must not contain too many assumptions about design intent. It will therefore contain a large set of independent controlling dimensions and parameters. The base model would essentially be un-parametric according to the definition of parametric models described and referred to in this thesis. Each dimension, parameter, or expression will control only the entity to which it is directly associated.

The idea is that the design teams from each discipline would be handed a template to work with. When the parameterizations developed by each discipline are to be applied to the generic model, it will be as if they are simply filling in the blanks from the template to create a fully parametric model specific to their design intent. Much like

handing two sculptors an identical lump of clay, these groups could create very different configurations from the same base model. The method laid out here will describe how these configurations can be quickly and automatically applied to the same model in a way not allowed by the parametric CAD system. In conjunction with switching between parameterizations, the users will also be able to observe key geometry or features while manipulating the driving parameters from each parameterization.

The parameterizations of the assembly from each discipline will contain rules and relationships which would not be able to exist in the same model. This is due to several factors including inter-part relationships, dependant parameters, and geometric incompatibilities that will most certainly exist. These restrictions and limitations will be described in the next section by means of Set Theory and notation.

The parameterizations for use in demonstrating the method of this thesis will include a structural set, an aerodynamic set, and a general parameterization with emphasis on the overall dimensions driving the system. Each of these parameterizations will be carefully planned and created in order to obtain an effective and robust model able to be flexed within a reasonable range of values. The driving parameters for each discipline will be varied according to their appropriate limits and the model will respond by changing and updating in a reliable and valid manner.

4.2.1 Test Case 1: Simple Assembly

The first test case assembly will involve only simple geometry. This assembly will serve to validate the proper function of the custom application. It will be used to show the effects of incorporating different parameterizations within the same model, and

to give simplified examples of typical scenarios that are expected to be encountered when implementing the methods discussed in this thesis. Once the functionality of the custom application has been verified, a more complex and industry typical assembly will be used as a test case. This is presented in the next section.

4.2.2 Test Case 2: Exit Nozzle Assembly

The test case assembly to be used will be an exit nozzle subassembly that will be loosely based on a typical jet engine configuration like the one shown in figure 4-3. The components will include a tailcone, strut, support bracket, and a mixer. This section of the jet engine is designed to allow the exit flow from the main engine and the bypass flow to mix in order to reduce excessive noise. Noise from the exhaust system occurs due to shearing of the high temperature, high velocity gases coming from the turbine with the low temperature, low velocity air from the engine's bypass flow. The design of this geometry involves considerable analysis from aerodynamic and structural/vibrational engineers. Each of the components in this subassembly has a significant contribution to the effectiveness and efficiency of the overall system.

Parameterization of the two test cases will be determined and generated by the author. This may not necessarily be representative of the most effective or appropriate parameterization of this geometry, but should illustrate the concept and method sufficiently. The method outlined in this thesis is intended to be used by a team of experienced engineers working on the same geometry, and should involve their careful planning and judgment in parameterizing the model. My parameterization of the exit nozzle model will, however, effectively show the usefulness and validity of the method

presented and program to be written. This model will contain rules and parameters that exhibit the inter-part relationships and dependencies previously discussed which make any swapping of parameterizations cumbersome through interactive methods.

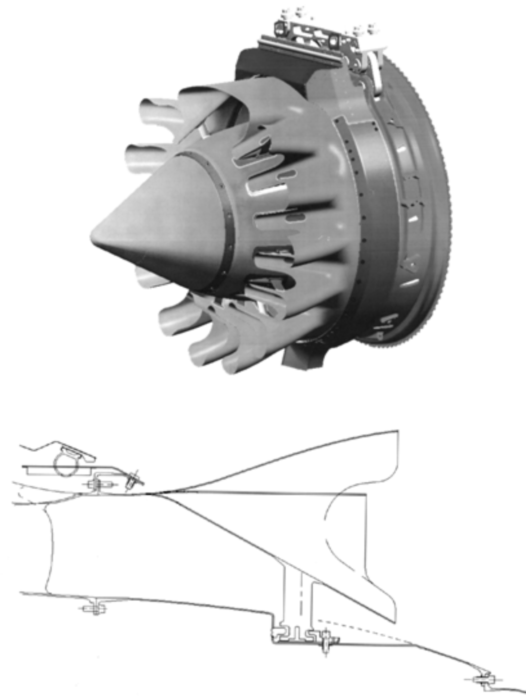


Figure 4-3 Exit nozzle subassembly from a Pratt & Whitney jet engine

4.4 Parametric Data and Set Theory

The problem addressed by this thesis lies mainly in the interaction of multiple sets of parametric data used to control a single CAD assembly. This section will discuss how the interchange of parametric rules and data will be approached from a theoretical point of view. Through the use of Set Theory and/or mathematical notation, a method will be

outlined explaining the various scenarios that will possibly be encountered, and how to avoid or work through any difficulties.

4.3.1 Definition of Data Sets

The sets of rules and expressions used in the different parameterizations of the assembly will be represented in a symbolic form in order to more easily represent the nature of their interaction. The definitions of these sets are given in Figure 4-4:

P = The global set of all parameters associated with the assembly
M = The set of parameters currently being used in the assembly
S = The set of parameters determined by the Structures discipline
A = The set of parameters determined by the Aerodynamics discipline
G = The set of parameters governed by assembly geometric properties
N = The set of independent parameters within the base model

Figure 4-4 Definition of sets of parametric data

At any one time the parameters controlling the model are represented by the set *M*. This set will contain at least one subset consisting of one of the sets defined above as well as possibly containing other parameters from the global set *P*. The sets *S, A,* and *G* contain all of the rules and expressions used to control the model as dictated by each of the disciplines. Each of the sets *S, A,* and *G* are fully capable of controlling the parametric model separately, but cannot exist simultaneously as subsets of the set *M*. This will be discussed in more detail later in this chapter.

Many of the parameters in the model will remain unchanged from one set to another and are thus identical. In this sense there is some overlap in the definition of

each of the subsets of the global set *P* as shown in Figure 4-5 below. Each of the elements in the sets *S, A,* and *G* are either derived from the elements of the set *N*, or are created as controlling parameters, calculations, or other representations within the model. In other words, each of the design teams would be given a model containing the independent and essentially un-parametric expressions used to create the geometry (the set *N*). From those expressions, the teams would generate the desired relationships, rules, and driving parameters for the system. Therefore, any overlap or existence of identical elements between each of the sets *S, A,* and *G*, represents elements of the original set *N*.

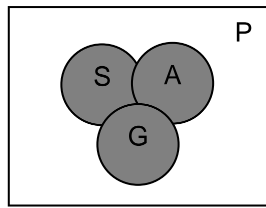


Figure 4-5 Overlap of sets

Along with overlap, elements of a set may have been created as either a driving parameter, or a reference variable holding a calculated value. These are parameters specific to a set, and have not been derived or modified from the original set *N*. Thus, the parameter sets *S, A,* and *G,* used to control the model, are not subsets of the base set *N*. An example of this would be if a parameter called Max_Load was created in the structural set. This parameter would have an associated value which would be calculated from other parameter values. This example is shown in Figure 4-6.

Table 4-1 Example parametric expressions

Max_Load = (Avg_Velocity*Profile)*Safety_Factor

Avg_Velocity = (Inlet_Velocity+ExitVelocity)/2

Safety_Factor = 1.35

Inlet_Velocity = 240

. . . . .

Max_Load is not directly associated with any geometry or features of the model, but serves as a reference for the designers to check against as the geometry is varied.

Another example might be a parameter called Inlet_Velocity from the aerodynamic set. This parameter is likely to be varied in design exploration in a different manner than the Max_Load variable mentioned earlier. In other words, the Inlet_Velocity variable would be used to drive the variation of other variables or parameters in the assembly. The parameters Max_Load, and Inlet_Velocity are examples of variables created specifically for a certain parameterization to be referenced within the assembly, but which are meaningless and ineffective outside of that parameterization.

4.3.2 Conflicts, Overlap, and Other Issues

Each of the fully parametric sets of data will contain an intertwining network of rules and relationships between elements of the set which make the geometry behave appropriately when the controlling parameters are varied. Because each of these sets is derived from the original set *N*, most of the parameters are directly related to a geometric entity. Thus each set is created to literally control the same geometry. When the

parametric sets are developed, however, they are kept separate and independent from each other as shown in Figure 4-7. Each discipline or design group does not, and should not have to, worry about what the other disciplines might be doing to control the CAD model.

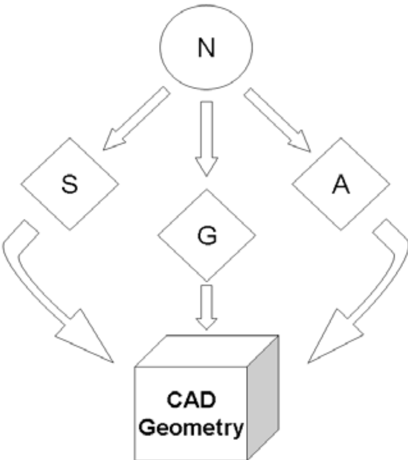


Figure 4-6 Neutral parameter set used by each discipline to control the CAD geometry

The only thing that needs to be maintained between groups is that the geometry itself is not altered. In other words, the features and geometric entities of the model cannot be deleted, or otherwise redefined in such a way that requires a new or altered set of the controlling parameters for that feature. If the base model were a lump of clay handed to each design group, they would not be allowed to take pieces away from the lump, add to it, or change its composition. They would only be allowed to push, pull, and deform the clay according to their intended design to create their model.

The parametric data sets are simply a new set of instructions for the geometry to follow. With multiple assembly parameterizations there would be multiple sets of instructions for the same model. It is this concept which best illustrates the problem addressed by this thesis. How can a single parametric CAD assembly model be able to follow multiple sets of instructions? In particular, how can conflicting sets of instructions be incorporated into the same model? The off-the-shelf CAD system does not, and cannot, allow for conflicting relationships or constraints to exist within the model. The solver for the system will not allow what would essentially be an over constrained system of equations. This is one reason that multiple parameterizations in a single model are a difficult problem. A more fundamental reason making multiple parameterizations difficult is that a single variable, or parameter, cannot easily have two values associated with it. Although there are provisions in the CAD systems which allow conditional statements to control the value of a parameter, the use is limited and cumbersome.

The issue of conflicting instructions is the center of the argument given for the method proposed here for utilizing multiple parameterizations. If conflicting rules are imported into the model, the software will not allow the user to proceed because of the over constrained system mentioned earlier. In order to overcome this issue, the system must be satisfied. For example:

```
Length = 2*Height
Length = Width/3
```

This is a simple case where one designer might wish to associate the length of the part to the height, while another might link the length to the width. There are two things that are wrong with the conflicting equations shown. First of all, there are two variables called “Length”. This will not be allowed by the software for the simple reason that it

cannot distinguish between the two. The other part of the problem is that the system does not know which value to assign to the parameter “Length” because of the conflicting instructions on the right hand side of the equation. A union of the sets is attempted when the user tries to update the model. This union will fail because of the inability of the system to decide which expression is correct for the parameterization. In terms of Set Theory this situation would violate one of the basic definitions of a set, which states, “A set is a collection of distinct objects, without repetition, and without ordering” (Levy, 1980). The resulting set shown in figure 4-8 contains objects that are not distinct and are repeated.

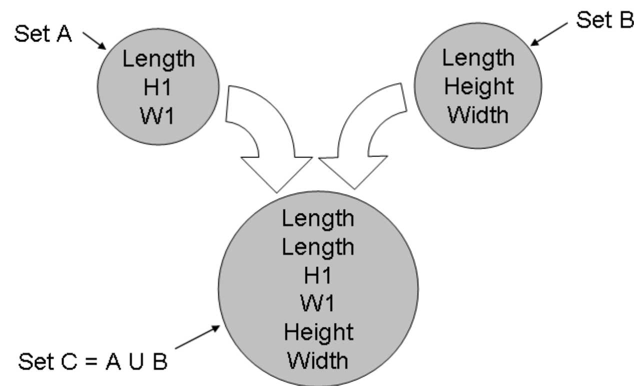


Figure 4-7 Invalid union of sets with identical elements

In terms of set theory, the above scenario occurs when two sets of parametric data are to be imported into the same model. Both sets contain a parameter called “Length”, but the dependency of the parameter upon other parameters or equations differs from set

to set. Therefore when the two sets are brought into the same model there is an intersection.

4.4 Using the CAD API for the Custom Application

The API of NX will be used to create a custom program enabling the interchange of parameterizations and observation of key geometry during model manipulation for the test case assemblies. The development of the program includes the design and creation of a Graphical User Interface (GUI) to gather the required input data and execute any functions or processes necessary. Accompanying the GUI is the C/C++ code that will execute the API commands and process the rules and expressions for each parameterization.

The process for observing the behaviors of different assembly parameterizations and exchanging the assembly parameterizations will proceed as follows:

1. With an assembly model open, the API program interrogates the assembly model upon being launched to gather component information and file locations
2. The user chooses model geometry, features, or expressions to observe and an expression to vary
3. When the manipulation is executed within a specified range, relative output is written to a file for later use
4. The user locates a new set of parameter expression files to be imported and replace the current parameterization, changing the behavior and interaction of the assembly components and geometry

5. The user again chooses model geometry, features, or expressions to observe (probably the same as was chosen earlier) and an expression to vary
6. During manipulation, the output is again written to the same file to be used for comparison

4.4.1 Assembly Interrogation/ Application Initialization

Step one of the process is accomplished upon launching the program as part of the "Constructor Callback". This is a function that is automatically executed when the application is started up and is used to perform any initialization necessary when using a custom application. It may be used to set default values in entry fields, to set the sensitivity or availability of buttons or other features on the GUI. In this case the code within the constructor executes a series of functions used to gather information about the assembly such as component names, CAD system identifiers, and file locations. This function is shown in Figure 4-9.

Essentially, the code below is creating the data structure to store all of the information about the assembly model, including file names and locations, parameters, and other data to be used during execution of the program. The "root" part in the assembly is the part onto which all of the other components are assembled. Once the root part is determined, the program stores the names of each component in an assembly structure using a C++ class data structure.

```
int EXP_constructor_cb ( int dialog_id, void *
client_data,      UF_STYLER_item_value_type_p_t
callback_data)
{
     if ( UF_initialize() != 0)
          return ( UF_UI_CB_CONTINUE_DIALOG );

      char *file_name = new char[132];
      char *file_path = new char[132];

      Component* root = new Component();
      root_part = UF_PART_ask_display_part ( );
      root->SetPartTag(root_part);
      UF_PART_ask_part_name(root_part, root_name);
      FilestringDecomp(root_name,file_name, file_path);
      root->SetName(file_name);
      Assembly.push_back(root);

      root_occ = UF_ASSEM_ask_root_part_occ(root_part);

      Component* comp;
      int num_parts =
      UF_ASSEM_ask_part_occ_children(root_occ,
      &child_part_occs);

      for(int i=0; i<num_parts; i++)
        {
         comp = new Component();
         comp->SetPartTag
         (UF_ASSEM_ask_prototype_of_occ
         (child_part_occs[i]));
         char name[132];
         UF_PART_ask_part_name(comp->GetPartTag(),
name);
         FilestringDecomp(name, file_name, file_path);
         comp->SetName(file_name);
         Assembly.push_back(comp);
       }
```

Figure 4-8 Constructor callback code for initializing the custom application

4.4.2 Observing And Manipulating The Assembly Model

The key component in the custom application developed for this thesis will be the ability to efficiently observe the effects of parametric manipulation from multiple schemes on common geometry. The user will be able to choose a feature, or expression within a component of the assembly to observe, while varying a controlling parameter from another component in the assembly. Then within a very short time the user would be able to observe that same feature or expression as it is affected by the manipulation of a driving parameter from an entirely different controlling parameterization. In this way, one can observe the commonality or differences in the viewpoints of different disciplines as it relates to key elements of the assembly.

The user would specify whether a feature or a single expression is going to be tracked during the parameter variation, and then choose the feature or expression from those available within the currently active component in the assembly. Next the parameter to vary would be chosen, within another component of the assembly if desired. The limits for variation would be specified, along with the number of increments to be evaluated between the limits. Once all of these items have been determined, the user executes the operation to initiate manipulation. The model is continuously updated during the parameter variation so that any visible changes in the model are easily viewed. The model is refreshed at each increment, showing the variation from lower to upper limit. Upon completion of the manipulation, the user is then able to move on to the next step of switching parameterizations so that similar observations may be executed again.

4.4.3 Tracking Changes / Writing the Output File

As explained above, a driving parameter in the assembly model will be automatically varied between user specified limits with a specified number of increments. At each of these increments, the model will be updated to propagate the effects of changing the driving parameter. The feature, geometry, or expression that the user has designated for observation will be changed according to the parametric relationship, whether directly or indirectly related to the driving parameter. At each of the increments the numeric value of the item(s) being observed will be recorded in a variable array. At the end of the incrementing cycle the variable array will be written to a text file and formatted for convenient use at a later time. The values of the driving parameter are also stored and written to the file so that a correlation may be observed.

When the manipulation and update cycle is completed, the user will change parameterizations as explained in section 4.2.3, and follow a similar execution for the new model parameters. The value here will be to observe the behavior of the same geometry or features while manipulating a different driving parameter that coincides with the design strategy of a different discipline. In this, it will become evident as to what the comparable affects are in manipulating the key parameters from different disciplines. The user may quickly observe any overlapping, conflicting, or other sensitivities associated with the geometry of interest.

4.4.4 Swapping Parameterizations

It was mentioned earlier that importing new expressions to replace the current set at the part level is easily performed by the CAD system. This is not true for an assembly, especially a robust parametric assembly containing complicated rules and relationships

that may occur between parts. The method employed in this research will manipulate the CAD system into dealing with the components one at a time, after they have been "neutralized," thus avoiding any complicated relationships or system errors.

The manipulation of the CAD system to enable the import of a new assembly parameterization is made possible by controlling the current state of the assembly during importing, and by delaying the system update until all changes have been made. The function call **`UF_model_update`** within the API is usually executed whenever the user finishes an operation interactively. This function triggers a system update which verifies and updates the system of constraints and geometric entities that may have changed during the previous operation. If this update is executed while inconsistencies or conflicting constraints exist, a failure message is displayed (see Figure 4-10) stating that the action cannot be performed and changes to the model are undone.

By delaying or disabling the system update during the importing of parameters, the tangled web of relationships and equations in the parametric assembly can be untangled and rebuilt without any interruptions or road blocks from the CAD system.

The second way in which the CAD system is manipulated during the importing of new parameters is by changing the state of the assembly so that the system is only working with one part at a time. This is done by changing the "working part" of the assembly so that during the operations only the active component is operated on.

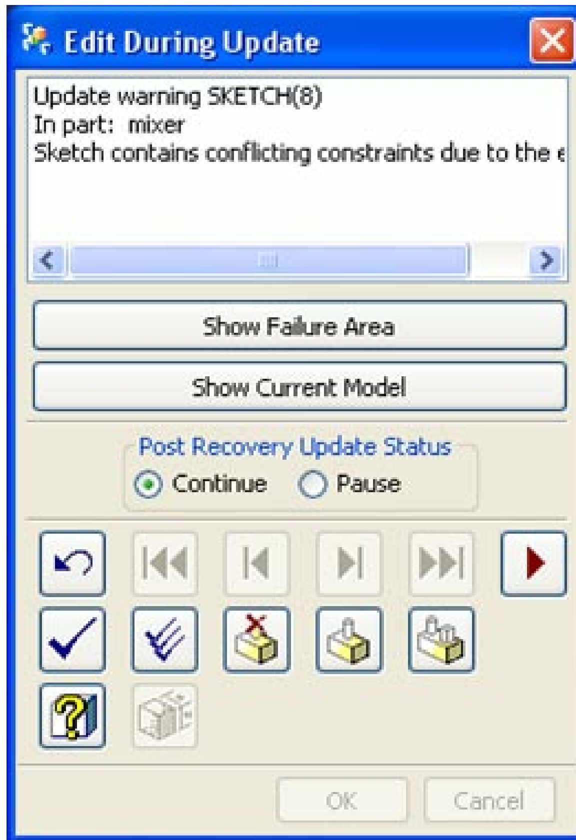


Figure 4-9 Update failure menu

Each of the components is made the working part and is operated on while the system updates are suspended. When a component is made to be the working part as shown in Figure 4-11, the user can access the only the features and expressions of that part to manipulate its geometry. This allows the user to interact with each component without having to open the part in a new window, and does not require that the assembly be closed while its components are operated on. Much of this functionality is built into the CAD system to enable a “top down” approach to modeling with assemblies. The method described here both utilizes and has to work around these built-in assembly features.

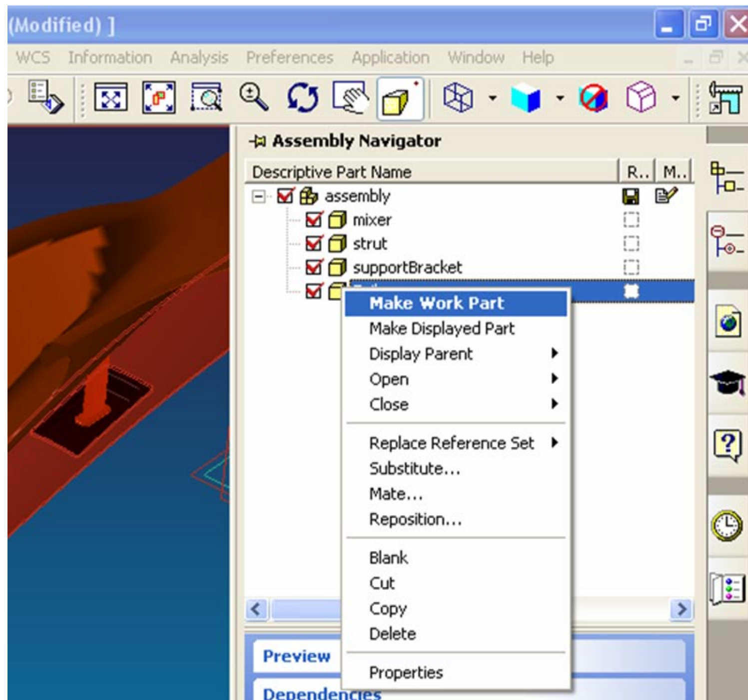


Figure 4-10 Switching an assembly component to the working model

The automated switching of parameterizations will allow previously unavailable operations to be quickly executed by the user. The behavior of the entire assembly can be modified in seconds to allow observation and manipulation of model geometry and relationships heretofore not possible. This step is the key to enabling the engineer to gather important information for comparison and collaboration in the development of optimized geometry without compromising on differing viewpoints up front.

4.5 Verification of Valid Geometry

To accomplish the steps described in the previous sections of this chapter, a complicated set of parametric data needed to be neutralized and replaced with another complicated set of data. When this has been done, it is necessary to verify that the

assembly is still viable. The first indication of whether or not the exchange of parameterizations was successful is in the system update of the model. As explained earlier, the system update was suspended while the assembly parameterization was being manipulated. Once this update is executed, the new parameterization is referenced by the system in solving geometric constraints as well as the relationships and dependencies among the parametric expressions themselves. If the assembly model updates to the new configuration without errors, this is a good indication that the parametric model has been restored to the fully functional state from which the imported parameterization was originally created.

The guidelines associated with the proper implementation of this method require that the parameterizations developed from each discipline use the same base geometry, and comply to a generic naming convention for the base parameters. If these guidelines are followed, the import of new parameters should go smoothly. Only the interactions between parameters as well as the existence of other key parameters not depended upon by any geometry will change from one parameterization to another. These and other items are summarized below.

- All disciplines will use the same base assembly model
- The base model will contain a generic naming convention that will not change
- No new features or geometry should be added
- No new information or parametric control should come from new expressions or parameters only
- Efforts should be made to create a robust parametric model to promote stability during parametric manipulation

As a precaution a few additional checks should be performed on the model after the parameters have been successfully imported. First, the user should visually verify that the assembly configuration looks appropriate from the standpoint of spatial relationships and size/shape of components. The user should also verify that the model responds appropriately to the variation of several key parameters in the newly imported set.

In addition to the manual checks performed by the user, two other checks will be performed on the model upon successfully importing the new parameter set. The custom application to be developed will first check each of the parameters to see if it is referenced or used in the assembly. Any unused or unreferenced parameters will be removed from the set. This not only cleans up the data set, but prevents the user from using or accidentally referencing a parameter that does not belong to the new set.

Next, an interference analysis should be run on the model to verify that the new parameterization has not changed the geometry into a configuration that is physically implausible. A picture showing the menu for a simple interference check is given in Figure 4-12 below.

This check verifies the proper configuration of geometry prior to performing analysis or optimization. As mentioned in chapter 2, one of the greatest problems associated with parametric assembly analysis and optimization is the wasted computing time spent on invalid models (Srinivasan et. al, 2000). This is why it is so important for each of the disciplines developing the different parametric schemes follow the guidelines mentioned above as well as good parametric modeling practices.

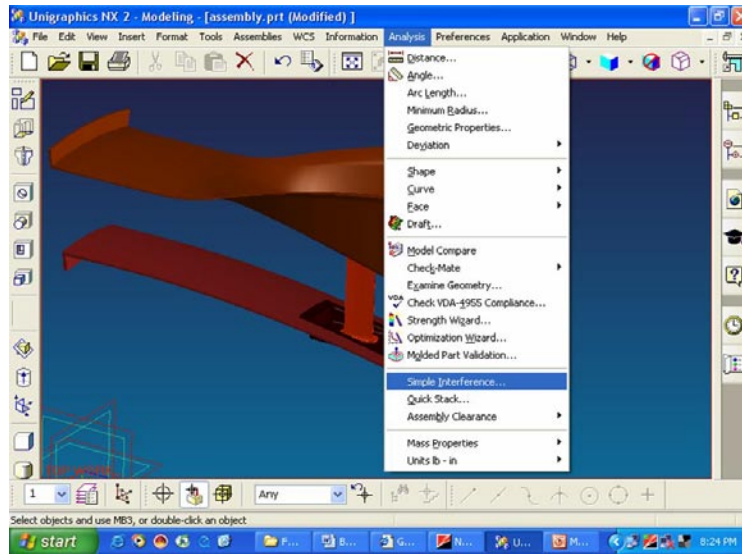


Figure 4-11 Launching the system interference analysis

Once the assembly containing the new parameter set has been verified, the process is complete. This whole process will take only a minute or two, regardless of the size of the assembly. A manual effort at a similar assembly model re-parameterization process, if it were to be attempted, could take several hours depending on the disciplines involved and the size and complexity of the assembly. The method makes possible a process that has been altogether avoided due to the magnitude of time and effort required to undertake such a task.

CHAPTER 5: DEVELOPMENT

This chapter describes how the method presented in chapter four was applied and tested. The questions to be answered during the development of this thesis were outlined in chapter one as follows:

1. What are the types of parametric assembly problems that are typically encountered that will be addressed by this research?
2. How can multiple parametric schemes be seamlessly “swapped” to change the driving parameters of the system?
3. How can the effects of manipulating driving parameters from multiple parameterizations be observed by a concurrent engineering team?
4. How should these effects be presented for comparison and review?
5. What are the uses and drawbacks of this method in design exploration?

This chapter will show how the models and custom application that were developed for this thesis answer the questions above, and overcome the limitations of current practices.

5.1 Classification and Exploration of Typical Problems

The classification of problems addressed by this research was given in the previous chapter, and will be repeated here for convenience. The problem classifications are as follows:

1. How to track "downstream" effects of complex parameterizations
2. How to discover the effects of changes in one parameterization on key geometry of another discipline's parameterization
3. How to explore the sensitivity of key geometry or key analysis parameters to high level changes or vice versa
4. How to find the common or conflicting design space of parameterizations from different disciplines

This section will give examples of how each of these types of problems was addressed during the development of the custom application.

.

5.1.1 Tracking Downstream Effects and Sensitivity

For the purpose of this discussion, tracking the "downstream" effects of parametric manipulation and exploring the sensitivity of key geometry will be discussed together. These two problems were separated into different classes because they describe different problems. However, both scenarios can be addressed in similar fashion by the custom application discussed here.

Enabling the user to track the "downstream" effects of a complex parameterization makes an otherwise rigorous process fairly easy. It may become the case in a large or complex parametric assembly that the effects of dependencies among

parametric relationships become difficult to track or retrace. It is a simple process to query the model to discover parent/child relationships between parameters or assembly components. However, it is not so simple to discover the mathematical relationship between parameters, or in other words the effect or sensitivity that a downstream parameter has to the variation of an upstream or driving parameter.

Consider the following scenario:

P1 = 10, P2 = P1/3, P3 = sqrt(P2), P4 = P2+P3/P1, and P5 = P4^3

If P1 were changed to a value of 13, what would the value of P5 be? Alternatively, if P1 were varied between the values of 8 and 18, what would be the range of values for P5? Discovering the answers to these questions without the aid of the custom application developed here is certainly possible, although it may take a while to dig out each of the relationships above and solve for the values in question. With the custom application, the user is able to simply select the parameters to be varied and observed and within seconds not only watch the model morph into its new form, but have the parameter values of interest stored and outputted to a file. This is of particular value if the user is not the person who developed the parameterization currently active in the assembly. The user does not have to be familiar with the relationships that exist. He/she may only be interested in the sensitivity of key geometry to the driving parameters of the system. This scenario will be explained further in the next section.

5.1.2 Tracking Effects from One Parameterization to Another

One of the main purposes of the NX custom application used for this research is to enable more effective collaborative efforts between engineering disciplines working on

the same design. One way that this is accomplished is through the ability to track the effect that manipulating the driving parameters of the system will have on key geometry or features as viewed by the different disciplines involved in the design.

As mentioned above, the user is likely to be primarily involved with a design team from only one discipline. Design teams are likely to pass certain criteria back and forth to make sure certain envelopes or criteria are not violated in developing the product design. Examples of this might be minimum thicknesses, clearance between components, or profiles of critical geometry. Often in aircraft design, the loft of the flight control surfaces is set before any of the interior components are designed and placed in the assembly, thus this geometry becomes a limiting and controlling envelope for the design of inward components.

With the custom application used here, the user can easily check to ensure that the changes or design envelope that his/her discipline has been exploring does not result in any violation of the criteria set forth by other disciplines. The following example shows how different parameterizations of the exit nozzle assembly, shown earlier, could be used to determine the affect of one discipline's model manipulation on key geometry of another discipline.

The structural group may require certain proportions for the geometry of the strut to avoid any buckling or vibration concerns. The structural parameterization of the exit nozzle assembly allows for direct control of the strut geometry. By contrast, the aerodynamic parameterization of the exit nozzle assembly forces the strut to change its geometry dependent upon other high level parameters such as flow rates. The "STRUT_HEIGHT" parameter will be tracked to show how the variation of aerodynamic

parameters might affect the geometry that is of key concern to the structural discipline. This will illustrate a scenario that would require consideration and cooperation between these two disciplines.

The “STRUT_HEIGHT” parameter in the structural discipline has a numerical value only, and is one of the parameters in the assembly that can be directly manipulated. In the aerodynamic parameterization the same parameter has the following value:

STRUT_HEIGHT=AEROassembly::MainFlowArea/(AEROmixer::FWD_RADIUS-AEROTailcone::fwd_rad)

This parameter is immediately dependent on three other parameters, some of which are also dependent on even more parameters. Ultimately, the numerical value of the “STRUT_HEIGHT” parameter is determined by manipulating one or more parameters at the assembly level. This shows the place that this application could have in performing a sensitivity analysis for certain parameters. If proper communication exists between the two disciplines, the structural group may request that the aerodynamic group provide an envelope, or perhaps a maximum value that they expect to be required for the strut height. By using the custom NX application, a member of the aerodynamics group can observe how the manipulating these high level parameters will affect the “STRUT_HEIGHT” parameter which is of key importance to the structural group. Figure 5-1 shows the resulting values for the parameter of interest when the “TotalFlowRate” parameter is manipulated in the desired range.

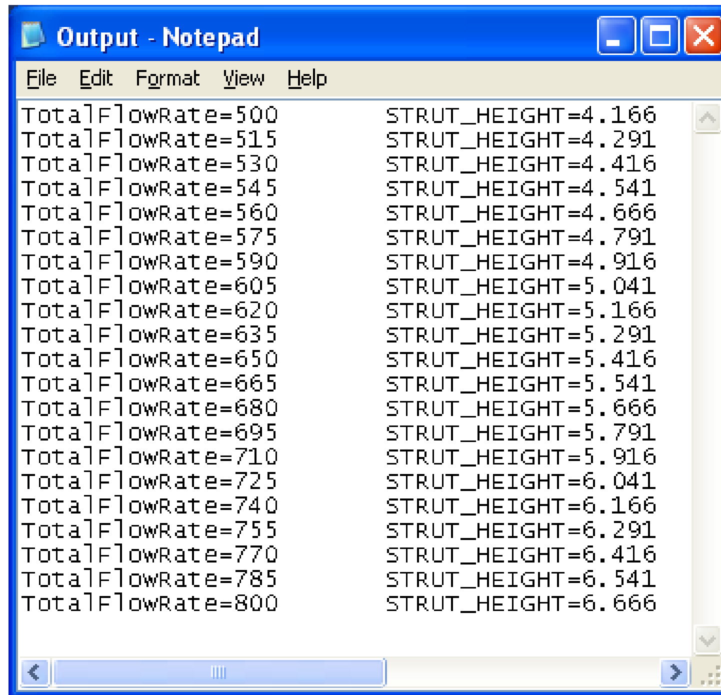


Figure 5-1 Example output from "Aero" parameterization

The data shown can now be used in communication between the disciplines to make any adjustments to either the design of the strut, or to place limitations on the range that the model is allowed to change within. This type of scenario is one motivation for developing the method and application presented in this thesis, and shows the value of being able to quickly analyze dependent parameters, or parameters that play a key role in the multidisciplinary design.

5.1.3 Finding Common or Conflicting Design Space

Another goal in developing the custom application was to be able to compare parametric assembly designs in a way that allowed the discovery of common or conflicting areas of the design space being explored by different disciplines. The

common design space is being defined as regions within which the base geometry being used by different disciplines has the same size or shape characteristics regardless of which parameterization is controlling the assembly. It can also mean that the parameters from each discipline have the same numerical value in this region. This can be classified for individual parameters, features, or for a collection of items.

Finding this shared design space would be of great interest to a multidisciplinary design team. This is the region within which they can safely optimize their design without compromising on the performance objectives of each discipline. This would be very useful if the multidisciplinary design were to be consolidated into a single parametric model for this purpose. Obtaining knowledge about the common design space would allow the development of a master parametric model without unnecessary compromise on key features of the assembly.

The commonalities of the different parameterizations are found in much the same manner as the operations described in the previous two sections. The user is essentially performing the same operations, but with a different intent. Also, when searching for common design space, it is likely that the use of the custom application will be much more extensive, and the output more thoroughly analyzed. To find the common space, the user would observe the same features or parameters from different parameterizations, while manipulating driving parameters. Afterward, the user would analyze the output to determine the overlapping areas of the items being observed. This process could be repeated as many times as needed to encompass all of the features, parameters, or geometry of importance to the common design.

5.2 Swapping the Assembly Parameterizations

One of the key elements in the application developed for this thesis is the ability to quickly and automatically replace the current CAD assembly parameterization with an entirely new one. This was accomplished using the C++ programming language in conjunction with the NX API. The interface was developed using UIstyler, an integrated user interface creation program inside of NX. UIstyler allows a programmer to create a graphical user interface that facilitates the use of NX functionality in a custom application. This approach allowed for the use of custom functions and operations not available directly from NX while at the same time making access and execution of the program very easy. The program was created to fit easily into the NX environment, but is meant to illustrate concepts and methods which could be implemented on other CAD systems as well. Similar challenges exposed in this thesis exist in each of the available CAD systems found today, and could be approached in like fashion.

5.2.1 Use of the API

As discussed in Chapter 3, the NX API (Application Programming Interface) allows for advanced users and developers to access the functions used to execute the operations of the software. This enables the user to customize the application to perform specified tasks tailored to his or her needs. This section will present the methods and functions that allow the swapping of an entire assembly parameterization and how this information is processed by the software.

There are a few main functions which are executed by the GUI while swapping parameterizations. These are called "callbacks" in the NX nomenclature. Each callback is executed at a different stage in the program.

The constructor callback is an initialization for the program that is executed when the application is launched. It can be used for many things such as setting the sensitivity or availability of features on the GUI. The constructor may also be useful in setting the initial value of certain fields. For the custom application the constructor callback is used to gather all relevant information about the current assembly and each of its components. This information is then utilized automatically when other operations are performed. The data is stored as long as the application is running, and can be retrieved at any time.

The browse callback is use to invoke a file selection dialog. The user is able to browse the computer or network to choose the folder containing the parameter expression files to be imported. The programmer is able to specify filters enabling only the selection of certain file types based on their extension, such as ".doc", ".txt", or in this case ".exp".

When the "save current" callback is executed, another file selection dialog box is opened for the user. This will prompt the user to choose a location for the export of expression files from the current assembly. When the location is determined, and the user clicks the "Ok" button, this location is stored into the class data structure mentioned above.

5.2.2 Custom Functions: Enabling the Process

The key to custom applications and enhancing the capabilities of the CAD system is the integration of custom functions into the program. The research and development of

this thesis came about upon discovering a potential need that the current parametric CAD systems would not be able to fulfill. It was apparent from multiple sources discussed in chapter two that a great amount of effort was being put toward developing parametric models and integrated systems to facilitate design optimization. One subject that was addressed only by a fellow graduate student was using parametric modeling principles and custom applications to change the behavior of the CAD model to explore different designs. This thesis has taken that idea from a different angle by dealing with parametric assemblies and manipulating the controlling parameters located within the model itself. This is only made possible through integrating custom functions into the application.

There are several custom functions that were developed to enable the operations that were desired for an interchange of parameterizations. The primary roles of the custom functions were gathering model data and user input, processing or manipulating the data, and verifying and cleaning up the model after the swap. A few of the critical functions will be presented and explained here.

5.2.3 Exporting Parameter Sets

The first operation of the program to be executed, if the user has chosen to save the current state of the assembly, is to export the existing parameter sets. This has the effect of "time stamping" the assembly, and creates a set of expression files that can later be imported to restore the assembly to its current state and configuration.

The operation of exporting expressions is quite simple to do interactively for a single part. Exporting expression files for an entire assembly is not much more difficult, but has to be done for each component in the assembly. This could obviously be a hassle

for larger assemblies. The code used to execute these operations automatically will be shown below. The input that is gathered from the user consists of the location that he/she would like to export the files to, and a keyword to be attached to the end of each file name to help identify the files for future reference and import. The main excerpt of code used to perform this operation is shown in Figure 5-2 below.

```
save_key = UIS_getStringValue(dialog_id, EXP_SAVE_KEYWORD);
char* key_and_ext = strcat(save_key, ".exp");
      for(int i=0; i<Assembly.size(); i++)
      {
      char* path = Assembly.at(i)->GetExportPath();
      char* name = Assembly.at(i)->GetName();
      strcat(path, name);
      char* export_file = strcat(path, key_and_ext);
      UF_ASSEM_set_work_part(Assembly[i]->GetPartTag());
      Assembly[i]->ExportExp(export_file);
      }
UF_ASSEM_set_work_part(Assembly[0]->GetPartTag());
```

Figure 5-2 Code used to export expression files from the CAD assembly

The first two lines of code in the function shown in Figure 5-2 declare variables to set the key word to be appended on the end of the file name and to set the file extension. The function then enters a loop in which the expression file is exported for each component in the assembly. There are three things that happen for each component of the assembly when exporting the parameter or expression file. First the full path for the location of the file is retrieved and concatenated with the file name and extension,

including the key word, at the end. Second, each component is temporarily made the "work part" for the assembly. This has the effect of opening the part for editing and operating on. Once the component is made the work part for the assembly, the final step is taken to export the expression file. This process is repeated for each component in the assembly until all controlling parameters have been exported and saved as expression files corresponding to each component. The base or root part is then set to be the working part of the assembly again, and the update process is complete.

5.2.4 Importing Parameter Sets

Importing new parameter sets into the assembly is performed in a similar manner to the export operation described in the previous section. The difference is that the expressions being imported have to replace the existing expressions in the assembly. This requires some preparation and evaluation of the assembly to allow the process to happen without errors or system failures. The code used for importing new expression files into the assembly will be shown in Figure 5-3 below, with explanations of each segment to follow.

The first few lines of code, before entering the for-loop, retrieve the key word that is appended to each expression file name and concatenate it with the file extension ".exp". This will be used by the program to identify the parameter files to be imported. The custom function "joinStrings" is used frequently throughout the program to dynamically join two strings together.

Once all of the file location and naming operations are complete, there are three functions executed to take care of the importing of new parameter sets:

UF_ASSEM_SET_WORK_PART(), Assembly[i]->DerefExp(), and Assembly[i]->ImportExp. The first function sets the assembly work part. The only argument needed to execute this function is the part "Tag", which is a NX identifier unique to each feature or entity in the CAD model. Once the designated component has been made the work part, all operations will affect that part only.

```
new_key = UIS_getStringValue(dialog_id, EXP_KEYWORD);
char* ext = ".exp";
char* new_and_ext = joinStrings(new_key, ext);
for(int i=0; i<Assembly.size(); i++)
{
char* path = Assembly[i]->GetImportPath();
char* name = Assembly[i]->GetName();
char* pathName = joinStrings(path, name);
char* import_file = joinStrings(pathname,new_and_ext);

cout << "Import file: " << import_file << endl;
UF_ASSEM_set_work_part(Assembly[i]->GetPartTag());
Assembly[i]->DerefExp(Assembly[i]->GetPartTag());
Assembly[i]->ImportExp(import_file, Assembly[i]-
>GetPartTag());
}
UF_ASSEM_set_work_part(Assembly[0]->GetPartTag());
```

Figure 5-3 Code used to import expression files into the CAD assembly

The **Assembly[i]->DerefExp()** function is a custom function developed to "de-reference" the expressions in the active part. This is necessary to avoid conflicting parameter sets as well as system errors that occur when dependencies and other references exist between expressions and assembly components. The active component becomes neutral and independent of the other components in the assembly temporarily to allow for the import of new parameters and rules. The code for this operation will be shown in figures 5-4 and 5-5 below.

```
void Component::DerefExp(tag_t part_tag)
{
     double exp_value;
     int num_exps;
     tag_t *exps;

     UF_MODL_ask_exps_of_part(part_tag, &num_exps, &exps);
     for(int i=0; i<num_exps; i++)
     {
      char* exp_string = new char[128];
      char* new_string = new char[128];
          UF_MODL_ask_exp_tag_value(exps[i], &exp_value);
          UF_MODL_ask_exp_tag_string(exps[i], &exp_string);
          new_string = RewriteExp(exp_string, exp_value);
      UF_MODL_edit_exp(new_string);

      UF_free(exp_string);
      delete [] new_string;

   }
}
```

Figure 5-4 The custom function "DerefExp", which neutralizes the expressions in the active model

The "**DerefExp**" function shown in figure 5-4 is a method within the C++ Component class. The reader may remember that the data stored during the execution of

the program is stored in an object oriented class structure where the assembly consists of a vector of Component classes. The function above begins by asking for all expressions within the active part in the assembly using **`UF_MODL_ask_exps_of_part`**. The function loops through each of the expressions and extracts the expression string, or left hand side, and the expression value, or right hand side. A typical parametric expression from the assembly would be something like "strutHeight=intakeHeight*3.2". This expression contains a reference to another expression "intakeHeight", as well as an arithmetic operator. This expression could not be replaced by the system because of the dependency that exists. The expression will be re-written to neutralize it and avoid any errors. The expression gets rewritten by another custom function called "**`RewriteExp`**", which is shown below in figure 5-5.

This function takes the existing expression and replaces the right hand side with its equivalent numerical value with twelve significant figures of precision. This is accomplished by first converting the expression value to a double variable with twelve significant figures. The next portion of the function is a little bit tricky to follow, but proceeds like this. The "cursor" is moved to the end of the expression, simulating this same action as if it were done interactively to edit a string of characters. The cursor is then stepped backward until it encounters the '=' sign in the expression. It is then stepped forward once and a carriage return '\0' is placed there, effectively chopping off the right hand side of the original expression. The final string is then created by appending the numerical value extracted earlier to the remains of the left hand side and '=' sign just created. The final product is the original left hand side of the expression, and the numerical right hand side with twelve significant figures of precision.

```
char* RewriteExp(char* exp_string, double exp_value)
{
     // changes something like: p1=2*p2 to p1=12
   char* final_str;
     char* conv_val;
     char val_string[256];
     int sig_figs = 12;
     conv_val = _gcvt(exp_value, sig_figs, val_string);

     char* new_one = strdup(exp_string);
     int path_length = strlen(new_one);
     char* cursor = new_one + (path_length -1);
               // set cursor to end of string
     while(1)
     {
          if(*cursor == '=')
          {
               cursor++;
               *cursor = '\0';
               break;
          }
          cursor--;
     }
     final_str = joinStrings(new_one, conv_val);
     return final_str;
}
```

Figure 5-5 The "RewriteExp" custom function which neutralizes the parametric expression

With the rewritten expression, the final operation of the **`DerefExp`** function is performed by calling **`UF_MODL_edit_exp()`** which replaces the expression in the CAD model with the new neutral expression. The operations shown in figures 5-4 and 5-5 are performed for each expression in each component of the assembly in a fraction of a second. This portion of the overall process alone would take several minutes to perform interactively, even on a small model.

The only remaining operation as shown in figure 5-5 is the "**`Assembly[i]->ImportExp()`**" function. This function is called only once for each component in the

assembly. This is a simple function because of all of the preparation already performed on the expressions in the part model. The program calls the UGOpen function **`UF_MODL_import_exp()`** which requires the file to be imported, and an integer which specifies how to treat expressions encountered that have the same name. In reference to importing expression files, the NX documentation states:

> "There may be times when you have expressions in the text file that have the same name as expressions already in your part file. When this conflict occurs, the system either keeps the existing expression or replaces it with the expression in the text file. You control how expression name conflicts are handled with one of these settings: Replace Existing, Keep Existing, or Delete Imported."

Because of the preparation discussed above, the "Replace Existing" option is specified in the code, and an imported expression with the new rules and references is able to be imported to replace the numerical or neutral expressions that have been set up. This issue was discussed in chapters three and four in terms of Set Theory. The conflict seen by the system occurs because there are two sets of expressions containing elements that are identical. By importing the new set of parameters, an attempt is being made to unite the two sets. The conflict is easily overcome because of the preparation of each expression discussed earlier in this section.

5.2.5 Updating the Assembly

While the parameter sets from the assembly model were being exported and imported, any update of the CAD model was suspended. This was done to avoid potential update failures that could occur due to inconsistencies that would exist when the assembly was only partially parameterized. Any links, dependencies, or relationships between components of the assembly could cause an update failure if the new parameters

were loaded for one component, but not another when the system tried to update the model. Suspending any model updates facilitates the mass import and export of parameter sets without any check or interference in the process.

This is an enabling step for the automation of the process; however, it is also a risk that is taken. There is some risk that the parameter sets being imported have been tampered with, or are not compatible with the current CAD assembly. This should not happen if the original neutral model was used by each of the participating design teams. This is one reason why the user would want to have the current state of the assembly saved before importing a new set of parameters. If a problem occurred that seemed difficult to fix, the user could simply import the original parameter set, thus restoring the model to the state that it was in before the errors occurred.

The actual update of the assembly is triggered by a simple call to the UG Open function **`UF_MODL_update()`.** This function updates all features of the model in the order in which they were created. Each of the features being updated are dependent upon or affected by the expression list. This same type of update may also be observed when interactively editing the model expressions. The user may edit one or more expressions without actually changing the model. Once the user clicks "Apply" or "Ok", the model updates, and any errors are brought to the user's attention. The user has the options of undoing the changes, ignoring the errors, or listing the errors found.

With the automated import as discussed above, the update is suspended while nearly every expression in the assembly is modified. Once this is complete, the update is executed and the model responds to the changes made. If no errors are communicated to the user, the system has been satisfied, and the model has updated successfully. There

are now a few steps the user should take to inspect the model and ensure valid geometry and assembly configuration.

5.2.6 Validation and Verification of the Model

As discussed in section 4.4, the model needs to be verified once the parameter exchange has taken place. There are four main questions that need to be answered, once the model has had a successful system update, to verify that the assembly model is still viable:

1. Does the model visually look correct?
2. Does the model seem to respond appropriately to parameter changes?
3. Are there any unreferenced or unused parameters in the model?
4. Are there any interference conditions within or between components?

The first two questions were quickly and easily answered by the user. A visual inspection was performed which included checking to see that each component is in the correct position relative to the others. By modifying a few of the key driving parameters the user verified that the model responded appropriately. This was one of the most basic ways in which the execution of the code was verified while developing and testing the software. A few key parameters would be different in the set of parameters to be imported when the program was executed. This allowed for an expected visual change to occur. Another simple test was to check the list of expressions to verify that the newly imported parameters were present.

To answer the third question about model validation, there were automatic checks put into place in the code to verify each parameter or expression in the assembly. An unreferenced expression is likely to be "left over" from the previous parameterization and should be removed. Any expression that is being referenced or used by a feature in the model or by another expression cannot be deleted directly. Thus, any expression that can be deleted directly is an unreferenced expression. The code was set up to loop through the expressions attempting to delete each of them. If there was an error, the code simply moved on, resulting in the removal of all unreferenced expressions.

The drawback of this method is that the user would not be able to create expressions just for the sake of holding data or numbers for reference only. If the user wanted to bookmark a value or equation for reference while modeling, the clean up process discussed would delete such expressions. Fortunately, this should not be a big concern because there are several other ways of accomplishing the same thing. One way to create reference data that does not control the model in any way is through what NX refers to as model attributes. The user can store extraneous information about any feature, point, curve, or even information not associated with any geometry, with a model attribute. This data can be retrieved manually, or through the programming API for reference by the user. Other places to store such information would be in a spreadsheet, database, or text file, which could all be linked to from the within the model or by the programming API.

The fourth question to be answered was "Are there any interference conditions within or between components?" It was mentioned in the method chapter of this thesis that this question was to be answered either automatically by the custom application, or

by having the user execute the analysis functions built into the CAD system itself. It was determined that even with very significant effort, it would not be likely that the custom application could reproduce the built in analysis functions within NX with any improvement on time spent, or feedback given to the user. The analysis and visualization tools that are already a part of the interference analysis are very helpful, and would have been difficult to reproduce, let alone improve upon. Therefore, in order to verify the lack of interference conditions in the assembly, the user is advised to become familiar with and utilize the built in interference analysis operations of the CAD system.

5.3 How to Track, Observe, and Output Parametric Manipulation

The ability to track and observe the changing values of key parameters or features of the parametric assembly is crucial to studying the design space of common geometry. Typical use of parametric manipulation only allows the user to view snapshots of the model, or to export current values of the parametric expressions. This does not permit the user to observe *how* the values changed, or what the assembly configuration looked like during the transition from one state to another.

The custom application used here allows the user to know exactly what is happening to geometry of interest during model variation. The observation takes place both visually as well as being recorded analytically. The user is able to control the displayed changes and the data being recorded by specifying the number of increments that the driving parameter will use to change from the lower to upper limits which are also specified by the user. Figure 5-6 shows the user interface with items selected for variation and observation. The lower and upper limits are set manually, and the number

of increments or evaluation points is set using a slider bar. If the current value of the variational parameter is not between the chosen limits, the user is notified with a pop up message. Upon execution, if the number of observed points does not seem sufficient, or if the model seems to "jump" from one value to another, the user can increase the number of increments to be evaluated.

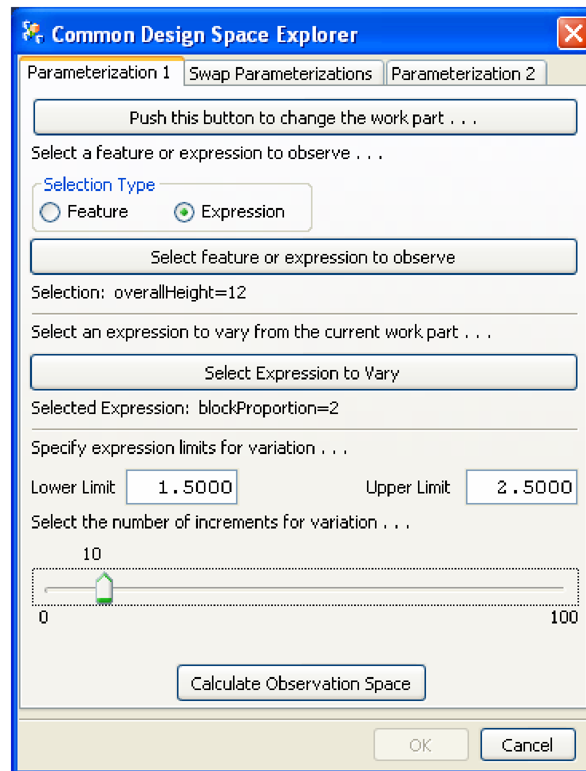


Figure 5-6 User interface with values specified

While the model is changing, the values of the parameter(s) that are of interest are recorded in a variable array at each incremental change. When the user is done manipulating the driving parameters, these values are written to a file and formatted for ease of use. A simple output file is shown in figure 5-7. As you can probably guess, this

file was created to track the values of the parameter "b1Length" within two parameterizations. The values in the first section range from 5.97 to 6.97, while the values for the same parameter in the second section range from 5.0 to 7.5. This data could be used to help address one or more of the questions raised in section 5.1. The data is also easily copied into a spreadsheet for further analysis or graphical representation. The format for the output data is something that can be easily changed in the program code, but has been left as simple as possible for ease of use.

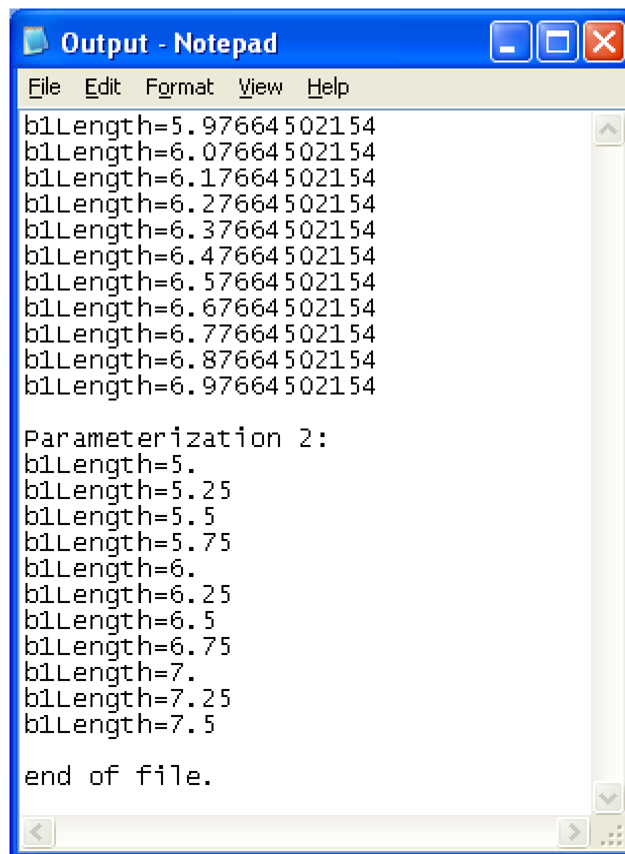


Figure 5-7 Simple output file comparing parameters

5.4 Uses and Drawbacks of this Method

The uses of this method have been thoroughly covered in the previous sections, and will only be listed for review with brief comments here. The problems addressed by this research were listed in section 4.1 as:

1. How to track "downstream" effects of complex parameterizations
2. How to discover the effects of changes in one parameterization on key geometry of another discipline's parameterization
3. How to explore the sensitivity of key geometry or key analysis parameters to high level changes or vice versa
4. How to find the common or conflicting design space of parameterizations from different disciplines

Each of these problems is addressed when using the custom application in NX. This tool is basically a program which enables exploration, manipulation, and observation of a parametric assembly model that has been previously unavailable. The application overcomes some of the limitations of parametric CAD systems by employing custom functions designed to allow the user to manipulate assembly models and extract desired information quickly without running into typical system errors. This is done through utilization of the NX API to build a custom application that performs both internal NX functions combined with custom functions enabling new operations to be performed.

There are some drawbacks and limitations inherent in the methods and application presented here. The first limitation is that the model to be used by more that one discipline must contain enough information to be an effective base or starting point.

Also, a naming convention for the parameters has to be chosen and adhered to throughout the processes put forth in this thesis. This is not a difficult thing to do, but there must be proper consideration up front to ensure that the generic model contains the proper type of geometry and definitions to be able to be parameterized effectively by each of the disciplines.

While the application developed here is put forth as a means of performing complicated or unavailable tasks quickly and easily, the size of the assembly being used will still be prohibitive at some point. Very large assemblies (with hundreds or even thousands of components) may not work well with a program such as the one here. The methods here are most suitable for a small to medium sized system, containing a more manageable amount of components and parameters. While some of the functions used here certainly could work on large assemblies, this has not been tested, and will be left for future research.

CHAPTER 6: RESULTS

Chapter 5 showed how the questions and problems addressed by this thesis were confronted and resolved. As explained, a custom application was designed and programmed to accomplish parametric exploration and parameterization swapping as outlined in the opening chapter of this thesis. The results of the test executions and validation of this program and the models used will be discussed here.

6.1 Creation and Validation of Parametric Assembly Models

As discussed in the previous section, a neutral or "dummy" model was developed to be the basis from which all other parameterizations were derived. This was both necessary and useful in the preparation of CAD assemblies to be controlled by parameterizations influenced by different analysis disciplines. Generic variable names such as "strut_length", or "flange_height" were assigned in the dummy assembly model. When a variable was not expected to be specifically referenced or used by another expression or component, the automatically assigned variable name, such as "p7", was left unchanged.

The parametric assemblies for each discipline were created by copying the dummy assembly and then changing expression values, equations, and relationships to exhibit behaviors specific to the intended performance variables of that discipline. In

some cases, new expressions were created as reference points for calculations, limits for expression values, or results of calculations based on the current value of expressions in the assembly. The parametric assembly models were created by the author, and were intended to exhibit differences from each other significant enough to cause the conflicts and interactions discussed throughout this thesis. The models proved to be sufficient for these purposes, and behaved as expected during the testing and execution of the parameter swapping application. The models were not intended to be fully representative of a robust and well thought out parameterization that could have been developed by a team of engineers from a specific discipline. The development of such a parameterization, if done using the guidelines and methods discussed in this thesis, should exhibit the same behaviors if an attempt were to be made to perform parameter exchange, or multidisciplinary optimization, as outlined here. This exercise will be left to future research and exploration by a team of designers.

6.2 Successful Swapping of Assembly Parameterizations

The testing of the parameter swapping application proved to be interesting since the actual execution of the program took only a few seconds. One might think that this portion of the program either works or it does not, with little room in between. There are, however, several things that can cause the program to succeed or fail. The tests that were performed to validate successful switching of parameterizations is presented here.

Whenever any problems arose during development, it had to be determined if the problems were caused by bugs in the code, or by errors or incompatibilities in the CAD

model. As discussed earlier, the use of the custom application developed here requires a well thought out parametric model with robust characteristics.

For initial testing, a few simple assemblies were modeled using basic geometric shapes. This made the changes resulting from a successful parameter swap very easy to observe. The expression sets being imported would change the size or shape of the geometry in the model as expected.

Once the code seemed to be performing as expected, the jet engine exit nozzle assembly discussed throughout this thesis was tested. With the test assembly, the same visual observations could be made regarding geometry changes, but further investigation would be required to completely prove the success of a full parameterization swap.

To test the execution of the program, each of the parameterizations developed were used as a starting point for parameter swapping. The structural parameterization would be imported in place of the aerodynamic parameterization and vice versa. Each combination of exchanges was tested. This was done to simulate any of the possible scenarios that would be encountered in an optimization setup. An example showing a typical geometry configuration change due to a parameterization swap is shown below in Figure 6-1. This figure shows a change in the size and shape of components, as well as their relative positions in the assembly. What may appear as a simple geometry change actually involves an entire shift in driving parameters and model behavior. The change is completed in a fraction of a second for an assembly this size.

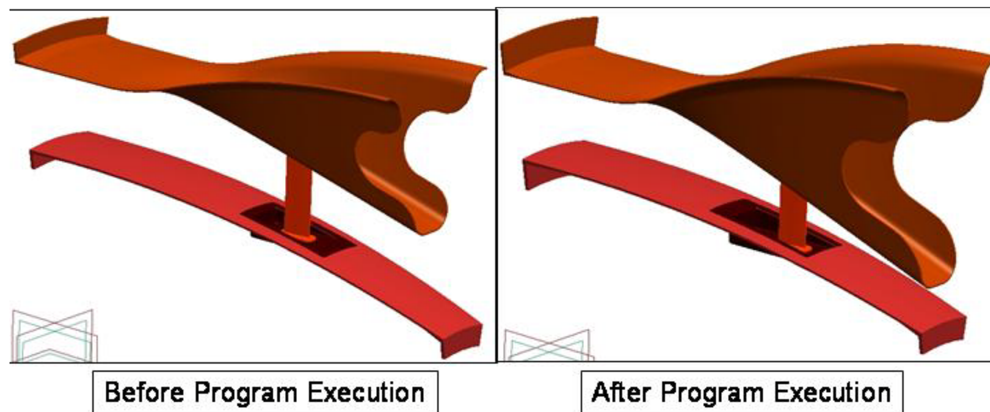


Figure 6-1 Example of geometry changes upon execution of the program

Another test that was performed was to simulate moving from one extreme of the design space to another extreme while swapping parameterizations. In other words the values of many of the driving parameters in the current assembly model would be at or near the likely lower or upper bound of the typical variation for that parameter. When the new parameterization was imported, these values would be changed to the other boundary in the design space. Large changes in parameter values have a greater likelihood of causing regeneration failures than small changes. This would test whether or not the assembly models were robust in the way they were parameterized.

In each of the tests performed, the application code, and the parametric models performed as expected. Any failures due to implausible or invalid geometry being generated were discovered early in the development stages, and were corrected with minimal effort. Also in each case, the code was executed quickly, with each of the internal functions performing consistently as designed.

As discussed in chapter one of this thesis, the time savings represented by automated parameterization swapping increases with the complexity of the assembly that the operations are being performed upon. The test assemblies used here contained relatively few components, with each component being controlled by only 10-20 parametric expressions. With larger assemblies, a team of engineers would not be likely to work collaboratively on separate parameterizations of the assembly even if they were aware of that as an option. This is because until now it wouldn't have made much sense to create separate parameterizations with a common goal in mind due to the cumbersome and tedious effort that would have been required to switch parameterizations within the same assembly model. In other words, the validation of automated swapping has opened the door for a new kind of design collaboration that can bring different disciplines onto the same page with regards to parametric assemblies. A parametric assembly model can now be used as a baseline for analysis and optimization, and information about model configuration can be quickly adjusted and transmitted between disciplines.

6.3 The Application Interface Design

The application GUI is the front end of the processes described chapter 5. This section describes how the interface accomplishes successful setup and execution. The interface is designed for minimal input, allowing for as much automation as possible.

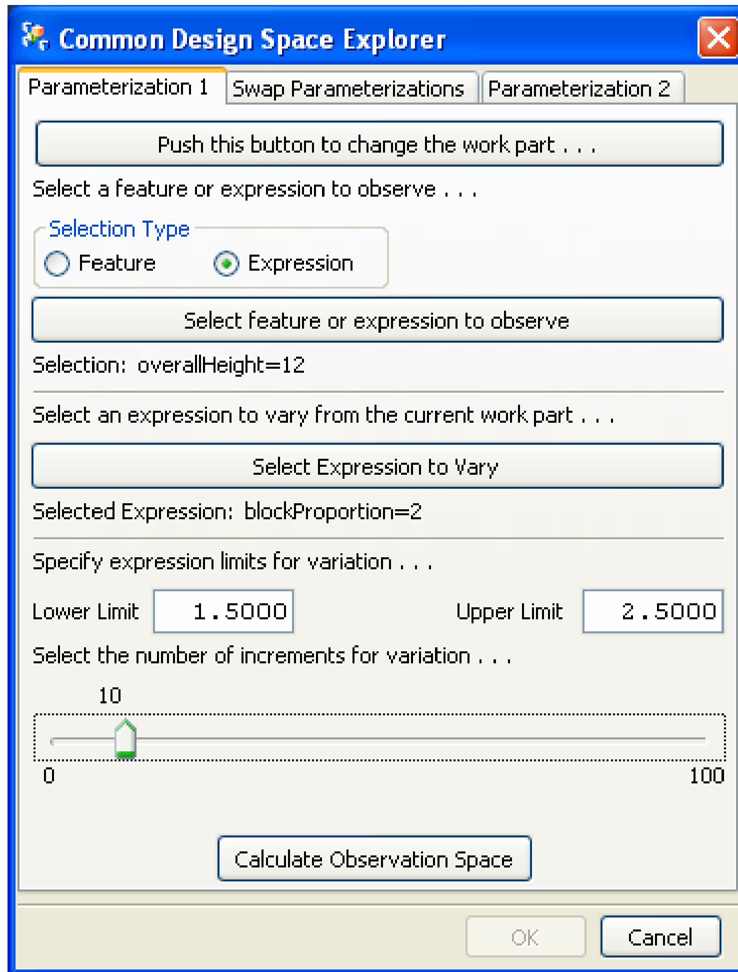


Figure 6-2 Assembly parameterization GUI

As shown in figure 6-2, the user interface requires relatively few pieces of information from the user. This is the first of three tabs available to the user. Setup flows from the top of the menu down. First the user can change the "work part" in the assembly if needed. Next the user selects whether to observe a single expression, or all the expressions of a chosen feature. The user then selects an expression to vary, and specifies the lower and upper limit of the variation. The number of increments to

evaluate during variation is chosen with a simple slider bar. Once each of these items is specified, the user can execute the calculations and observe the geometry changes.

The second tab shows the options for switching assembly parameterizations. This is a simple and quick process that is highly automated as explained in chapter 5. This menu is shown in figure 6-3 below.

Again proceeding from the top down, the user first selects whether to save the current parameterization and locates a folder to place the expression files. A key word is designated which is appended to the expression file names for later use. Similarly, the file location and keyword for the expression files to be imported is specified. All that remains is to click the swap button and a few seconds later the new parameterization has replaced the existing one in the assembly model.

The third tab in the user interface is identical in appearance to the first tab. The only difference is that all of the operations are being performed on the second parameterization that was just imported. In this way, the data from both parameterizations can be stored and formatted for the output file. The user steps through the same operations described above and the process is complete, presses “OK” to exit the application. All of the gathered information is written to a default output file to be used later.

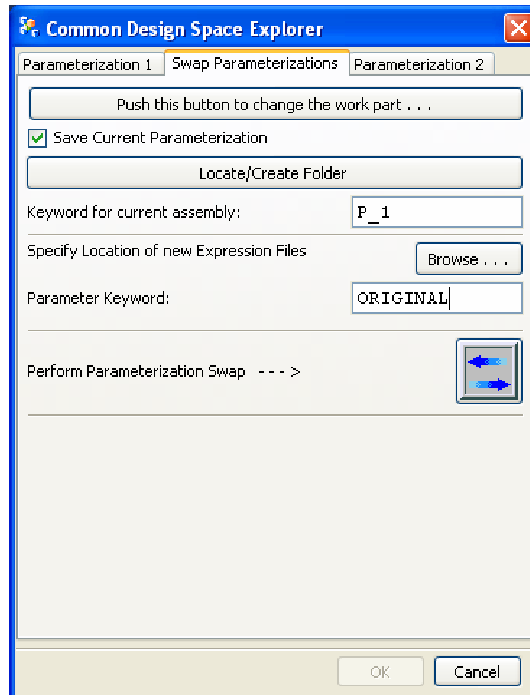


Figure 6-3 Second tab of the custom GUI

If proper model preparation guidelines and good modeling practices have been followed, the execution of the program takes only a few minutes. The user interface is simple enough to be self explanatory, but it is intended to be used by fairly advanced users who understand what the execution of the program entails. The interface has proved successful in fulfilling the intended needs and purposes of its design.

6.4 How the Effects of Parametric Manipulation Were Observed

Observation of chosen features or expressions is a feature of the custom application that is not available through normal use of the CAD system. The user can track the changes to an item that may be in a completely different component in the

assembly than the current work part. In lay terms, the user can push on one end of the assembly and see what is happening on the other end

This process is made possible through the use of the NX API in conjunction with some custom functions developed by the author. As presented in previous chapters, the key to enhancing, circumventing, or otherwise overcoming the limitations of the CAD system is the use of such functions to tailor the performance of the software to the needs of the apparent design challenge. In this case, the author programmed into the application the ability to let the user point to which objects in the assembly he/she would like to deal with. The program, unlike the CAD system on its own, has the ability to track the details of these objects or features. Each time the user selected an item to observe or to manipulate, the program stores the "tag" of that object. The "tag" is used to query information for the object, whether or not it is in the active component. Some example code showing the gathering of observation data is shown in Figure 6-4 below:

```
if(toggle_label == "Expression")
     {

      pObserve1 = getPartExps();

      cout << "pObserve1 = " << pObserve1 << endl;

      UIS_setLabel(dialog_id, "SELECTED_EXP_LABEL",
pObserve1);
      //UF_free(exp_tags);
     }

     obs1Part_tag = UF_ASSEM_ask_work_part ( );
```

Figure 6-4 Code fragment for choosing an expression to observe

The code above sets a variable called “pObserve1” equal to the string that is returned from the function “getPartExps()”. This function allows the user to choose an expression from the active part, and returns the value of the expression chosen by the user so that it can be stored into memory. This function is shown below in Figure 6-5.

```
extern char* getPartExps ()
{
    int  error_code = 0;
     int response;

    if ( ( error_code = UF_initialize() ) != 0 )
          return (0) ;

    if ( ( error_code = UF_STYLER_create_dialog (
"expList.dlg",
         EXP_cbs,      /* Callbacks from dialog */
         EXP_CB_COUNT, /* number of callbacks*/
         NULL,        /* This is your client data */
         &response ) ) != 0 )
    {
          char fail_message[133];

          // Get the user function fail message based on
the fail code.
          UF_get_fail_message(error_code, fail_message);
          UF_UI_set_status (fail_message);
          printf ( "%s\n", fail_message );
    }

     return retExpression;

     UF_terminate();
}
```

Figure 6-5 Code Fragment for the “get PartExps” Function

Once the user chose the item(s) to observe, and specified the parameter to vary, including limits and number of increments, the main execution was ready to be performed. The execution of the program was tested several times under differing circumstances in order to verify the functionality of the program and to make sure that all of the desired information was tracked and stored. As described in earlier chapters, as the assembly model is manipulated, the resulting changes are tracked and stored in variable arrays to be written to an output file. The code segment used to perform these steps is too large to include here, but will be included in the appendices at the end of this thesis.

In each of the cases where the test models were used, the program successfully tracked and stored the information for the specified features, expressions, or geometry of interest. This information was then formatted and written to and output file for future use. The results of this output are shown in the next section.

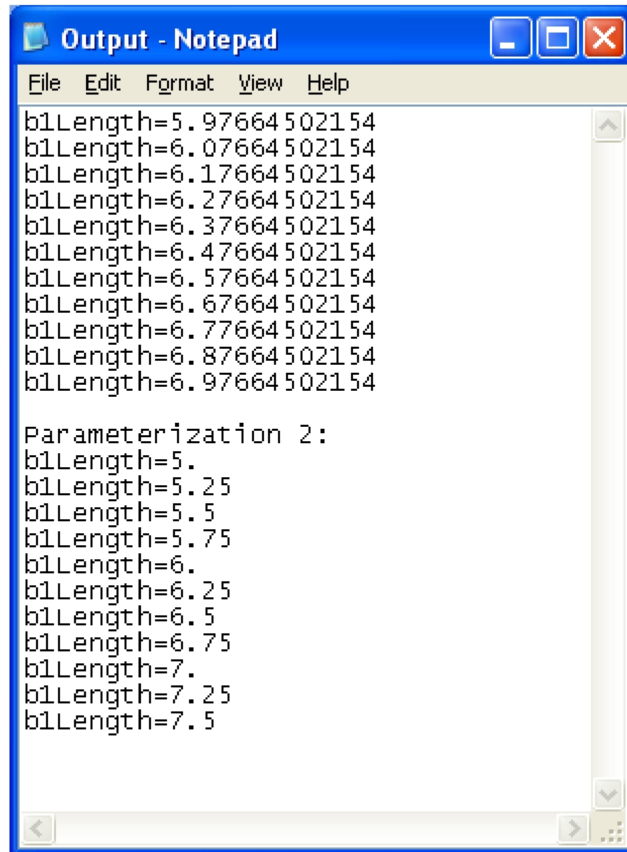
6.5 Presenting the Observation Data for Review

One of the questions to be answered in this thesis was to determine how to present the results of parametric variation so that they could be used and interpreted by the user. There were several possible ways to present the output from the custom application. The choice was essentially whether to present the results graphically with some sort of chart or graph, or to simply present the user with the numerical values of the data being observed or manipulated. Ultimately, it was decided that the output should be given to the user in the form of a text file containing the numerical data in a simple delineated format. This solution certainly is not the most glamorous, but is more universal and can

easily be used in a variety of ways. Presenting the user with a simplified output file allows him/her to the freedom to use the information in whatever way is most beneficial.

The output file shown below gives an example of the type of data that would typically be recorded, and the format of the output file. Figure 6-6 shows the results of the user choosing to observe the parameter b1Length during manipulation of two different parameterizations. This data can easily be imported into a spreadsheet or other program for comparison or graphical analysis. One thing that should be pointed out is that the parameter being observed was dependent on other parameters. This means that what the user saw in the output file was the numerical value of the parameter of interest at each increment. The actual parametric expression used and shown in the CAD system would look the same at each stage. In this case, the parametric expression in the model was "b1Length = overallHeight/3" in one parameterization, and "b1Length = overallHeight/blockProportion" in the other. In both cases, the numerical result of this expression was recorded at each increment during the variation of the parameter "overallHeight".

The file below shows the typical format for the expected output files, showing the values of the parameter(s) being observed for two or more parameterizations of the assembly model. As discussed in chapter 4, there could be a myriad of uses for this data depending on the interest or intent of the user. The custom application was executed in much the same way to address several different types of problems.




```
b1Length=5.97664502154
b1Length=6.07664502154
b1Length=6.17664502154
b1Length=6.27664502154
b1Length=6.37664502154
b1Length=6.47664502154
b1Length=6.57664502154
b1Length=6.67664502154
b1Length=6.77664502154
b1Length=6.87664502154
b1Length=6.97664502154

Parameterization 2:
b1Length=5.
b1Length=5.25
b1Length=5.5
b1Length=5.75
b1Length=6.
b1Length=6.25
b1Length=6.5
b1Length=6.75
b1Length=7.
b1Length=7.25
b1Length=7.5
```

Figure 6-6 Example Output File

CHAPTER 7: CONCLUSIONS / RECOMMENDATIONS

The objective of this research was to develop a method to allow better exploration of parametric assembly design involving multiple disciplines. Included in this effort, was the ability to perform a complete change of parameterizations controlling the assembly CAD model. This was accomplished through the creation of a custom application within the NX CAD system. The application was programmed to use a combination of custom functions and CAD API functions to perform all of the desired operations. The capabilities developed in the custom application enable multidisciplinary exploration of common geometry while maintaining discipline specific parameterizations.

The questions to be addressed during development are summarized as: What are the types of problems addressed by this research? How can parametric schemes be seamlessly "swapped?" How can the effects of manipulating driving parameters from multiple parameterizations be observed by a concurrent engineering team? How should these effects be presented for comparison and review? What are the appropriate uses of the method? And, what are the drawbacks and limitations?

The questions above were answered, and the objectives of this thesis accomplished as presented in chapters 5 and 6. A summary of these conclusions is given here. The types of problems typically encountered in parametric assemblies that were addressed by this thesis were defined as:

1. How to track "downstream" effects of complex parameterizations
2. How to discover the effects of changes in one parameterization on key geometry of another discipline's parameterization
3. How to explore the sensitivity of key geometry or key analysis parameters to high level changes or vice versa
4. How to find the common or conflicting design space of parameterizations from different disciplines

Examples of how each of these problems was addressed by the custom application were given in section 5.1. As demonstrated, the custom application addresses several types of issues, although the execution of the program is performed similarly for each type.

One of the menus within the custom application provided the answer to the question of how to switch the active assembly parameterization with an entirely different parameterization of the same geometry. Section 5.2 explained that this was done using custom functions that performed the necessary neutralization and parameter exchange operations. This was done while avoiding typical CAD system errors and warnings that occur when manipulating interdependent parameters and assembly components.

The question of how the effects of manipulating driving parameters from multiple parameterizations should be observed and presented was answered in section 5.3. The custom application tracked the value of any desired feature or expression chosen by the engineer during parametric manipulation. The results of this automated observation were recorded in an output file that was formatted to be easily transferred to a spreadsheet or

presentation file for further comparison and review. This method was chosen for simplicity, and to allow compatibility and portability to the user.

The appropriate uses and drawbacks of the method presented in this thesis were proposed in section 5.4. To summarize this section, the custom application developed for this thesis is a program which enables exploration, manipulation, and observation of a parametric assembly model that has been previously unavailable. The application overcomes some of the limitations of parametric CAD systems when dealing with parametric assemblies. Some drawbacks of the method and application are that there is still a certain amount of up front consideration and cooperation that needs to take place when developing the initial “neutral” geometry. Also, this research has only tested the effectiveness of the custom application on relatively small assemblies. Very large assemblies, with several hundred or more components, may prove to be not well suited for the method and application presented here.

As set forth from the beginning of this research, the efforts discussed above were accomplished through the development of a custom application created to enhanced design exploration and collaborative efforts. In general terms, this thesis reinforced the idea that the use of the API of the CAD system allows engineers to perform customized tasks providing functionality not available in the CAD software alone. Through creative use of the built in API functions, and the development of new functions to perform data manipulation and monitoring of the activities of the CAD models, significant enhancement of the design exploration and collaborative engineering process is now possible through the methods and techniques set forth in this thesis.

7.1 Future Work

There are several things that could be done to build upon the work of this thesis. First of all, one of the main limitations that could be addressed, and could open up further possibilities, is the restriction on naming convention across each discipline involved in the development of parametric assembly models. It seems that the geometry would still need to remain identical, but there may be a way to utilize the CAD API to employ some sort of recognition between parameters and features across multiple parameterizations.

Another item that could be explored would be to design a way to implement this type of research into a batch optimization program. This would require modification of the current application to be able to operate in a batch mode. Also, the user inputs would have to be able to be incorporated into the optimization set up. There is some question as to whether this could become part of a fully automated optimization process, and this may be worth exploring in future research.

The other suggestions that I would have for future work include mostly enhancements to the functionality and usability of the interface of the custom application. These would include such things as graphical selection of features or geometry and providing more user control over the output of the program.

A final recommendation for further testing and expansion of this research is to implement the methods described here in a professional environment. This would truly test the feasibility of the proposals in real world scenarios, and the collaboration required for successful design exploration.

REFERENCES

Ardalan, S., (2000), "DrawCraft: A Spacecraft Design Tool for Integrated Concurrent Engineering," *Aerospace Conference Proceedings*, IEEE, Vol. 11, pp. 501-510.

Bidarra, R., Kranendonk, N., Noort, A., Bronsvoort, W. F. (2002), "A collaborateive framework for integrated part and assembly modeling", *Proceedings of the Symposium on Solid Modeling and Applications,* 2002, p 389-400.

Cheng, F. Y., and Li, X. S., (1999), "Generalized Center Method for Multiobjective Engineering Optimization", *Engineering Optimization*, v 31, p 641-661.

Delap, D., *CAD Based Creation and Optimization of a Gas Turbine Flowpath Module with Multiple Parameterizations,* M.S. Thesis, Brigham Young University, 2002

Delap, D., Hogge D. and Jensen, C. G., "CAD-centric Creation and Optimization of a Gas Turbine Flowpath Module with Multiple Parameterizations," Computer-Aided Design and Applications, vol. 3, nos. 1-4, 2006, pp. 175-185.

Hoffman, C.M., Kim, K.J., (2000), "Towards Valid Parametric CAD Models", *Computer-Aided Design,* v 33, p81-90.

Hoffman, C. M., Sitharam, M., Yuan, B., (2004), "Making constraint solvers more usable: overconstraint problem", *Computer-Aided Design,* v 36, p 377-399.

Hogge, D., *Integrating Commercial CAx Software to Perform Multidisciplinary Design Optimization*, M.S. Thesis, Brigham Young University, 2002.

Hrbacek, K., Jech, T., (1999), Introduction to Set Theory, Marcel D.

Jensen, C.G., Jones, C., Rohm III, T. and Tucker, S., (2000), "Parametric engineering design tools and applications", *Proceedings of ASME Design Automation Conference*, Baltimore, MD, Sept. 10-13, p 657-664.

Kim, J., Soonhung, H., (2004), "Manipulating Geometry in a STEP DB from Commercial CAD Systems", *Concurrent Engineering: Research and Applications*, v 12, n 1, p 49-57.

King, M. L., *A CAD-Centric Approach To CFD Analysis With Discrete Features*, M.S. Thesis, Brigham Yound University, 2004

Ledermann, C., Claus, H., Wenzel, J., Ermanni, P., Kelm, R. (2005), "Associative parametric CAE methods in the aircraft pre-design", *Aerospace Science and Technology*, v 9, n 7, p 641 – 651.

Lee, J. Y., Kim, K., (1996), "Geometric reasoning for knowledge-based parametric design using graph representation", *Computer- Aided Design,* v 28, n 10, p 831-841.

Lee, K., (1999), Principles of CAD/CAM/CAE Systems, Addison Wesley Longman, Inc.

Levy, L. S., (1980), Discrete Structures of Computer Science, John Wiley & Sons.

Myung S., Han S., (2001), "Knowledge-based Parametric Design of Mechanical Products Based on Configuration Design Method", *Expert Systems with Applications,* v 21, n 2, August, 2001, p 99-107.

Srinivasan, H., Jensen, C. G., Kopper, F. C., Staubach, J. B., Pack, D. R., (2001), "Assembly Parametrics for Multidisciplinary Design Optimization", *ISPE International Conference on Concurrent Engineering*

Townsend, J. C., Samareh, J.A., Weston, R. P., Zorumski, W. E., (1998), "Integration of a CAD System into an MDO Framework", *NASA/TM-207672,* May 1998.

Tucker, S., Rohm III, T., Jensen, C. G., "Concurrent Engineering with Parametric Design", *Proceedings of the 3rd World Conference on Intelligent Manufacturing of Processes and Systems,* Cambridge, MA, June 28-30.

Wan, S. Shin, R. A., (1991), "Interactive Multiple Objective Optimization: Survey 1-Continuous Case",

Wang, J.H., Wu, J.K., (1994), "An Assembly Constraint Model For Parameterized Mechanical Systems", *ASME Database Symposium*, *Engineering Data Management: Integrating the Engineering Enterprise*, 1994, p 67-73.

Wilson, M. G., Integration Of Rapid Prototyping Preprocessing Operations With A Commercial CAD System, M.S. Thesis, Brigham Young University, 2004

Youngjun, K., (2003), "Cad Model Assembly Hierarchy Reorganization for Application in Virtual Assembly – a Hybrid Approach Using the CAD System and a Visualization Tool", *Proceedings of the ASME Design Engineering Technical Conference,* v 1b, 2003, p. 1173 - 1181.

Zeid, I., (2005), Mastering CAD/CAM, McGraw-Hill.

Appendix A: Example Expression files

mixerSTRUCT.exp:

```
DATUM_OFFSET=MIXER_LENGTH
FWD_RADIUS=18
HOLE_LEN=1.75
HOLE_WIDTH=0.75
L2HOLE_CENTER=10
LOBE_FLAT=HOLE_WIDTH/2
LOBE_LOW_RAD=0.75
LOBE_UP_RAD=1
LOW_LOBE_ANGLE=20
MIXER_LENGTH=16
TIP_RADIUS=21
UPPER_STRAIGHT=4
UPPER_TRANS=8
connector_diameter=TIP_RADIUS*2
cut_top=TIP_RADIUS-1
p0=MIXER_LENGTH
p1=UPPER_STRAIGHT
p2=FWD_RADIUS
p3=6
p6=-10
p19=3.59224793597646
p29=0.5
p31=1.5
p32=1.75
p44=FWD_RADIUS
p52=0.025
p57=0.08
p58=0.125
p59=0.1875
p60=1
p61=1
p62=-10
p63=10
p64=0.05
```

p66=90
p70=0
p80=10
p86=TIP_RADIUS
p87=UPPER_TRANS
p88=MIXER_LENGTH
p89=UPPER_STRAIGHT
p90=FWD_RADIUS
p91=UPPER_STRAIGHT

supportbracketSTRUCT.exp:

thickness=0.125
axial_start=10
fwd_rad=Tailcone::fwd_rad
straight_profile=Tailcone::flange_length
radius=64
axial_end=18
p12=-10
p13=10
pocket_datum=fwd_edge+length
p15=0.125
p16=0.05
p17=pocket_width/2
pocket_width=2.0
p19=0
p20=0
p21=pocket_length+thickness
p22=thickness
p41=p38/2
p23=thickness+0.125
p24=thickness
p25=1.5
p26=0
p27=0
pocket_length=4.5
p29=fillet
p38=2.5
p32=fillet
fillet=0.125
p33=fillet+thickness
p34=0.125
p35=0.125
p37=0.25
p43=0.125

```
length=Tailcone::hole_len
width=Tailcone::hole_width
fwd_edge=Tailcone::len_to_hole
p47=3
p48=3
p49=3
p50=3
p51=0
p54=0.125
```

Appendix B: Component.cpp program file

```
//-----------------------------------------------------------
//
// File:  Component.cpp
//
// Description:
//         Declaration of the base Component class.  Other
//         Components will be derived from this class.
//
// Author: Brady Larson
//
//-----------------------------------------------------------

#include "Component.hpp"

Component::Component(void)
{

}

void Component::ImportExp(char* file, tag_t part_tag)
{
  // NEED TO CHECK IMPORT FILE AGAINST CURRENT EXPS
  // TO SEE IF ANY NEW EXPS NEED TO BE CREATED TO MAKE
  // IMPORT POSSIBLE (IF EVEN ONE DOESN'T EXIST, IT WON'T IMPORT
ANYTHING AT ALL)

  /*vector<char*> names;
  int num_exps;
        tag_t *exps;
  UF_MODL_ask_exps_of_part(part_tag, &num_exps, &exps);
        for(int i=0; i<num_exps; i++)
        {
    char* exp_string = new char[128];
                UF_MODL_ask_exp_tag_string(exps[i], &exp_string);
    char* name = strtok(exp_string,"=");
    names.push_back(name);
```

```
      UF_free(exp_string);
    }

    char buf[256];
    FILE* fp = fopen(file,"r");
    if(fp==NULL) {
      printf("ERROR opening expression file \n");
      return;
    }
    while(fgets(buf,sizeof(buf),fp)!=NULL)
    {
      char* token = strtok(buf,"=");
      if(token==NULL)
        continue;
      int found=0;
      for(i=0;i<num_exps;i++) {
        if(strcmp(token,names[i])==0) {
          found = 1;
          break;
        }
      }

      if(found==0) {
        // expression exists in part

      } else {
        char left[256];
        strcpy(left,token);
        token = strtok(NULL, "=\n");
        char new_exp[256];
        sprintf(new_exp,"%s=%s", left, token);
        UF_MODL_create_exp(new_exp);
        cout << "CREATED: " << new_exp << endl;
      }

    }
    fclose(fp);
    */

    UF_MODL_import_exp(file, 0);

}

void Component::ExportExp(char* file)
{
        UF_MODL_export_exp(file);
```

```
}

void Component::DerefExp(tag_t part_tag)
{
        double exp_value;
        int num_exps;
        tag_t *exps;

        UF_MODL_ask_exps_of_part(part_tag, &num_exps, &exps);
        for(int i=0; i<num_exps; i++)
        {
    char* exp_string = new char[128];
    char* new_string = new char[128];
                UF_MODL_ask_exp_tag_value(exps[i], &exp_value);
                UF_MODL_ask_exp_tag_string(exps[i], &exp_string);
                new_string = RewriteExp(exp_string, exp_value);
    UF_MODL_edit_exp(new_string);

    UF_free(exp_string);
    delete [] new_string;

  }
}

void Component::CleanupExps(tag_t part_tag)
{
        int num_exps;
        tag_t *exps;
  int exps_deleted = 0;

        UF_MODL_ask_exps_of_part(part_tag, &num_exps, &exps);
        for(int i=0; i<num_exps; i++)
        {
    //ask exp tag string, dissect string, delete exp, output
    char* exp_string = new char[128];
    char* exp_name = new char[128];
    char* exp_val = new char[128];
    tag_t* out_tag;
                UF_MODL_ask_exp_tag_string(exps[i], &exp_string);
                UF_MODL_dissect_exp_string (exp_string, &exp_name, &exp_val,
out_tag);

    if(UF_MODL_delete_exp(exp_name)==NULL)
    {
      exps_deleted++;
    }
```

```
     /*UF_free(exp_string);
     UF_free(exp_name);
     UF_free(exp_val);
     */
  }

  cout<< "Number of expressions deleted in " << GetName() << ": " << exps_deleted <<
endl;

}

char* Component::GetName()
{
        return mName;
}

void Component::SetName(char* name)
{
        mName = strdup(name);
}

void Component::SetPartTag(tag_t tag)
{
        mPrtTag = tag;
}

tag_t Component::GetPartTag()
{
        return mPrtTag;
}

void Component::SetExportPath(char* path)
{
        mExportPath = strdup(path);
}

char* Component::GetExportPath()
{
        return mExportPath;
}

void Component::SetImportPath(char* path)
{
        mImportPath = strdup(path);
}
```

```
char* Component::GetImportPath()
{
	return mImportPath;
}

Component::~Component()
{

}
```

Appendix C: dSpace.cpp program file

```
#include "UIStylerFuncs.hpp"
#include <stdio.h>
#include <uf.h>
#include <uf_defs.h>
#include <uf_exit.h>
#include <uf_ui.h>
#include <uf_styler.h>
#include <uf_mb.h>
#include <dSpace.h>
#include "FilestringDecomp.hpp"
#include "Component.hpp"
#include <fstream>
#include <iostream>
#include <string>
#include "expList.h"
#include "workPart.h"


using namespace std;

vector<Component*> Assembly;
char* save_key;
char* new_key;
tag_t root_part;
char root_name[256 + 1];
tag_t root_occ;
tag_t* child_part_occs;

char* pVary1;
char* pObserve1;
tag_t* feature_tag;
int number_of_exps = 0;
int num_parts;
tag_t *exps;
char* pVary2;
char* pObserve2;
```

```
vector<char**> ObsExpArray;
vector<char**> ObsExpArray2;
bool ObserveFeature1 = false;
bool ObserveFeature2 = false;
vector<char*> CompNames;
tag_t obs1Part_tag;
tag_t vary1Part_tag;
tag_t obs2Part_tag;
tag_t vary2Part_tag;

/* The following definition defines the number of callback entries */
/* in the callback structure:                                    */
/* UF_STYLER_callback_info_t CDS_cbs */
#define CDS_CB_COUNT ( 14 + 1 ) /* Add 1 for the terminator */

/*------------------------------------------------------------------------
The following structure defines the callback entries used by the
styler file.  This structure MUST be passed into the user function,
UF_STYLER_create_dialog along with CDS_CB_COUNT.
------------------------------------------------------------------------*/
static UF_STYLER_callback_info_t CDS_cbs[CDS_CB_COUNT] =
{
 {UF_STYLER_DIALOG_INDEX, UF_STYLER_CONSTRUCTOR_CB  , 0,
CDS_constructor_cb},
 {CDS_CHANGE_WORK_1     , UF_STYLER_ACTIVATE_CB     , 1,
CDS_CHANGE_WORKPART_CB},
 {CDS_P1_OBSERVE_BUTTON , UF_STYLER_ACTIVATE_CB     , 1,
CDS_P1_OBSERVE_CB},
 {CDS_P1_VARY_BUTTON    , UF_STYLER_ACTIVATE_CB     , 1,
CDS_P1_VARY_CB},
 {CDS_CALC_1            , UF_STYLER_ACTIVATE_CB     , 0, CDS_P1_CALC_CB},
 {CDS_CHANGE_WORK_1_P_0 , UF_STYLER_ACTIVATE_CB     , 1,
CDS_CHANGE_WORKPART_CB},
 {CDS_SAVE_TOGGLE       , UF_STYLER_VALUE_CHANGED_CB, 0,
CDS_save_toggle_cb},
 {CDS_SAVE_CURRENT_BUTTON, UF_STYLER_ACTIVATE_CB     , 0,
CDS_SAVE_CURRENT_CB},
 {CDS_BROWSE_BUTTON     , UF_STYLER_ACTIVATE_CB     , 0,
CDS_BROWSE_CB},
 {CDS_SWAP_BUTTON       , UF_STYLER_ACTIVATE_CB     , 0,
CDS_SWAP_CB},
 {CDS_CHANGE_WORK_3     , UF_STYLER_ACTIVATE_CB     , 1,
CDS_CHANGE_WORKPART_CB},
 {CDS_P2_OBSERVE_BUTTON , UF_STYLER_ACTIVATE_CB     , 1,
CDS_P2_OBSERVEIT_CB},
```

```
 {CDS_P2_VARY_BUTTON    , UF_STYLER_ACTIVATE_CB     , 1,
CDS_P2_VARY_CB},
 {CDS_CALC_2            , UF_STYLER_ACTIVATE_CB     , 0, CDS_P2_CALC_CB},
 {UF_STYLER_NULL_OBJECT, UF_STYLER_NO_CB, 0, 0 }
};

static UF_MB_styler_actions_t actions[] = {
   { "dSpace.dlg",  NULL,   CDS_cbs,  UF_MB_STYLER_IS_NOT_TOP },
   { NULL,  NULL,  NULL,  0 } /* This is a NULL terminated list */
};


extern void ufsta (char *param, int *retcode, int rlen)
{
   int  response   = 0;
   int  error_code = 0;

   if ( ( UF_initialize() ) != 0 )
        return;

   if ( ( error_code = UF_STYLER_create_dialog ( "dSpace.dlg",
        CDS_cbs,      /* Callbacks from dialog */
        CDS_CB_COUNT, /* number of callbacks*/
        NULL,        /* This is your client data */
        &response ) ) != 0 )
   {
       char fail_message[133];

       /* Get the user function fail message based on the fail code.*/
       UF_get_fail_message(error_code, fail_message);
                 UF_UI_set_status (fail_message);
       printf ( "%s\n", fail_message );
   }


   UF_terminate();
   return;
}




extern int ufusr_ask_unload (void)
{
    /* unload immediately after application exits*/
```

```
    return ( UF_UNLOAD_IMMEDIATELY );

    /*via the unload selection dialog... */
    /*return ( UF_UNLOAD_SEL_DIALOG );   */
    /*when UG terminates...              */
    /*return ( UF_UNLOAD_UG_TERMINATE ); */
}



extern void ufusr_cleanup (void)
{
   return;
}






/*----------------------------------------------------------------------*/
/*---------------------- UIStyler Callback Functions ----------------------*/
/*----------------------------------------------------------------------*/

/* ----------------------------------------------------------------------
 * Callback Name: CDS_constructor_cb
int CDS_constructor_cb ( int dialog_id,
           void * client_data,
           UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

    ////////////////////////////////////////

           char *file_name = new char[132];
           char *file_path = new char[132];

           Component* root = new Component();
    root_part = UF_PART_ask_display_part ( );
           root->SetPartTag(root_part);
           UF_PART_ask_part_name(root_part, root_name);
           FilestringDecomp(root_name,file_name, file_path);
           root->SetName(file_name);
           Assembly.push_back(root);
```

```
	cout << "Root Part: " << root_name << endl;

	root_occ = UF_ASSEM_ask_root_part_occ(root_part);

	Component* comp;
	num_parts = UF_ASSEM_ask_part_occ_children(root_occ, &child_part_occs);
	for(int i=0; i<num_parts; i++)
	{
		comp = new Component();
		comp-
>SetPartTag(UF_ASSEM_ask_prototype_of_occ(child_part_occs[i]));
		char name[132];
		UF_PART_ask_part_name(comp->GetPartTag(), name);
		FilestringDecomp(name, file_name, file_path);
		comp->SetName(file_name);
		Assembly.push_back(comp);

	}

	for(i=0; i<num_parts+1; i++)
	{
		cout << "Tag" << i << ": " << Assembly[i]->GetPartTag() << endl;
		cout << "Name" << i << ": " << Assembly[i]->GetName() << endl <<
endl;
		CompNames.push_back(Assembly[i]->GetName());
	}

	////////////////////////////////////////////
    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);
   /* A return value of UF_UI_CB_EXIT_DIALOG will not be accepted    */
   /* for this callback type.  You must continue dialog construction.*/

}

/* ----------------------------------------------------------------------
 * Callback Name: CDS_CHANGE_WORKPART_CB
int CDS_CHANGE_WORKPART_CB ( int dialog_id,
	void * client_data,
	UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );
```

```
    /* ---- Enter your callback code here ----- */
            //launch dialog which lists all components of the assembly
            //send the function an array of char*'s (component names)
            //return the selected char*
            //retrieve part tag of the chosen part
            //set the new work part
            //update the model?
            //continue dialog

            char* workPart = changeWorkPart(CompNames, num_parts);

            tag_t new_work_tag = UF_PART_ask_part_tag (workPart);

            UF_ASSEM_set_work_part (new_work_tag);
            UF_MODL_update();




    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}


/* -------------------------------------------------------------------------
 * Callback Name: CDS_save_toggle_cb
int CDS_save_toggle_cb ( int dialog_id,
            void * client_data,
            UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

            ////////////////////////////////////////////

            int checked = UIS_getIntValue(dialog_id, CDS_SAVE_TOGGLE);

            if(checked == 0)
            {
```

```
                UIS_setSingleSens(dialog_id, CDS_SAVE_CURRENT_BUTTON,
false);
                UIS_setSingleSens(dialog_id, CDS_SAVE_KEYWORD, false);
        }
        else
        {
                UIS_setSingleSens(dialog_id, CDS_SAVE_CURRENT_BUTTON, true);
                UIS_setSingleSens(dialog_id, CDS_SAVE_KEYWORD, true);
        }

        ////////////////////////////////////////////



    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.  */
   /* return ( UF_UI_CB_EXIT_DIALOG );            */

}


/* ----------------------------------------------------------------------
 * Callback Name: CDS_save_current_cb
int CDS_SAVE_CURRENT_CB ( int dialog_id,
        void * client_data,
        UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

    ////////////// My Code //////////////////

    char *filter = new char[132];
        strcpy(filter, "*.exp");
        char *path_and_name = new char[132];
        char *file_name = new char[132];
        char *file_path = new char[132];
        int response=0;

        //prompt for user file selection
```

```
            int rc = UF_UI_create_filebox("Browse...", "Browse...", filter, "default.exp",
path_and_name, &response);
            if(rc==0 && response==UF_UI_OK)
            {
                        //FilestringDecomp(path_and_name, file_name, file_path);
                        FilestringDecomp(path_and_name, file_name, file_path);

                        for(int i=0; i<Assembly.size(); i++)
                        {
                                    Assembly[i]->SetExportPath(file_path);
                        }
            }
            else if(response==UF_UI_CANCEL); //continue like normal
            else  uc1601("Unable to create file selection box.", 1);

            ////////////////////////////////////////////

    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}


/* -------------------------------------------------------------------------
 * Callback Name: CDS_p1_observe_cb
int CDS_P1_OBSERVE_CB ( int dialog_id,
            void * client_data,
            UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

          /* THIS CALLBACK WILL STORE DATA REGARDING THE
EXPRESSIONS, FEATURES, OR COMPONENTS
                    CHOSEN BY THE USER.  THE DATA WILL BE STORED SO THAT
IT CAN BE EASILY EXPORTED AND
                    FORMATTED FOR LATER REVIEW. */

          char* cue;
          cue = "Select a features to observe during expression variation.";
```

```
   char* title;
         title = "Feature or Expression to observe";
   int response, count;
         response = 0;
         int i, j;
         void* filter;
         filter = NULL;
         string toggle_label = "nothing yet";

         UF_STYLER_item_value_type_t data;
   int irc = 0;
         data.item_id = "FEAT_EXP_TOGGLE";
         /* Set the item id */
   data.item_attr = UF_STYLER_VALUE;
   irc = UF_STYLER_ask_value ( dialog_id, &data );
         /* Ask for the info*/
         int val = data.value.integer;
         if (val == 1)
                toggle_label = "Expression";
         if (val == 0)
                toggle_label = "Feature";


         if(toggle_label == "Feature")
         {
                UF_CALL(UF_UI_select_feature(cue, filter, &count, &feature_tag,
&response ));
                if( response == UF_UI_OK && feature_tag != NULL)
                {

                 cout << "Feature count = "<< count << endl;
                        for (i=0; i<count; i++)
                        {
                               UF_MODL_ask_exps_of_feature (feature_tag[i],
&number_of_exps, &exps );
                               UF_MODL_ask_feat_name (feature_tag[i], &pObserve1 );

                               cout << "Expressions for the feature: "<< pObserve1 <<
endl;
                               char* exp_string;
                               for(j=0;j<number_of_exps;j++)
                               {
                                      UF_MODL_ask_exp_tag_string (exps[j],
&exp_string);
                                      cout << exp_string << endl;
                               }
```

```
                        }
                }
                // SET LABEL SHOWING THE NAME OF THE SELECTED
FEATURE
                UIS_setLabel(dialog_id, "SELECTED_EXP_LABEL", pObserve1);


                //UF_free(feature_tag);
                ObserveFeature1 = true;

        }

        if(toggle_label == "Expression")
        {

         pObserve1 = getPartExps();

         cout << "pObserve1 = " << pObserve1 << endl;

         UIS_setLabel(dialog_id, "SELECTED_EXP_LABEL", pObserve1);
         //UF_free(exp_tags);
        }

        obs1Part_tag = UF_ASSEM_ask_work_part ( );

   UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}


/* ----------------------------------------------------------------------
 * Callback Name: CDS_p1_vary_cb
int CDS_P1_VARY_CB ( int dialog_id,
             void * client_data,
             UF_STYLER_item_value_type_p_t callback_data)
{
     /* Make sure User Function is available. */
     if ( UF_initialize() != 0)
          return ( UF_UI_CB_CONTINUE_DIALOG );
```

```
        pVary1 = getPartExps();

        cout << "pVary1 = " << pVary1 << endl;

        UIS_setLabel(dialog_id, "SELECTED_EXP_LABEL_P_13", pVary1);

    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}



/* ----------------------------------------------------------------------
 * Callback Name: CDS_P1_CALC_CB
int CDS_P1_CALC_CB ( int dialog_id,
        void * client_data,
        UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

        ////////////////////////////////// MY CODE ////////////////////////////

        // PSEUDO CODE:
        // GET EXPRESSION TO VARY, AND EXPRESSION/FEATURE TO
OBSERVE
        // GET LIMITS
        // VARY SELECTED EXPRESSION AND ASK FOR VALUE OF
EXPRESSION(S) AT EACH INCREMENT (LOOP)
        // STORE OBSERVED VALUES IN AN ARRAY, OR GROUP OF ARRAYS
        // POSSIBLY OUTPUT VALUES TO A TEXT OR CSV FILE

        double lowerBound, upperBound;
        int numSteps = 0;
        tag_t p1_tag;
        double p1_val;
        char** lhs_str = new char*[1];
        char** rhs_str = new char*[1];
```

```
        lowerBound = UIS_getRealValue(dialog_id, "LOWLIMITENTRY");
        upperBound = UIS_getRealValue(dialog_id, "UPLIMITENTRY");
        numSteps = UIS_getIntValue(dialog_id, "DRAG_SCALE_1");

        cout << "Limits: " << lowerBound << ", " << upperBound << endl;

        UF_CALL(UF_MODL_dissect_exp_string (pVary1, lhs_str, rhs_str, &p1_tag ));
        UF_MODL_ask_exp_tag_value (p1_tag, &p1_val);

        cout << "Expression value: " << p1_val << endl;

        if (p1_val < lowerBound || p1_val > upperBound)
        {
                char* title_string = "User Definition Error";
                UF_UI_MESSAGE_DIALOG_TYPE dialog_type =
UF_UI_MESSAGE_WARNING;
                char** messages = new char*[1];
                *messages = "The current value of the expression to vary is outside of the
specified limits.";
                int num_messages = 1;
                logical translate = false;
                UF_UI_message_buttons_t* buttons = NULL;
                int* response;

                UF_UI_message_dialog (title_string, dialog_type, messages,
num_messages, translate, buttons, response );
        }
        // ASK FOR NUMBER OF INCREMENTS FROM USER???
        double testVal = lowerBound;
        double inc = (upperBound - lowerBound)/numSteps;

        for(int i=0;i<numSteps+1;i++)
        {
                if (i != 0)
                        testVal = testVal + inc; //add incremental amount to rhs
                char* newExp = new char[256];
                sprintf(newExp,"%s=%lf",*lhs_str,testVal);//update expression
                cout << "Modified expression: " << newExp << endl;
                UF_CALL(UF_MODL_edit_exp(newExp));  //edit expression
                UF_MODL_update( );  //update model

                vary1Part_tag = UF_ASSEM_ask_work_part ( );
                //gather observed values of the expression(s) to be observed
                if(ObserveFeature1 == true)
                {
```

```
				UF_ASSEM_set_work_part_quietly (obs1Part_tag, 
&vary1Part_tag);
				char** Observe1Exps = new char*[50];
				UF_MODL_ask_exps_of_feature (feature_tag[i], 
&number_of_exps, &exps );
					// NEEDED???   UF_MODL_ask_feat_name 
(feature_tag[i], &pObserve1 );

					char* exp_string;
					for(int j=0;j<number_of_exps;j++)
					{
						UF_MODL_ask_exp_tag_string (exps[j], 
&exp_string);
						cout << exp_string << endl;
						//store expression
						Observe1Exps[j] = exp_string;
					}
				UF_ASSEM_set_work_part_quietly (vary1Part_tag, 
&obs1Part_tag);

			ObsExpArray.push_back(Observe1Exps);
			}
			else
			{
				//get expression to observe and store value in observe array
				char** Observe1Exps = new char*[50];
				UF_ASSEM_set_work_part_quietly (obs1Part_tag, 
&vary1Part_tag);

				char ** lhs_str = new char*[1];
				char ** rhs_str = new char*[1];
				tag_t  exp_tag;
				double exp_value;
				char* numeric_exp;

				UF_MODL_dissect_exp_string (pObserve1, lhs_str, rhs_str, 
&exp_tag );

				UF_MODL_ask_exp_tag_value (exp_tag, &exp_value);

				numeric_exp = RewriteExp(pObserve1, exp_value);

				Observe1Exps[0] = numeric_exp;
				number_of_exps = 1;
				UF_ASSEM_set_work_part_quietly (vary1Part_tag, 
&obs1Part_tag);
```

```
                ObsExpArray.push_back(Observe1Exps);
                }

        }

        //TEST WRITING OUTPUT TO FILE:
        ofstream f("Output.txt",ios::app);
        for(i=0;i<numSteps+1;i++)
        {
                for(int j=0;j<number_of_exps;j++)
                {
                        f<<ObsExpArray[i][j] << "\t";
                }
                f<<endl;
        }

        //UF_free (lhs_str);
        //UF_free (rhs_str);

    UF_terminate ();

    /* Callback acknowledged, do not terminate dialog */
    return (UF_UI_CB_CONTINUE_DIALOG);

    /* or Callback acknowledged, terminate dialog.    */
    /* return ( UF_UI_CB_EXIT_DIALOG );              */

}


/* ----------------------------------------------------------------------
 * Callback Name: CDS_browse_cb
int CDS_BROWSE_CB ( int dialog_id,
          void * client_data,
          UF_STYLER_item_value_type_p_t callback_data)
{
     /* Make sure User Function is available. */
     if ( UF_initialize() != 0)
          return ( UF_UI_CB_CONTINUE_DIALOG );

     //////////////////////////////////////////////////

          char *filter = new char[132];
          strcpy(filter, "*.exp");
          char *path_and_name = new char[132];
          char *file_name = new char[132];
```

```
	char *file_path = new char[132];
	int response=0;

	//prompt for user file selection
	int rc = UF_UI_create_filebox("Browse...", "Browse...", filter, "default.exp",
path_and_name, &response);
	if(rc==0 && response==UF_UI_OK)
	{
		FilestringDecomp(path_and_name, file_name, file_path);
		cout << "Import path and name: " << path_and_name << endl;
		for(int i=0; i<Assembly.size(); i++)
		{
			Assembly[i]->SetImportPath(file_path);
		}
	}
	else if(response==UF_UI_CANCEL); //continue like normal
	else  uc1601("Unable to create file selection box.", 1);

	//////////////////////////////////////////////////////


    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}


/* -------------------------------------------------------------------------
 * Callback Name: CDS_swap_cb
int CDS_SWAP_CB ( int dialog_id,
	void * client_data,
	UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

    //    CHECK TO SEE IF SAVING CURRENT ASSEMBLY EXPS IS DESIRED
	// IF SO, EXPORT EXP FILES TO THE CHOSEN DIRECTORY AND
KEYWORD
	int checked = UIS_getIntValue(dialog_id, CDS_SAVE_TOGGLE);
```

```
        if(checked == 1)
        {
                save_key = UIS_getStringValue(dialog_id, CDS_SAVE_KEYWORD);
                char* key_and_ext = strcat(save_key, ".exp");
                for(int i=0; i<Assembly.size(); i++)
                {
                        char* path = Assembly.at(i)->GetExportPath();
                        char* name = Assembly.at(i)->GetName();
                        strcat(path, name);
                        char* export_file = strcat(path, key_and_ext);
                        UF_ASSEM_set_work_part(Assembly[i]->GetPartTag());
                        Assembly[i]->ExportExp(export_file);
                }
                UF_ASSEM_set_work_part(Assembly[0]->GetPartTag());
    cout << "exit Export " << endl;
        }

        //      GET PATH TO IMPORT FILES FROM AND IMPORT NEW
EXPRESSIONS
        new_key = UIS_getStringValue(dialog_id, CDS_KEYWORD);
   char* ext = ".exp";
        char* new_and_ext = joinStrings(new_key, ext);
        for(int i=0; i<Assembly.size(); i++)
        {
                char* path = Assembly[i]->GetImportPath();
                char* name = Assembly[i]->GetName();
                char* pathName = joinStrings(path, name);
                char* import_file = joinStrings(pathName, new_and_ext);
                cout << "Import file: " << import_file << endl;
                UF_ASSEM_set_work_part(Assembly[i]->GetPartTag());
                Assembly[i]->DerefExp(Assembly[i]->GetPartTag());
                Assembly[i]->ImportExp(import_file, Assembly[i]->GetPartTag());
        }
   UF_ASSEM_set_work_part(Assembly[0]->GetPartTag());

        UF_MODL_update();

   //  CHECK FOR UNUSED EXPRESSIONS BY ATTEMPTING TO DELETE EXPS.
   /*for(i=0; i<Assembly.size(); i++)
        {
                Assembly[i]->CleanupExps(Assembly[i]->GetPartTag());
        }
        */


   UF_terminate ();
```

```
   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}


/* ----------------------------------------------------------------------
 * Callback Name: CDS_p2_observe_cb
int CDS_P2_OBSERVEIT_CB ( int dialog_id,
           void * client_data,
           UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

        /* THIS CALLBACK WILL STORE DATA REGARDING THE
EXPRESSIONS, FEATURES, OR COMPONENTS
                CHOSEN BY THE USER.  THE DATA WILL BE STORED SO THAT
IT CAN BE EASILY EXPORTED AND
                FORMATTED FOR LATER REVIEW. */

        char* cue;
        cue = "Select a features to observe during expression variation.";
   char* title;
        title = "Feature or Expression to observe";
   int response, count;
        response = 0;
        int i, j;
        void* filter;
        filter = NULL;
        string toggle_label = "nothing yet";

        UF_STYLER_item_value_type_t data;
   int irc = 0;
        data.item_id = "P2_FEAT_EXP_TOGGLE";
        /* Set the item id */
   data.item_attr = UF_STYLER_VALUE;
   irc = UF_STYLER_ask_value ( dialog_id, &data );
        /* Ask for the info*/
        int val = data.value.integer;
        if (val == 1)
                toggle_label = "Expression";
```

```
	if (val == 0)
		toggle_label = "Feature";


	if(toggle_label == "Feature")
	{
		UF_CALL(UF_UI_select_feature(cue, filter, &count, &feature_tag,
&response ));
		if( response == UF_UI_OK && feature_tag != NULL)
		{

		 cout << "Feature count = "<< count << endl;

			for (i=0; i<count; i++)
			{
				UF_MODL_ask_exps_of_feature (feature_tag[i],
&number_of_exps, &exps );
				UF_MODL_ask_feat_name (feature_tag[i], &pObserve2 );

				cout << "Expressions for the feature: "<< pObserve2 <<
endl;
				char* exp_string;
				for(j=0;j<number_of_exps;j++)
				{
					UF_MODL_ask_exp_tag_string (exps[j],
&exp_string);
					cout << exp_string << endl;
				}
			}
		}
		// SET LABEL SHOWING THE NAME OF THE SELECTED
FEATURE
		UIS_setLabel(dialog_id, "SELECTED_EXP_LABEL_P_5", pObserve2);


		//UF_free(feature_tag);
		ObserveFeature2 = true;

	}

	if(toggle_label == "Expression")
	{
	 pObserve2 = getPartExps();

	 cout << "pObserve2 = " << pObserve2 << endl;
```

```
          UIS_setLabel(dialog_id, "SELECTED_EXP_LABEL_P_5", pObserve2);
          //UF_free(exp_tags);
          }

          obs2Part_tag = UF_ASSEM_ask_work_part ( );

   UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}

/* ----------------------------------------------------------------------
 * Callback Name: CDS_p2_vary_cb
 * This is a callback function associated with an action taken from a
 * UIStyler object.
 *
 * Input: dialog_id   -   The dialog id indicate which dialog this callback
 *                    is associated with.  The dialog id is a dynamic,
 *                    unique id and should not be stored.  It is
 *                    strictly for the use in the UG/Open API:
 *                          UF_STYLER_ask_value(s)
 *                          UF_STYLER_set_value
 *       client_data -   Client data is user defined data associated
 *                    with your dialog.  Client data may be bound
 *                    to your dialog with UF_MB_add_styler_actions
 *                    or UF_STYLER_create_dialog.
 *       callback_data - This structure pointer contains information
 *                    specific to the UIStyler Object type that
 *                    invoked this callback and the callback type.
 * -------------------------------------------------------------------*/
int CDS_P2_VARY_CB ( int dialog_id,
          void * client_data,
          UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

          pVary2 = getPartExps();

          cout << "pVary2 = " << pVary2 << endl;
```

```
 UIS_setLabel(dialog_id, "SELECTED_EXP_LABEL_P_17", pVary2);

    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}


/* -------------------------------------------------------------------------
 * Callback Name: CDS_P2_CALC_CB
int CDS_P2_CALC_CB ( int dialog_id,
             void * client_data,
             UF_STYLER_item_value_type_p_t callback_data)
{
    /* Make sure User Function is available. */
    if ( UF_initialize() != 0)
        return ( UF_UI_CB_CONTINUE_DIALOG );

    //////////////////////////////////// MY CODE //////////////////////////////

           double lowerBound, upperBound;
           int numSteps = 0;
           tag_t p2_tag;
           double p2_val;
           char** lhs_str = new char*[1];
           char** rhs_str = new char*[1];


           lowerBound = UIS_getRealValue(dialog_id, "LOWLIMITENTRY_P_8");
           upperBound = UIS_getRealValue(dialog_id, "UPLIMITENTRY_P_9");
           numSteps = UIS_getIntValue(dialog_id, "DRAG_SCALE_2");

           cout << "Limits: " << lowerBound << ", " << upperBound << endl;

           UF_CALL(UF_MODL_dissect_exp_string (pVary2, lhs_str, rhs_str, &p2_tag ));
           UF_MODL_ask_exp_tag_value (p2_tag, &p2_val);

           cout << "Expression value: " << p2_val << endl;

           if (p2_val < lowerBound || p2_val > upperBound)
           {
```

```
			char* title_string = "User Definition Error";
			UF_UI_MESSAGE_DIALOG_TYPE dialog_type =
UF_UI_MESSAGE_WARNING;
			char** messages = new char*[1];
			*messages = "The current value of the expression to vary is outside of the
specified limits.";
			int num_messages = 1;
			logical translate = false;
			UF_UI_message_buttons_t* buttons = NULL;
			int* response;

			UF_UI_message_dialog (title_string, dialog_type, messages,
num_messages, translate, buttons, response );
		}
		// ASK FOR NUMBER OF INCREMENTS FROM USER???
		double testVal = lowerBound;
		double inc = (upperBound - lowerBound)/numSteps;

		for(int i=0;i<numSteps+1;i++)
		{
			if (i != 0)
				testVal = testVal + inc; //add incremental amount to rhs
			char* newExp = new char[256];
			sprintf(newExp,"%s=%lf",*lhs_str,testVal);//update expression
			cout << "Modified expression: " << newExp << endl;
			UF_CALL(UF_MODL_edit_exp(newExp));  //edit expression
			UF_MODL_update( );  //update model

			vary2Part_tag = UF_ASSEM_ask_work_part ( );
			//gather observed values of the expression(s) to be observed
			if(ObserveFeature1 == true)
			{
				UF_ASSEM_set_work_part_quietly (obs2Part_tag,
&vary2Part_tag);
				char** Observe2Exps = new char*[50];
				UF_MODL_ask_exps_of_feature (feature_tag[i],
&number_of_exps, &exps );
					// NEEDED???   UF_MODL_ask_feat_name
(feature_tag[i], &pObserve1 );

					char* exp_string;
					for(int j=0;j<number_of_exps;j++)
					{
						UF_MODL_ask_exp_tag_string (exps[j],
&exp_string);
						cout << exp_string << endl;
```

```
					//store expression
					Observe2Exps[j] = exp_string;
				}
			UF_ASSEM_set_work_part_quietly (vary2Part_tag,
&obs2Part_tag);

		ObsExpArray2.push_back(Observe2Exps);
		}
		else
		{
			//get expression to observe and store value in observe array
			char** Observe2Exps = new char*[50];
			UF_ASSEM_set_work_part_quietly (obs2Part_tag,
&vary2Part_tag);

			char ** lhs_str = new char*[1];
			char ** rhs_str = new char*[1];
			tag_t  exp_tag;
			double exp_value;
			char* numeric_exp;

			UF_MODL_dissect_exp_string (pObserve2, lhs_str, rhs_str,
&exp_tag );

			UF_MODL_ask_exp_tag_value (exp_tag, &exp_value);

			numeric_exp = RewriteExp(pObserve2, exp_value);

			Observe2Exps[0] = numeric_exp;
			number_of_exps = 1;
			UF_ASSEM_set_work_part_quietly (vary2Part_tag,
&obs2Part_tag);

		ObsExpArray2.push_back(Observe2Exps);
		}

	}

	//TEST WRITING OUTPUT TO FILE:
	ofstream f("Output.txt",ios::app);

	f << endl << "Parameterization 2:" << endl;
	for(i=0;i<numSteps+1;i++)
	{
		for(int j=0;j<number_of_exps;j++)
		{
```

```
                        f<<ObsExpArray2[i][j] << "\t";
                }
                f<<endl;
        }

        //UF_free (lhs_str);
        //UF_free (rhs_str);

    UF_terminate ();

   /* Callback acknowledged, do not terminate dialog */
   return (UF_UI_CB_CONTINUE_DIALOG);

   /* or Callback acknowledged, terminate dialog.    */
   /* return ( UF_UI_CB_EXIT_DIALOG );               */

}



/////////////// ANY HOME GROWN FUNCTIONS ///////////////////


int round(double a) {
return int(a + 0.5);
}
```