High-level, Product Type-specific Programmatic Operations for Streamlining Associative Computer-aided Design

Nathan W. Scott
Brigham Young University - Provo

HIGH-LEVEL, PRODUCT TYPE-SPECIFIC PROGRAMMATIC OPERATIONS FOR STREAMLINING ASSOCIATIVE COMPUTER-AIDED DESIGN

by

Nathan W Scott

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

December 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Nathan W Scott

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

Date	C. Greg Jensen, Chair
Date	W. Edward Red
Date	Christopher A. Mattson

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate’s graduate committee, I have read the thesis of Nathan W Scott 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 materials 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

Larry L. Howell
Graduate Coordinator

Accepted for the College

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

ABSTRACT

HIGH-LEVEL, PRODUCT TYPE-SPECIFIC, PROGRAMMATIC OPERATIONS FOR STREAMLINING ASSOCIATIVE COMPUTER-AIDED DESIGN

Nathan W Scott

Department of Mechanical Engineering

Master of Science

Research in the field of Computer Aided Design (CAD) has long focused on reducing the time and effort required of engineers to define three dimensional digital product models. Parametric, feature-based modeling with inter-part associativity allows complex assembly designs to be defined and re-defined while maintaining the vital part-to-part interface relationships. The top-down modeling method which uses assembly level control structures to drive child level geometry has proved valuable in maintaining these interfaces. Creating robust parametric models like these, however, is very time consuming especially since there can be hundreds of features and thousands of mathematical expressions to create. Even if combinations of low-level features, known as User-Defined Features (UDFs), are used, this process still involves inserting individual features into individual components and creating all of the inter-part associativities by hand. This thesis shows that programmatic operations designed for a specific product type can streamline the assembly and component-level design process much further because a single programmatic operation can create an unlimited number of low-level features, modify geometry in multiple components, create new components, establish inter-part expressions, and define inter-part geometry links. Results from user testing show that a set of high-level programmatic operations can offer savings in time and effort of over 90% and can be general enough to support user-specified interface layouts and component cross sections while leaving the majority of the primary design decisions open to the engineer.

ACKNOWLEDGMENTS

I would like to thank Dr. Greg Jensen for his encouragement and advice and Dr. Ed Red and Dr. Christopher Mattson for their input. I thank my wife, Rebecca, for the sacrifices she has made to help me succeed. I also acknowledge Jesse Peterson, Jason Hamburg, and Dr. Angela Trego for their assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

The parametric, feature-based functionality within computer aided design (CAD) applications has increased engineering efficiency dramatically [Anderl, 1995 and Pritchard, 1996]. Well developed parametric models can be reused to produce many similar designs, and can simplify the process of incorporating design changes [Black, 1991 and Shaw, 1995]. In addition, associativity capabilities within modern CAD applications allow these parametric models to maintain relationships between features and components [Venkataraman, 2001] which extends the benefits to models of entire assemblies. There are also programming and scripting tools available that enable advanced users to programmatically perform most of the same interactive functions.

1.1 Problem statement

Creating robust parametric models is very time consuming and "this level of skill can take years of training and experience to acquire" [Delap, 2006 and Hoffmann, 2001]. Parametric models of assemblies are especially dificult and tedius to model because of the many inter-part relationships that must remain associative. Producing complex part models is less tedious and time-consuming when higher level features – features that combine several low-level features – are employed rather than relying on standard low-level features such as holes, ribs, and slots [Hoffmann, 1998 and Mosca, 2001]. Modern

CAD applications offer some high-level features for certain product types as well as User Defined Features (UDFs) which allow users to specify custom combinations of low-level features. These approaches to modeling assemblies are still restricted however because they do not have any inter-part associativity or intelligence capabilities and can only combine a limited number of low-level features. They require users to create the inter-part expressions and geometry links by hand and do not allow the user to operate on multiple components at once or to create new components as part of the operation.

1.2 Thesis objective

It is my objective to show how the current modeling practices described above can be streamlined further through the use of product type-specific programmatic operations that would combine the benefits of UDFs and inter-part associativity and would function at a level much higher than inserting single features into individual components. This thesis defines a programmatic operation as one that can create an unlimited number of low-level features, modify geometry in multiple components, create new components, establish inter-part expressions, and create inter-part geometry links. Since products of a similar type have similar primary and secondary features, programmatic operations can be written that can create most, if not all, of the CAD geometry necessary to define a product of a certain type. This will in affect reduce the time consuming modeling element to the designer's decision making time.

The specific test case that will be used to demonstrate and validate this research is the interstage sub-assembly on solid motor rockets. This particular assembly is a good candidate for this research because it is used repeatedly on a wide range of rockets and because its geometry is relatively simple. This research will therefore focus on the incorporation of rocket interstage-specific design operations into an application that will interoperate with Siemens NX 5. The goal of this study is to develop a set of proof-of-concept applications that will streamline the design of rocket assemblies and parts. The objectives of this thesis are:

1. Create a framework of intelligent, high-level, programmatic operations that can be used to quickly design a wide range of components and assemblies.
2. Implement this framework specifically for rocket interstage assemblies and components.
3. Show that the rocket interstage implementation of this framework decreases the design time without impeding innovation.

1.3 Delimitation of the problem

The applications developed to prove the methodology will only provide a representative number of possible operations in the design of rocket interstages. It is assumed that the joining method between parts will be bolted flanges. Other joining methods and are used in designing interstages, but it is not necessary to provide an exhaustive collection of operations to demonstrate the method. In addition, the implementation will be limited to a specific CAD application, NX 5, but the proposed methods will work with any application that has a sufficiently complete API library available to the developer. The application is not intended for public release. It is intended to prove the concept; however, if engineers and designers of rockets were to apply their rules, heuristics and knowledge to this rocket interstage framework, significant time-savings would result.

1.4 Document organization

Now that the problem has been described and the research objectives have been defined, chapter two will provide a review of relevant literature upon which this research has been founded. It will also describe similar research that has been conducted by others. Chapter three will outline and describe the methodologies/architecture used and created to reach the thesis objectives and chapter four will provide details on how these methods and the architected framework were implemented for the specific case study. Chapter five will then present the results of the case study which predict time savings of more than 90% . Finally, chapter six will conclude that high-level, product type-specific operations can, in fact, streamline the design process without impeding innovation.

2 Literature Review

A wide range of research has been performed in the field of computer aided design and especially in developing methods to streamline the product development process. This chapter describes those contributions and how they have impacted this research. The specific topics that will be discussed are:

1. Parametric modeling
2. High level features
3. Product-specific design applications
4. Associativity

First, however, it is necessary to provide definitions and descriptions of key terms and concepts which are crucial to understanding computer aided design and this thesis.

2.1 Definitions

These definitions and descriptions of terms are obtained from [Zeid, 2005]. If the reader seeks further background and clarification on these concepts he or she should refer to said reference.

Solid Model: A solid model is a "complete, valid, and unambiguous representation of an object." Complete means that any point in space can be classified as being inside, outside, or on the boundary of the object. Valid means there are no dangling edges or faces. Unambiguous means there is one and only one interpretation of the model.

Geometry and Topology: The geometry of a solid model is the metric information of the model's elements such as the lengths of lines, radii of arcs and depths of holes. Topology is the connectivity and associativity of the entities. For example, Line 1 shares a vertex with Line 2 and Arc 1. A solid model must store both the geometry and topology of its entities to satisfy the completeness and unambiguity requirements.

Set Theory: Set theory is the mathematical representation of solids. A solid is defined by "a point set S in 3D Euclidean space (E^3)." The set S is the union of its interior (iS) and its boundary (bS). The subset of all points on the exterior of S is called the complement of S (cS).

Parametric: Parametric modeling refers to the ability to change the values associated with geometry which results in a new definition of the solid model.

Variable, dimension, expression, equation: A variable is a name that can take on multiple values one at a time. A dimension is a variable tied to specific geometry usually in a sketch. An expression is a collection of variables and dimension names combined by mathematical operators. An equation is a statement containing a dimension name or variable followed by an equals sign followed by an expression.

Constraint: A topological condition between entities in a solid model. For example, a coincidence constraint ensures that two entities have the same coordinates in space. A perpendicular constraint makes the angle between two lines 90 degrees.

Parameter: Variables and dimensions are both referred to as parameters.

Feature: According to Zeid a feature "is defined as a shape and an operation to build parts. The shape is a two-dimensional sketch…the operation is an activity that converts the sketch into a three-dimensional shape. Sample operations include extrude, revolve, fillet, shell, chamfer, and sweep." This thesis uses a broader definition which includes the possibility of performing multiple operations on shapes as a single feature.

Associativity: Associativity means providing perpetual links between geometry, and expressions within multiple components of an assembly.

2.2 Parametric modeling

This thesis will provide superior methods of defining geometry and topology of product models. The fundamental CAD methods discussed in section 2.2 form the backbone of this research and the foundation for much of the research that will be discussed in subsequent sections.

Improvements to CAD have focused on reducing the number of user operations necessary to define the topology and geometry of products. Parametric modeling and relating parameters with equations are fundamental methods of reducing operations [Lendermann, 2005]. Once a parametric model is created, products with similar topology can be modeled simply by updating the key driving parameter values.

Feature based design is another "one of the fundamental design paradigms of CAD systems" [Hoffmann,1998]. It allows the user to define and modify the model at a higher level than the point and curve entities. Even though feature-based parametric modeling increases the efficiency of CAD design, complex parts may require hundreds of features and thousands of parameters and so more advanced methods have been developed that modify geometry at an even higher level [Elliott, 2004].

2.3 High level features

After the groundwork for feature based modeling was laid, it was determined that "to devise a universal set of features would lead to a potentially unmanageable number of features that a CAD system might be asked to provide" [Hoffmann, 1998]. Because of this, several research teams set out to "provide CAD systems with a basic mechanism to define features that fit the end user needs" [Hoffmann, 1998; Bidarra, 1998; Shaw, 1994; Tang, 2001]. These are referred to as User-Defined Features (UDFs). They work by allowing the user to define a set of standard, low level features that will be grouped together. For example, a sketch, an extrusion, and a corner blend feature can be grouped into a UDF called a boss. UDFs also allow the designer to limit which expressions will be available to the end user to minimize the number of inputs required and to prevent important parametric relations from being altered.

Many designers have achieved significant time savings by using UDFs, such as Bruno Lamarche and Louis Rivest [2007] who reported an 86% time improvement. The UDF they created added lightening pockets to an aircraft skin panel between any given set of stringers and frames.

Several CAD vendors also offer higher level design features as standard features. In the 2006 release of SolidWorks, Dassault Systemes "introduced a number of features designed to assist in building plastic part features" [Jankowski, 2005]. Among the new features were a “Mounting Boss”, “Snap Hook”, and a “Snap Hook Groove.” Most CAD vendors also provide optional features for sheet metal design and wiring design including Siemens’ NX, Dassault Systemes’ CATIA, and Parametric Technology Corporation’s Pro/Engineer.

The disadvantages of the current high-level features are that they cannot create the interpart associativities in an assembly, and can only combine a limited number of low-level features. This thesis builds upon the discussed methods of using high-level sets of features to minimize the effort required to define the model. However, the programmatic operations of this thesis are able to operate on a much higher level than inserting individual features into single components. Each operation can create large sets of features in multiple parts as well as the inter-part expressions and geometry links needed to maintain associativity. The proposed method establishes a strategic set of high-level feature operations that are tailored to meet the needs of a specific product type.

2.4 Product specific design applications

Other researchers have also found it useful to develop methods that are tailored to a certain product type.

Huh and Kim [Huh, 1991] developed the “RIBBER” application for adding supplementary features like ribs and bosses to plastic injection molded parts. They used Pro/Engineer as the geometric modeler and RIBBER synthesized the necessary parameters for the supplementary features based on manufacturing, molding, and strength information.

Ong and Lee [Ong, 1995] developed CADFEED, a CAD-based applications to automate the design of the feed system for plastic injection molds. Parts were designed using a features database of common plastic injection molded part bases and “add-on” features then the application found appropriate locations and sizes for various types of gates, sprues and runners.

Another product specific design application was developed by Delap, Hogge, and Jensen [Delap, 2006]. Using the NX application programming interface (API) they developed an application for preliminary design and optimization of jet engine flow paths. They created parametric models programmatically and interfaced the CAD geometry with simple analysis codes and optimization routines.

In a joint project between the Institute of Product Engineering at the University of Duisburg-Essen and Siemens, it was discovered that the design of shafts and impellers of compressors was effectively automated by integrating knowledge directly into the definition of UDFs using NX’s knowledge based engineering (KBE) software (Knowledge Fusion). By using these “modular” knowledgeable UDFs, “one can renounce an ‘omniscient’ KBE application … that would lead to a highly component specific application” [Danjou, 2008]. The definition of the components is stored in data files and the KBE application “imports the data file, analyzes the content and distributes the input parameters to all relevant UDFs” [Danjou, 2008]. This thesis also applies the idea of embedding intelligence into the feature operations to automate much of the design process while remaining somewhat flexible.

2.5 Associativity

Another area of research that has contributed to CAD's overall efficiency is the idea of associativity. When a design change is made in one part of the assembly, related components can also update automatically through the use of inter-part expressions or linked geometry. This capability within CAD has led to increased time savings due to the reduction of operations needed to update the design. It has also led to additional robustness in parametric modeling because once the associativity is established, mating errors are significantly reduced [Emch, 2002]. There is still ample room for improvement, however, because learning to create these "complex parametric models that are very robust…can take years of training and experience to acquire" [Delap, 2006]. Lendermann [2005] explains that "associativity requires a clearly defined data flow. The more associativities exist in a geometrical model, the more vulnerable it gets to circular references…" These difficulties associated with creating associative parametric models have been addressed in several ways including the method of top-down modeling, and the strategy of using custom defined assembly features to handle the associative aspects of modeling.

2.5.1 Top-Down modeling method

Lendermann [2005] explains that the problem of circular references mentioned previously "can be solved by hierarchical structures …a superior component could contain the information of the [sub-components]." This approach is known as the top-down modeling method. It is very useful for maintaining associativity in an assembly as has been noted in numerous studies.

Francesco Mosca [2001] used this approach to define "control structures" for gear box design and project management. "Such a way to manage a project (critical data organized in a control structure)," he said, "leads to a simple modifying of the product or to a redesign without remodeling the geometry but regenerating it."

Aircraft body design was also significantly improved using the top-down approach [Emch, 2002]. Emch explains that "another function of the Control Structure is the definition of interfaces between subsystems and between major elements within the subsystems themselves" and that "the advantage to the design process is the ability to greatly shorten development cycles while attaining aggressive performance goals."

The top-down modeling method and its use in maintaining correct interface definitions between parts is a fundamental strategy employed by this thesis. This thesis also presents methods for automating the generation of assembly control structures and defining the geometry links between the control structure and the component level parts.

2.5.2 Assembly features

Ma [2007] introduced associative assembly features (AAF) and a sub-category of AAFs called assembly design features as a higher level method of doing assembly level associative design. An example of such an assembly design feature is a guide pin pattern on a plastic injection mould base. The feature resides in the top level assembly file and includes a parametric expression-based representation of the pattern that "contains rules to determine the number of pins required for a specific mould size." The sub-components in the assembly inherit these parameters and "such links are retrieved, managed and saved via a set of feature object modification methods" which are part of the "feature manager." The user selects the desired type and size of mold assembly from a graphical user interface (GUI) and “an instance of the assembly is inserted into the design model.”

The author’s method will similarly create linked expressions and linked geometry between parts in the assembly. The superiority of the author’s method, however, is that there will not be pre-determined assembly configurations loaded from a library. Instead, the user defines custom assembly layouts through a GUI and the application creates the configurations automatically.

3 Methods

The methods discussed in this chapter represent a set of automated steps that can be used to design a specific type of product. The objective of this thesis is to prove that such a set of programmatic operations can reduce the time and expertise required for defining the geometry, topology, and associativity of assemblies. Although these methods are specifically chosen for their applicability to a specific product, they will be presented with as much generality as possible to enable their extension into other product types. These methods were developed using the NX Open API in C++ and this thesis may use language specific to that programming and modeling environment; however, the content of the methods should be general enough to apply to any similarly capable environment.

3.1 Overall method for design automation

The steps used in automating the design of assemblies and components employ the techniques discussed in Chapter 2, especially the control structure-based top down method. The steps can be grouped into three key operations.

Operation 1: Define the control structure and part ownership and create the geometric links between the parents and children for the primary design features.

Operation 2: Define the cross sections of children level parts, make them into solids, and add detail features such as chamfers, blend radii, and fastener holes.

Operation 3: Add secondary design features such as access doors.

For the remainder of this chapter and in subsequent chapters, A will represent an assembly-type part (a parent to at least one other part) and C will represent a component type part (one having no children). The superscript of A or C will represent the hierarchical level of the part and the subscript will represent its position relative to its sibling parts, e.g. ${A^2}_1$ is the first child of its parent and is a second level assembly, and ${C^3}_2$ is the second child of it's parent and is a third level component. No subscript will be used when referring to the collection of all parts on a certain hierarchical level. This is illustrated in Figure 3-1.

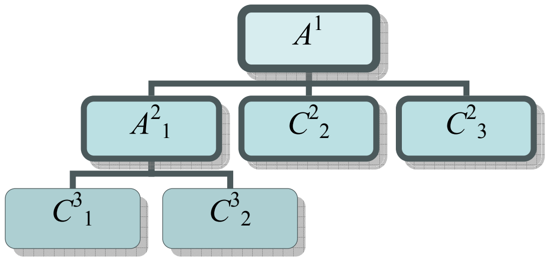


Figure 3-1: Assembly part notation

In addition, let $\underset{Sk}{O}(\ )$ represent an operation on sketch geometry. The notation from set theory for boundaries (bS) will be modified by a subscript 2 or 3 to distinguish between two-dimensional boundaries b_2 (sketch geometry) and three-dimensional boundaries b_3 (faces and edges of the solid). Figure 3-2 illustrates these terms as well as other key terms that will be introduced in this chapter. b_2C^2 represents the two dimensional boundary of a component. b_3C^2 represents the three dimensional boundary of a component. I_{ij} is a unit of the control structure which represents the interface between components i and j. The clearance between components is denoted by ε and the chain link symbol denotes the sketch constraints between I_{ij} and b_2C^2. A^1 is the top level assembly that contains the control structure.

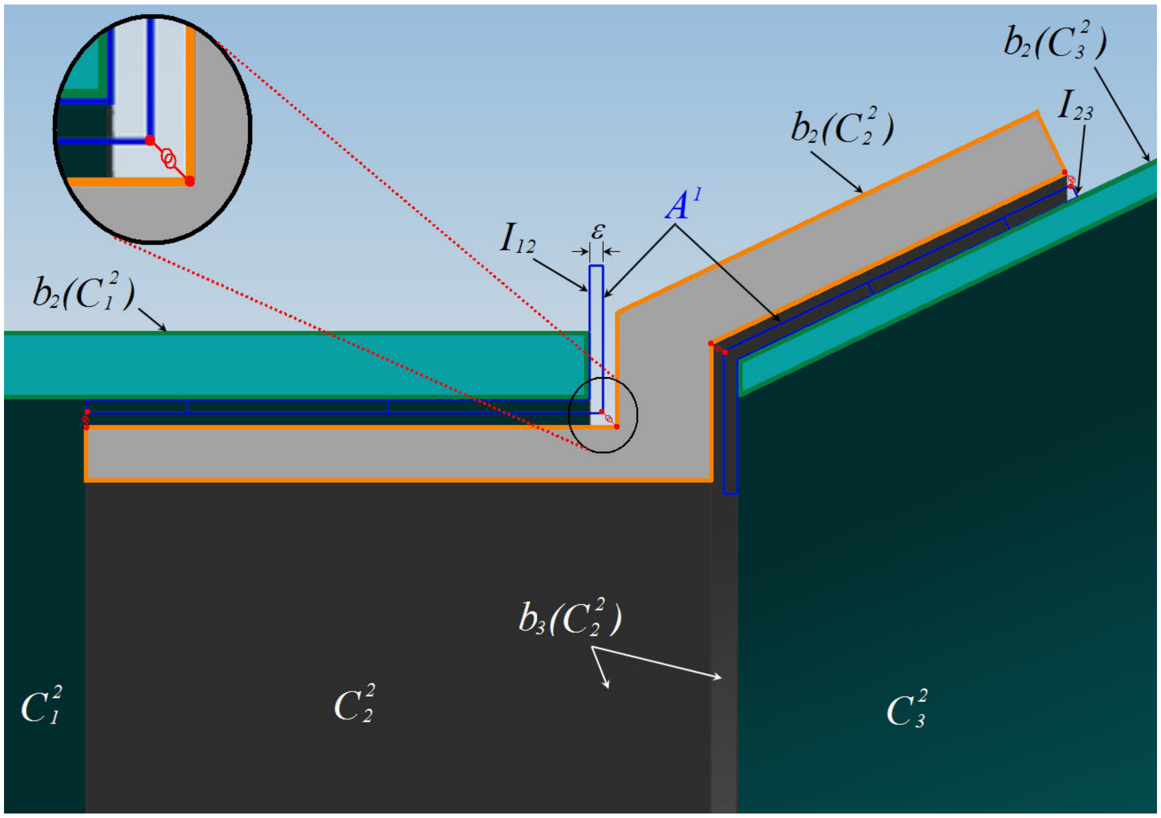


Figure 3-2: Key terms

3.2 Operation 1: control structure, part ownership, geometric links

The first operation helps the designer define the overall layout of the entire assembly and automatically creates the associative links from the top level control structure owned by A^1 to its children (A^2 and C^2). The control structure can be represented mathematically as the set of all points at which its children's boundaries (b_2C^2) are located within a given clearance of each other (ε). Let I_{ij} denote the intersection between components $C^2{}_i$ and $C^2{}_j$.

$$I_{ij} = b_2 C_i^2 \overset{\varepsilon}{\bigcap} b_2 C_j^2 \tag{3-1}$$

Then A^1 is the sum of all intersections between its n children.

$$A^1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} I_{ij} \tag{3-2}$$

See Figure 3-2 for illustrations of I_{ij}, b_2C^2, ε, and A^1.

When modeling assemblies, a designer will start with a hand drawing of the overall layout and will therefore know by inspection, what the control structure needs to look like. The automated method serves as a tool to quickly go from a hand drawn assembly configuration to fully defined models with associativity.

3.2.1 Application architecture

The control structure is defined through a framework of C++ class objects (See Figure 3-3 on page 19) which are instantiated based on user inputs from a graphical user interface (GUI). The Interface Object class contains all the data needed to define the geometry of one unit in the interface control structure (I_{ij}). It contains the member function $\underset{Sk}{\bigcirc}(I_{ij})$ to create the Sketch control geometry, and the member function $\overset{f}{\underset{Sk}{\bigcirc}}(I_{ij})$ to create the fastener control geometry based, on the member variables. It also contains the member function $I_{ij}(C_i^2) = \underset{Wl}{\bigcirc}(I_{ij}(A^1) \rightarrow C_i^2)$ to Wave link a unit (I_{ij}) of the control structure from A^1 into a child component C^2_i. The member function $\underset{Sk}{O}\left(b_2C_i^2 = I_{ij}(C_i^2)\right)$ constrains the sketch geometry of the child part ($b_2C^2_i$) to its wave linked geometry ($I_{ij}(C^2_i)$). The Interface Derived Part class contains a collection of interface objects as well as the positions on each interface. It also has a function that calls each of its interface objects' functions.

C++Architecture

Interface Object Class (I_{ij})

- Member variables
- Member functions
  - MakeSketchControlGeometry(Member variables) $\underset{Sk}{O}\left(I_{ij}\right)$
  - MakeFastenerControlGeometry(Member variables) $\overset{f}{\underset{Sk}{O}}\left(I_{ij}\right)$
  - LinkChildToParent(Child) $\underset{Wl}{O}(I_{ij} \rightarrow C_i^2)$
  - MakeChildGeometry(Child Position) $\underset{Sk}{O}\left(I_{ij}(A^1) \rightarrow I_{ij}(C_i^2)\right)$

Interface Derived Part Class

- Member variables
  - MyInterfaceObjects
  - MyInterfacePositions
- Member functions
  - SketchMyInterfaces(MyInterfaceObjects, MyInterfacePositions)

Figure 3-3: C++ class architecture

3.2.2 Application procedure

After the data is collected from the GUI, the application automatically performs the following steps associated with operation 1.

1. Create part files for A^1, A^2 , and C^2.
2. Create control sketch in A^1

a. Create and open sketch feature
b. For each interface object: call MakeControlGeometry(Data)
c. Close sketch feature
3. For each interface object: call MakeFastenerControlGeometry(Data)
4. For each Interface Derived Part Object
d. For each Interface Object in collection: call LinkChildToParent
e. Create and open sketch feature
f. Call SketchMyInterfaces(MyInterfaceObjects, MyInterfacePositions)
g. Close sketch feature

At this point each part will have the geometric links associated with the control structure and will have sketch geometry that is constrained to the control structure. A sample part sketch would look like that in Figure 3-4 where the blue lines represent the wave linked control structures $I_{12}(C_2^2)$ and $I_{23}(C_2^2)$, the orange lines represent the sketch geometry (a subset of $b_2C_2^2$), and the red points indicate constraints between the sketch geometry and the linked control structures (Red chain links in Figure 3-2). The component bodies are shown in gray for reference.

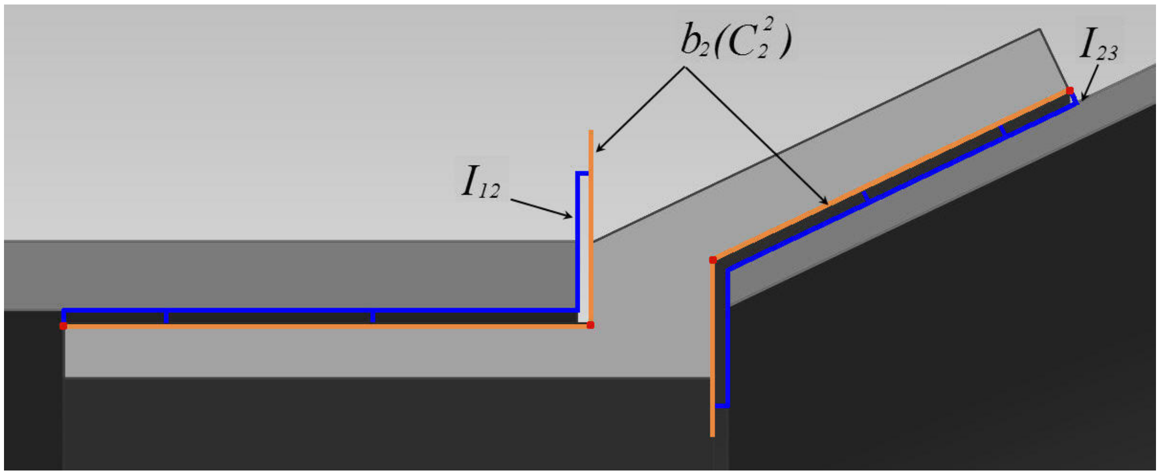


Figure 3-4: Completion of operation 1 showing the sketch of one part

3.2.3 Scope

This framework is general enough so that any interface type can be created by defining a class that inherits from the Interface Object Class. Possible interface types that may be created using this framework include a manacle joint, a weldment, or riveted joints. For this thesis, one interface type has been implemented that represents a bolted flange joint.

3.3 Operation 2: Cross sections, solids, detail features

The primary steps associated with Operation 2 are

1. Complete the part cross section sketches
2. Create solids from sketches
3. Apply blends and chamfers to edges
4. Create fastener hole features and patterns

3.3.1 Step 1: Part cross sections

The first step of Operation 2, completing the part cross section sketches, is performed interactively by the designer. This is where the most variability exists in the design and since interactive sketching is fairly easy and fast, it is more advantageous to not hard code the creation of individual part sketches. The use of macros, or a library of applications to generate common sketches would accelerate this step, but will be left for future work. It should also be noted that if this step were automated, Operation 2 could effectively be combined into Operation 1. This step completes the definition of b_2C^2 which is illustrated in Figure 3-5.

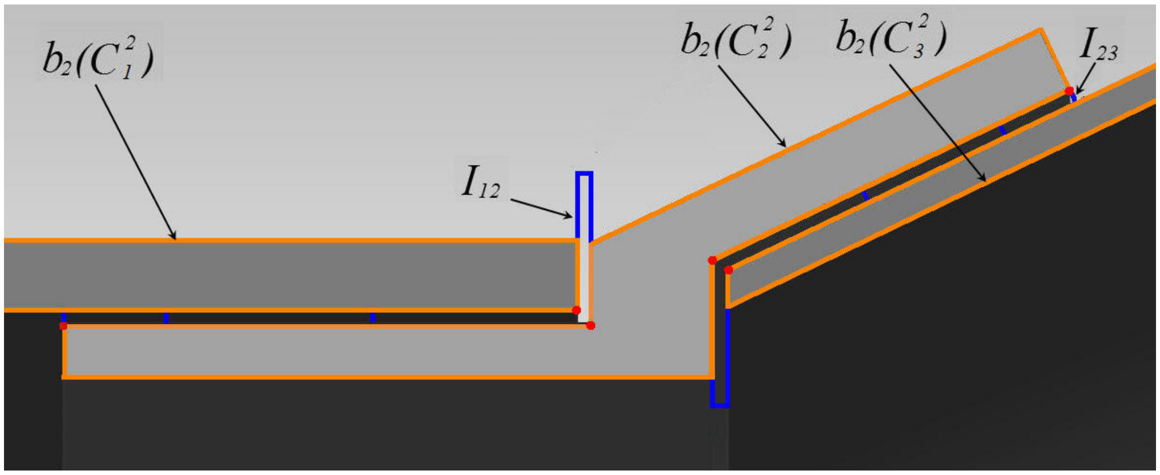


Figure 3-5: Completion of operation 2 step 1

3.3.2 Step 2: Solids

It is programmatically trivial to create the solidifying feature from the cross section sketches. As long as the sketch is named according to a pre-determined convention it can be retrieved and either extruded or revolved. Certain interface types may also contain parameters needed to perform this step such as an extrusion distance or revolve angle and would therefore create the necessary variables as part of Operation 1. In that case, the expressions would be named according to a convention and the application for Operation 2 would reference the established expression name while creating the feature.

3.3.3 Step 3: Blends and chamfers

The method for automating the creation of blends and chamfers is also based on the interface types associated with the part. Blends and chamfers are commonly used to facilitate the assembly of mating parts and are applied to easily predicted edges especially when the interface type is known. The method used in this step of Operation 2 queries the edges of each part and determines which edges intersect $I_{ij}(C_i^2)$. Then based on the rules for each interface type, blends and chamfers are applied. The rules for whether a chamfer or a blend would be inserted are based on the specific vertex of the interface and could reference either the expressions created by the interface object functions or values retrieved from a GUI. The specific rules for the interface type implemented by this thesis will be discussed in more detail in chapter 4, but here is an outline of the procedure. Figure 3-6 illustrates the models after Step 3 has been completed. The yellow points denote the key vertices to which chamfers or blends have been applied.

```
For all parts $C_i^2 \in A^1$  $(0 \le i \le n)$
    For all edges $e_b \in (C_i^2)$  $(0 \le b \le l)$
        For all key vertices $v_a \in I_{ij,k}(C_i^2)$  $(0 \le a \le m)$
            Calculate minimum distance $d_{min}$ between $v_a$ and $e_b$
                If $d_{min}$ < tolerance
                    Create Chamfer or Blend
                    Stop looping vertices
```

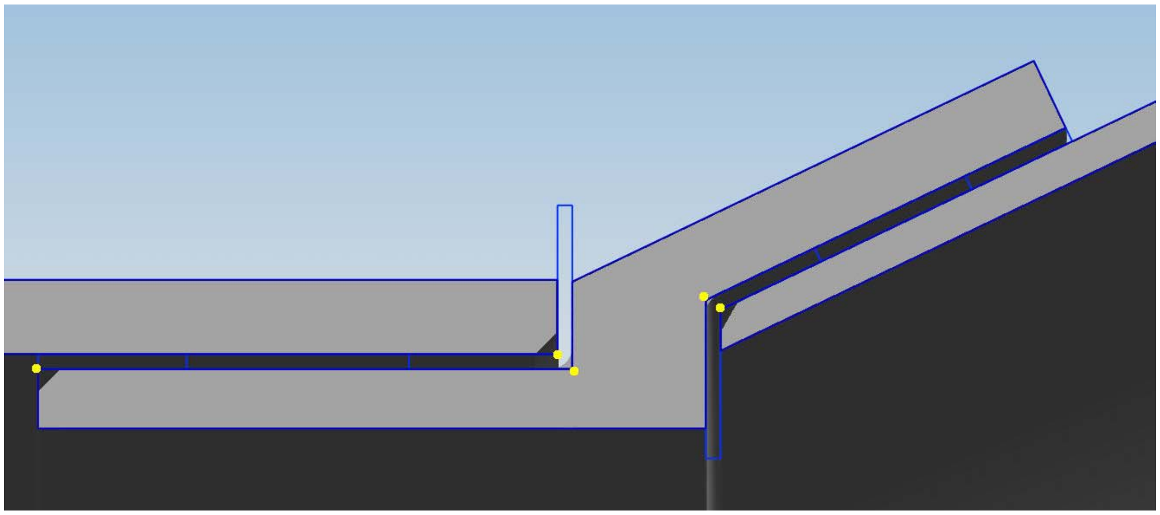


Figure 3-6: Operation 2 step 3 complete

3.3.4 Step 4: Fastener detail features

The method for adding fastener hole features is very similar to the blends and chamfers method, except instead of cycling through the vertices associated with the interface control geometry, it cycles through the fastener control geometry for each interface object. Again, the hole features and pattern features reference the expressions which were created by the Interface Object class member functions (Figure 3-3). The algorithm goes as follows (See Figure 3-7 for results):

```
For all parts $C_i^2 \in A^1$  $(0 \leq i \leq n)$
    For all center lines $cL_a \in I_{ij,k}(C_i^2)$  $(0 \leq a \leq m)$
        Insert Hole Feature
        Name Hole Feature according to convention
    For all features $f_b \in (C_i^2)$  $(0 \leq b \leq l)$
        If feature type and name match convention for an interface hole
            Create hole pattern
```

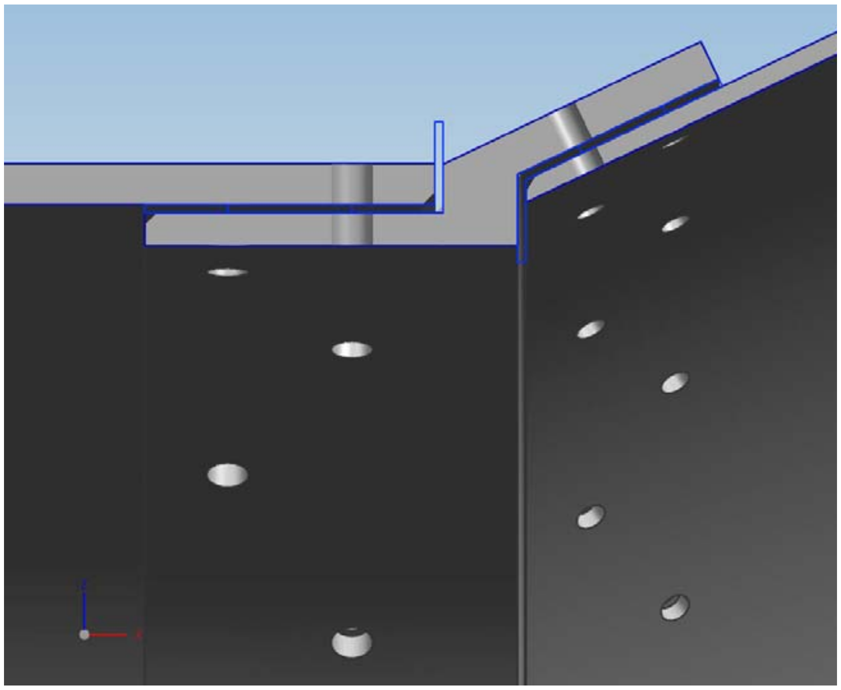


Figure 3-7: Operation 2 step 4 complete

3.4 Operation 3: Doors, door frames, and interstage alterations

Although adding doors and door frames has fewer applications in other product types, the methods presented are general enough to work on an arbitrary input surface, and an arbitrary door shape and therefore are not restricted to cylindrical and conical products or rectangular doors. This operation includes a method for creating and altering the component bodies, as well as a method for creating the fastener holes needed to assemble the components. If multiple doors are needed, this operation can be executed as many times as required.

3.4.1 Operation 3a: Component bodies

When inserting a door into a component, it is often required to stiffen the area surrounding the cutout. Step 1 of Operation 3a will outline the procedure for stiffening the interstage cutout. Step 2 will outline the procedure for modeling the door frame and Step 3 will present the method for modeling the door.

Step 1: Altering the interstage

There are two inputs required for this operation. The first is the inner most surface of the part body to which the door and frame will be added. We will call this S_{IML} for "inner mold line". The second input is a sketch of the cutout region. The sketch must be created on a plane which is tangent to S_{IML} at the center point of the cutout and must contain a single closed loop with no convex curvature. The input sketch will be called b_2S_{CUT}. The inputs are illustrated in Figure 3-8.

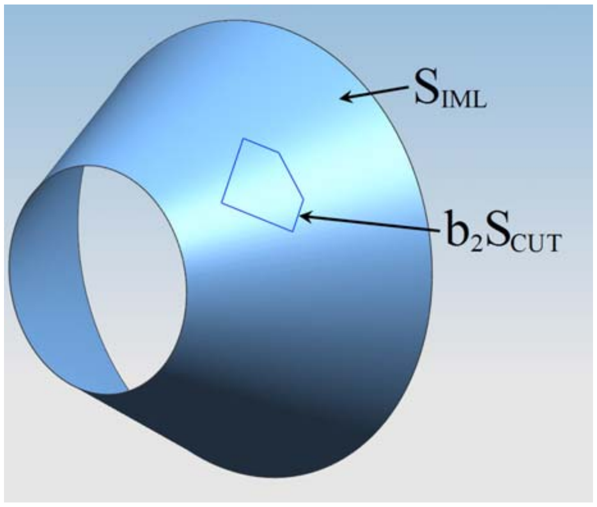


Figure 3-8: Inputs required for operation 3

First, let $\underset{Ext}{O}(b_2S_{CUT})$ be the symmetric Extrusion of b_2S_{CUT} normal to its sketch plane and let $S_{CUT} = \underset{Bl}{O}\left(\underset{Ext}{O}(b_2S_{CUT})\right)$ be the result of a Blending operation on $\underset{Ext}{O}(b_2S_{CUT})$ which ensures a set of tangent continuous surfaces normal to S_{IML} as illustrated in Figure 3-9.

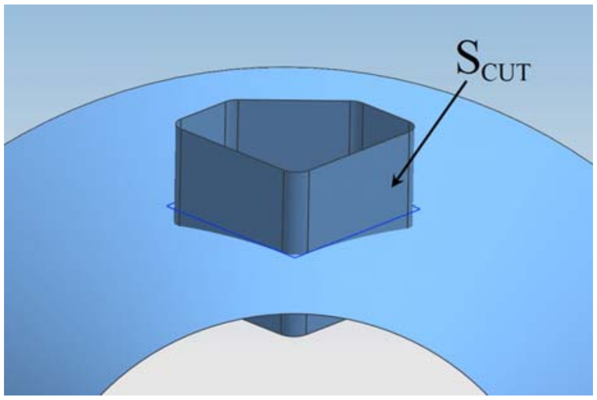


Figure 3-9: Blended extrusion of b_2S_{CUT}

Let O_S be a set of surfaces offset from a set of surfaces S. The next step in the procedure is to generate offsets from S_{IML} and S_{CUT} for each position on the cross section of the thickening body (B_{TH}). Figure 3-10 is an example of what a thickening cross section might look like and where offset surfaces would be needed. The grey rectangle represents the interstage body's cross section before the cutout and thickening. The dotted lines are the input surfaces and the solid black lines represent the new cross section around the cutout region. The hatched grey area would be trimmed from the current interstage body as will be discussed later. Figure 3-11 shows what the CAD model would look like after applying the offsets.

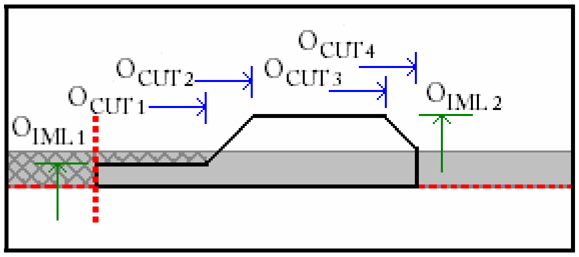


Figure 3-10: Stiffening cross section with offset surface markers

The segments of the thickening cross section which are either parallel or perpendicular to S_{IML} at the cutout's midplane can be produced by the operation $\underset{Tr}{\bigcirc}(S \rightarrow S_{1...n})$ which denotes a surface *S* Trimmed by *n* surfaces S_1 through S_n. For example, the outer most surface represented by the top horizontal line in Figure 3-10 would be produced by the operation $\underset{Tr}{\bigcirc}(O_{IML2} \rightarrow O_{CUT2}, O_{CUT3})$. The result of each trimming operation is a subset of b_3B_{TH}, the 3-dimensional boundary of the thickening body (see Equation 3-3).

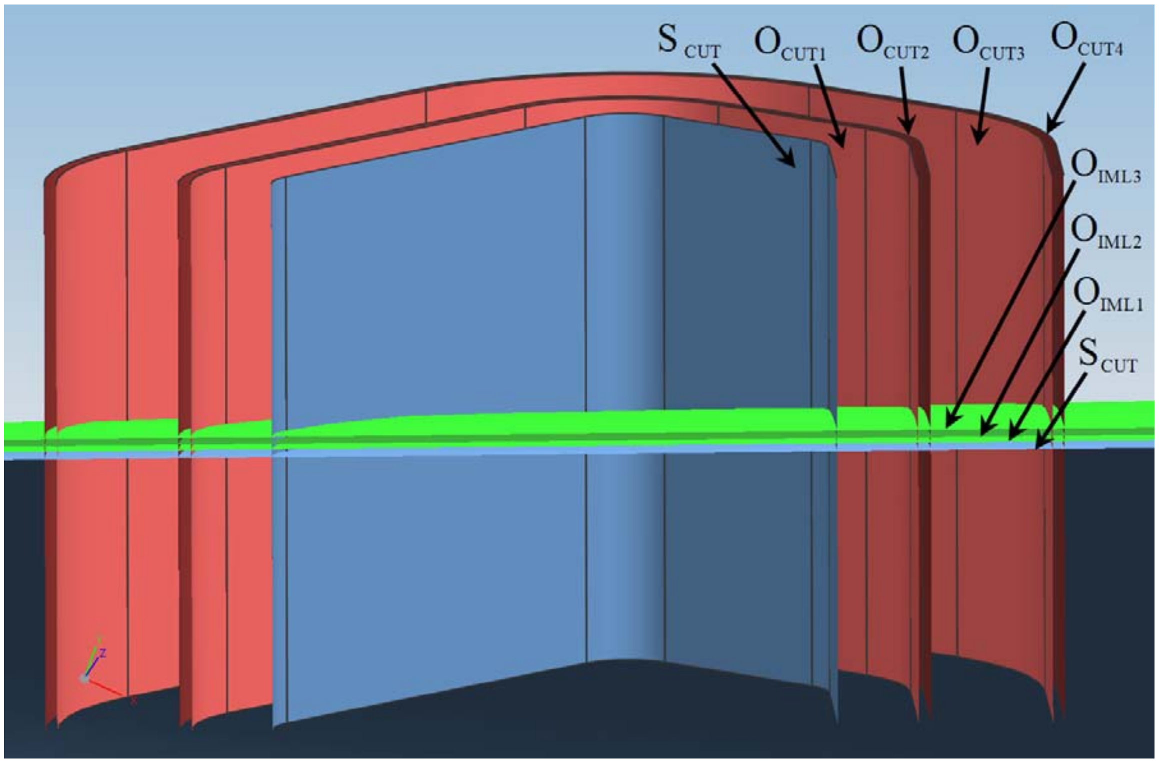
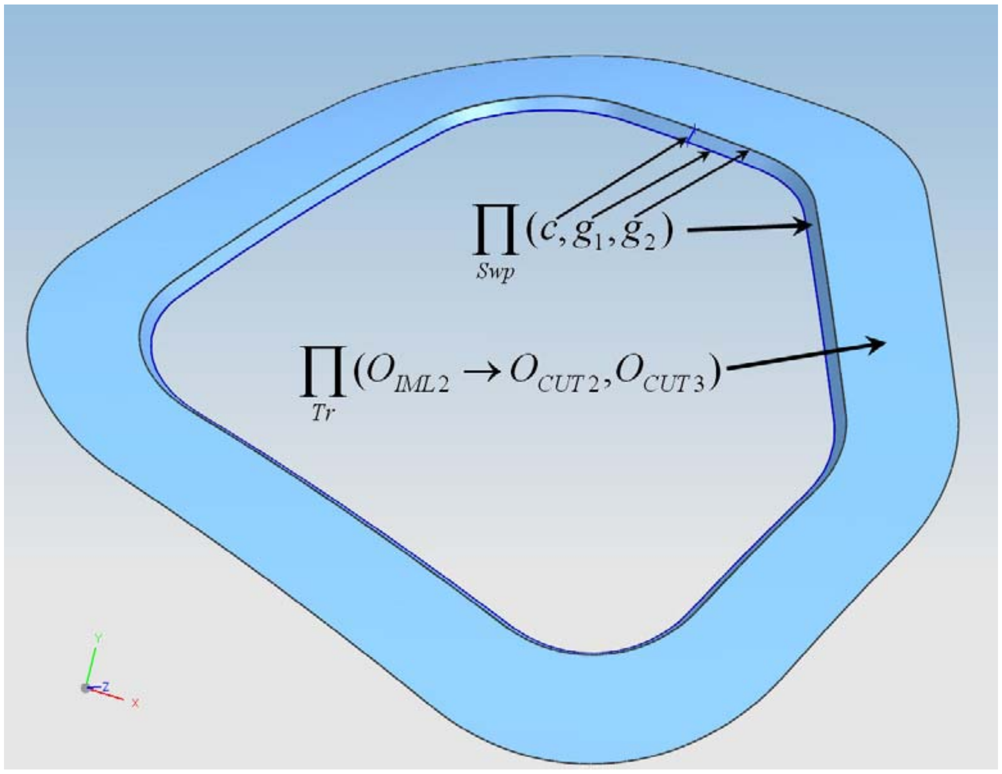


Figure 3-11: Input surfaces and offset surfaces

$$\underset{Tr}{\mathrm{O}}(O_{IML_i} \rightarrow O_{CUT_j}, O_{CUT_k}) \subseteq b_3 B_{TH} \qquad \textbf{(3-3)}$$

The remaining surfaces of b_3B_{TH} are produced by sweeping operations $\underset{Swp}{\mathrm{O}}(c, g_1, g_2)$ where a curve $c \in b_2 B_{TH}$ is swept along closed guide curves represented by $g_i = O_{IML} \cap O_{CUT}$. Figure 3-12 shows the result of this operation as well as the curve c and guide curves g_1 and g_2. It also shows the outermost trimmed surface mentioned as an example above.



$\prod_{Swp}(c, g_1, g_2)$
$\prod_{Tr}(O_{IML2} \rightarrow O_{CUT2}, O_{CUT3})$
Y
Z
X

Figure 3-12: A swept surface and a trimmed surface on b_3B_{TH}

Thus

$$b_3B_{TH} = \sum\left(\bigcirc_{Tr}(S_i \rightarrow S_j, S_k)\right) \cup \sum\left(\bigcirc_{Swp}(c, g_1, g_2)\right) \qquad \textbf{(3-4)}$$

and B_{th} can be created by sewing together its boundary surfaces b_3B_{TH}.

The final steps in altering the interstage part (equations 3-5 and 3-6) are trimming the main interstage body (B_{INT}) to remove the portions represented by grey hatching in Figure 3-10 and then uniting B_{INT} and B_{th}. The final cross section can be seen in Figure 3-13.

$$B_{INT} = \prod_{Tr}(B_{INT} \rightarrow b_3B_{TH}) \qquad \textbf{(3-5)}$$

$$B_{INT} = B_{INT} \cup B_{TH} \qquad \textbf{(3-6)}$$

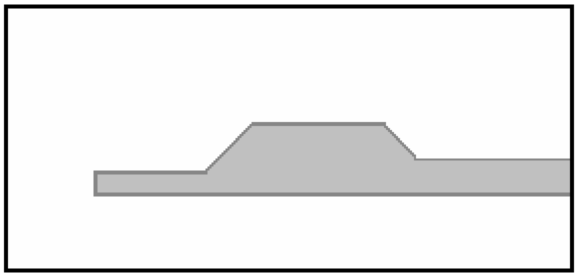


Figure 3-13: Final interstage cross section around door cutout

Step 2: Create door frame

First, add a blank part file $C^3{}_2$ (the frame) to A^2 (the interstage sub-assembly) and wave link S_{IML}, S_{CUT}, and O_{CUT1} from $C^3{}_1$ (the interstage) to $C^3{}_2$. We will refer to these local copies as

$$LS_{IML}, LS_{CUT}, LO_{CUT1} = \bigcirc_{WI}(S_{IML}, S_{CUT}, O_{CUT1} \rightarrow C_2^3) \tag{3-7}$$

Next, create offsets from LS_{IML} and LS_{cut} to maintain clearances.

$$CL_{IML}, CL_{CUT} = O_{LS_{IML}}, O_{LS_{CUT}} \tag{3-8}$$

Now create additional offset surfaces ($O_{1..n}$) from CL_{IML} and CL_{CUT} as needed to bound the door frame's body (B_{Fr}) and trim CL_{IML} , CL_{CUT} and $O_{1..n}$ to produce b_3B_{Fr}.

$$O_{1...n} = O_{(CL_{IML}, CL_{CUT})} \tag{3-9}$$

$$b_3B_{Fr} = \sum \bigcirc_{Tr}(CL_{IML}, CL_{CUT}, O_{1...n}) \tag{3-10}$$

B_{Fr} can now be created by sewing its bounding surfaces b_3B_{Fr}.

Step 3: Create door

The door is made by following the same process for creating the door frame except wave linked surfaces will be extracted from both the interstage part and/or the door frame part depending on the frame's cross section.

Once the bodies have been modeled, the expressions should be created to dictate the flange lengths and cutout sizes which will all be controlled by the offset distances. The wave linked geometry will ensure that the interfaces mate correctly.

3.4.2 Operation 3b: Door fastener holes

There are two sets of fastener holes for each door on the interstage. One set to fasten the frame to the interstage and one set to fasten the door to the frame. For each of these sets, the following procedure will be followed. Inputs such as hole sizes and distances from edges will be retrieved using either a GUI or existing expressions created by Operation 3a.

First, create a set of offset curves (c_o) from the edge (e) of the mating flange on the outer flange face (S_{fl}).

$$c_o = \underset{o}{\bigcirc}(e, S_{fl}) \tag{3-11}$$

Next, create evenly spaced point features ($\mathrm{f}_{\mathrm{pnt}_1} ... \mathrm{f}_{\mathrm{pnt}_n}$) on each segment of c_o.

Equations 3-12 and 3-13 calculate the parameter value (u) of the curve at which to place point i given n points on the curve. Equation 3-12 places points at the curve endpoints, and Equation 3-13 excludes the endpoints.

$$u_i = \frac{i}{n-1}, (0 \le i < n) \tag{3-12}$$

$$u_i = \frac{i+1}{n+1}, (0 \le i < n) \tag{3-13}$$

Now, create a hole feature that references each of the n points ($0 \ldots n$) on each of the m curves ($0 \ldots m$).

$$f_{Holes} = \underset{Holes}{O}\left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(c_i, f_{pnt_j})\right) \qquad \textbf{(3-14)}$$

To create mating hole features on the second part, first wave link each point to the second part. The local copy of the jth point on the ith curve is represented by

$$localf_{pnt_{ij}} = \underset{Wl}{O}(c_i, f_{pnt_j}). \qquad \textbf{(3-15)}$$

Next, insert a projection feature (f_{proj}) of all of the local points onto the outer flange surface (S_{fl}) with the operation

$$f_{proj} = \underset{P}{O}\left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(localf_{pnt_{ij}}, S_{fl})\right). \qquad \textbf{(3-16)}$$

Finally, create a hole feature of the projection feature points which uses inter-part expressions to match the hole parameters.

$$localf_{Holes} = \underset{Holes}{O}(f_{proj}) \qquad \textbf{(3-17)}$$

4 Implementation

This chapter gives the details involved in implementing the methods described in Chapter 3. The user can execute each of the operations from Chapter 3 by clicking on custom buttons in the NX environment as illustrated in Figure 4-1. Operation 1 is named Interface Manager. Operation 2 is called Detail Features. Operation 3a is called Insert Door and Operation 3b is called Door Fasteners.

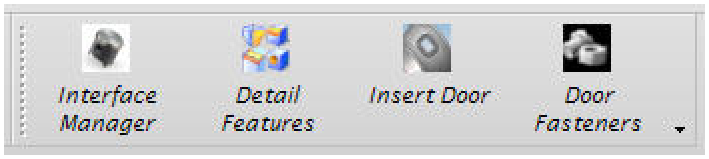


Figure 4-1: Custom buttons for executing automated operations

4.1 Operation 1: Interface manager

This thesis demonstrates the implementation of one type of interface, which represents a cylindrical or conical bolted flange joint. There are three key class objects which store the necessary data and perform the required CAD operations: the `JointObject` Class, the `ConicalJointObject` Class, and the `JointDerivedPart` Class. There is also a set of classes for the GUI through which the user instantiates and edits the three key class objects.

4.1.1 Joint object class

Below is a condensed version of the class definition file. The important member functions member variables will be described following the example code.

```
class JointObject{

public:
      //constructors
      JointObject();
      JointObject(std::string Name,        //Joint Name
                  int nRows,               //Number of Fastener Rows
                  double D,                //Fastener Diameter
                  int N,                   //Number of Fasteners
                  double xOut,             //Axial Position
                  double rOut,             //Radial Position
                  std::string matIn,       //Inner part Material
                  std::string matOut,      //Outer part Material
                  std::string dir,         //Flange is Fwd or Aft
                  std::string orient);     //Boss is Inner or Outer

      //destructor
      ~JointObject();
```

The constructor of the class takes in the necessary inputs to define the geometry of the interface. The comments beside each input describe their meaning. `Name` is used to distinguish between other interfaces in the assembly. The length of the flange is a function of `nRows`, `D`, `N`, `matIn`, and `matOut`. The position of the interface is determined by `xOut`, and `rOut`. The orientation is determined by `dir`, and `orient`.

```
      //accessor functions (OMITTED)
      //setter functions (OMITTED)

      //member functions
      virtual void CalculateGeometry();
      virtual void MakeExpressions();
      virtual void MakeGeometry();
      void MakeHoleControls();
            void RevolveSheet();
            void MakeDatumPlane(int Row);
            void HoleSketch(int Row);

      virtual void ReadMe(std::vector<std::string> &tokens, int &i);
      virtual void WriteMe(ofstream &fout);
      std::string GetBitmapPath();

      enum PartPosition{INNER,OUTER};//names of possible part positions
      void SketchPart(PartPosition pos);

protected:
      //member variables (OMITTED)
      //calculated values (OMITTED)
      //geometric objects (OMITTED)
};
```

The first group of member functions are responsible for creating the joint geometry in the top level assembly sketch. `CalculateGeometry` uses the member variables to determine the positions of the points and lines in the sketch including the location of the fastener holes. `MakeExpressions` creates the expressions in the part that the user can use to modify the joint parameters. The expressions are also dependent on the member variables and include conditional statements so that if for example, the part material changes, the flange length will be recalculated appropriately. `MakeGeometry` creates the actual lines, dimensions and constraints in the sketch as seen in Figure 4-2.

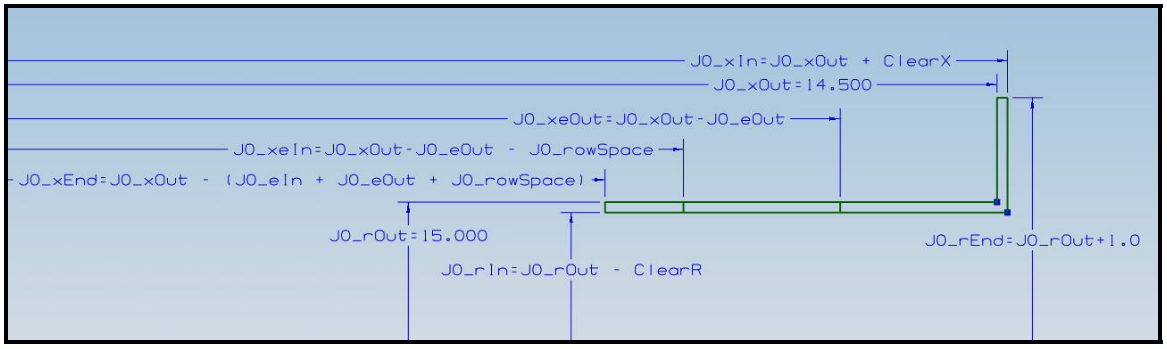


Figure 4-2: Control sketch displaying dimensions and expressions for a `JointObject` J0.

```
      void MakeHoleControls();
            void RevolveSheet();
            void MakeDatumPlane(int Row);
            void HoleSketch(int Row);
```

The second group of member functions is responsible for creating the additional geometry used to control the location and size of the fastener holes. MakeHoleControls works as follows.

```
void JointObject::MakeHoleControls(){
      MakeDatumPlane(1);
      HoleSketch(1);
      if(this->m_nRows ==2)
      {
            RevolveSheet();
            MakeDatumPlane(2);
            HoleSketch(2);
      }
}
```

`MakeDatumPlane` creates the plane for the hole sketch. If the input is 1, it creates the plane at the midpoint of the row 1 fastener location line (dimensioned as J0_xeOut in Figure 4-2). If the input is 2, it creates the plane at the midpoint of the edge of the sheet which is created by the `RevolveSheet` function. This function revolves the row 2 fastener location line (dimensioned as J0_xeIn in Figure 4-2) by half of the angle between holes in the first row $(\theta = 180 / N)$. Finally, HoleSketch creates a circle of size D and constrains it to be coincident with the same edge used to create the datum plane. Once completed, the control geometry will look like Figure 4-3.

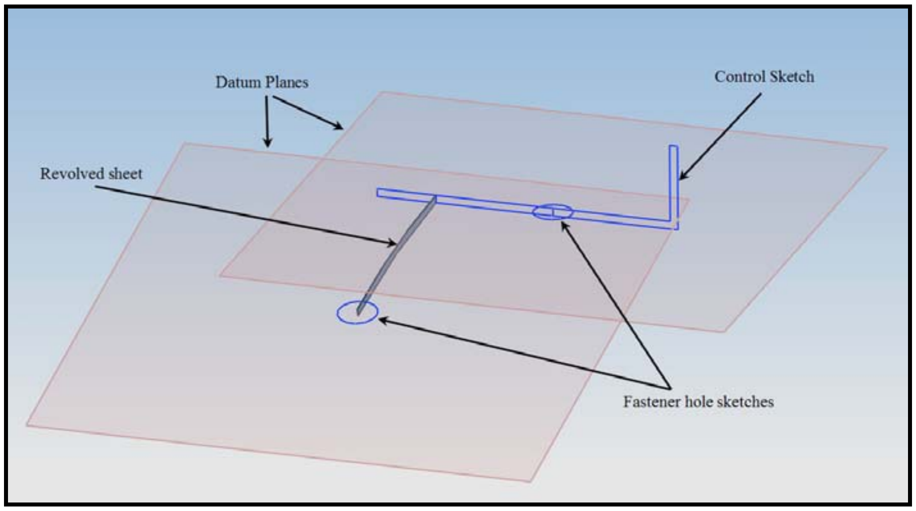


Figure 4-3: `JointObject` control sketch and hole control geometry

```
virtual void ReadMe(std::vector<std::string> &tokens, int &i);
virtual void WriteMe(ofstream &fout);
std::string GetBitmapPath();
```

The third group of member functions are used in the GUI. `ReadMe` parses an input file and sets the member variables associated with the `JointObject`. `WriteMe` writes the member variables to an output file. `GetBitmapPath` returns the path of an image which illustrates the direction and orientation of the `JointObject` for reference in the GUI.

```
enum PartPosition{INNER,OUTER};//names of possible part positions
void SketchPart(PartPosition pos);
```

The final member function, `SketchPart`, is used to create geometry in the cross section sketch of the child part and constrain it to the control structure. The `PartPosition` enumerator is used to distinguish the placement of the part geometry.

For parts in the INNER position, one line is created coincident with the both the start and end points of the flange line (dimensioned as J0_rIn in Figure 4-2) and another line is created coincident with the vertex between the boss and the flange (J0_xIn,J0_rIn in Figure 4-2) and parallel to the vertical boss line. For parts in the OUTER position, two lines are created coincident with the boss/flange vertex (J0_xOut,J0_rOut in Figure 4-2); one is parallel to the boss line (J0_xOut) and the other is parallel to the flange line (J0_rOut).

4.1.2 Conical joint object class

The ConicalJointObject class inherits from the JointObject class and therefore makes use of many of the JointObject member functions. The member functions declared as virtual in the JointObject class are overloaded in the ConicalJointObject class, and have the same functionality. The other key difference in the ConicalJointObject class is that it contains an additional input called Ang which designates the angle between the flange and the horizontal axis. The class definition is below and Figure 4-4 illustrates the control structure it creates.

```
class ConicalJointObject : public JointObject{

public:
      //constructor
      ConicalJointObject();
      ConicalJointObject(std::string Name,
                         int nRows,
                         double D,
                         int N,
                         double xOut,
                         double rOut,
                         double Ang,     //Angle between flange and axis
                         std::string matIn,
                         std::string matOut,
                         std::string dir,
                              std::string orient);

      //destructor
      ~ConicalJointObject();

      //overloaded member functions
      void CalculateGeometry();
      void MakeExpressions();
      void MakeGeometry();
      void WriteMe(ofstream &fout);
      void ReadMe(std::vector<std::string> &tokens,int &i);


private:
      //unique input value
      double m_Ang;
};
```

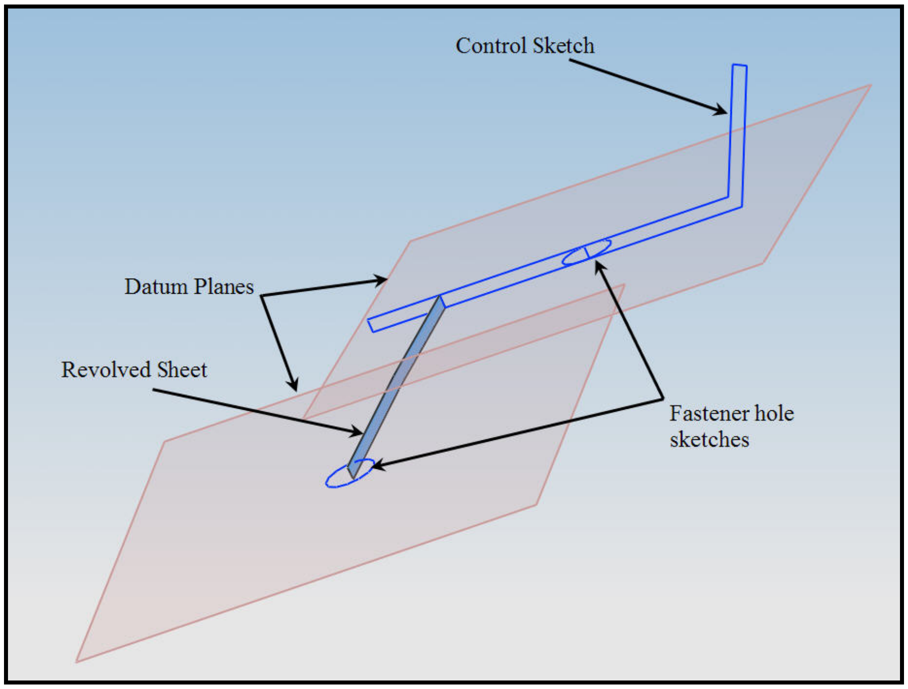
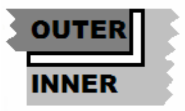
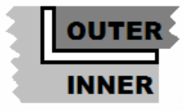
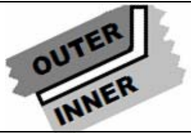
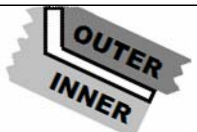
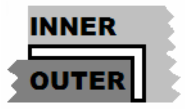
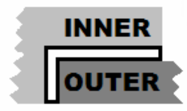
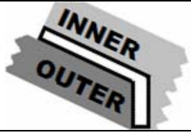
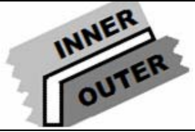


Figure 4-4: `ConicalJointObject` control sketch and hole control geometry

Table 4-1 illustrates the part positions and orientations of both the `JointObject` and `ConicalJointObject` control structures.

Table 4-1: Joint orientations and part positions

	`JointObject`		`ConicalJointObject`	
	Flange Forward	Flange Aft	Flange Forward	Flange Aft
Boss Outer				
Boss Inner				

4.1.3 Joint derived part class

The `JointDerivedPart` class contains the information for each component in the assembly and the member functions to begin defining the part geometry and topology and to create the associative links to the top level assembly part. The class definition file is shown below with descriptions.

```
class JointDerivedPart{
      friend class InterfaceManagerUI;
public:
      //constructors
      JointDerivedPart(void);
      JointDerivedPart(std::string Name,
            std::vector<JointObject*> Joints,
            std::vector<JointObject::PartPosition> Positions,
            std::string Mat,
            std::string subAssmName = "");
      //destructor
      ~JointDerivedPart(void);
```

The input variable, `Name`, is the actual component name in the assembly. `Joints` is the collection of `JointObjects` associate with the part. `Positions` is the collection of enumerators to define the location of the part on the interface. `Mat` is the material of the part, and `subAssmName` is the name of the sub-assembly which contains the part if it is not a direct child of the top level assembly. The default value of `subAssmName` is set to the null string to designate that the part does not belong to a sub-assembly.

```
	//accessor functions (OMITTED)
	//member functions
	void CreatePart();
	void MakeWaveLinks();
		void WaveLink(JointObject* Joint);
	void SketchCrossSection();
	void PassMaterialToJoints();
	void WriteMe(ofstream &fout);

private:
	//member variables (OMITTED)
};
```

The member function `CreatePart` creates the actual part file (and sub-assembly part file if applicable) and adds it to either the sub-assembly or the top level assembly. `MakeWaveLinks` loops through the `Joints` and calls `WaveLink` for each one to create the associative link features. Since each line in the control structure is named using the Joint name as a prefix, the `WaveLink` function can retrieve them by name. Note however, that NX stores the names of objects in upper case, so the joint name must be converted to upper case whenever the object is instantiated. `SketchCrossSection` loops through the joints and calls the `SketchPart` function for each `Joint` with the appropriate `PartPosition` as shown below.

```
void JointDerivedPart::SketchCrossSection(){
	. . . create the sketch, name it “CS” . . .
	for(int i=0;i<m_Joints.size();i++)
	{
		m_Joints[i]->SketchPart(m_Positions[i]);
	}
	//update and close control sketch
}
```

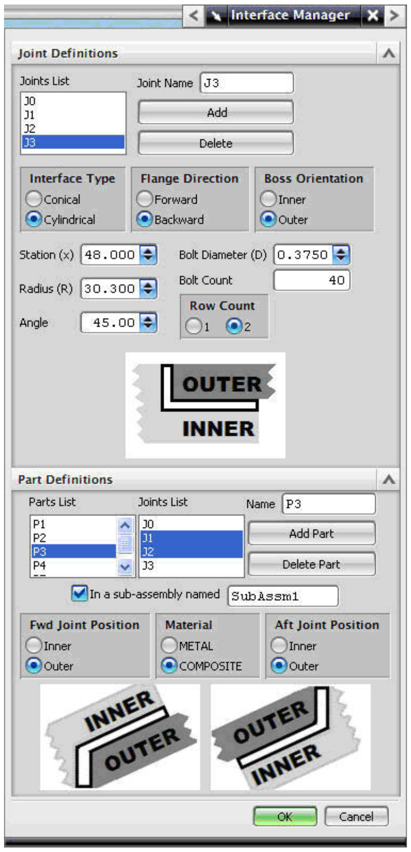
In order for subsequent operations to work, they need to know which JointObjects are associated with each part and what type of interfaces they are. This has been accomplished by creating attributes in the part file. When CreatePart is called, it adds an attribute to the part file named “Joints” with the value “0”. The WaveLink function creates the attributes “JointXName” and “JointXType” where X is incremented each time the function is called. This portion of the WaveLink code is shown below.

```
void JointDerivedPart::WaveLink(JointObject * Joint){
      ...
      //get the “JOINTS” string attribute and convert it to an integer
      NXString numJoints = workPart->GetStringAttribute("JOINTS");
      int n = atoi(numJoints.GetText());
      //increment it
      n++;
      //create the attribute “JOINTX = name of joint”
      workPart->SetAttribute("JOINT"+stringify(n), Joint->getName());
      //set the “JOINTS” attribute to the incremented value
      workPart->SetAttribute("JOINTS",stringify(n));

      //create the string variable for the joint type by concatenating
      //abbreviations for the type,direction, and orientation
      std::string JointType = Joint->IsConical ? "CON" : "CYL";
      JointType += Joint->getDir() == "FLANGE_FORWARD" ? "FWD" : "AFT";
      JointType += Joint->getOrient() == "BOSS_OUTER" ? "OUT" : "INN";
      //create the “JOINTX_TYPE = type” attribute
      workPart->SetAttribute("JOINT"+stringify(n)+"_TYPE",JointType);
      ...
}
```

PassMaterialToJoints sets the material variable for each Joint and calls its CalculateGeometry function. WriteMe, as in the JointObjectClass, writes the output file for the object.

4.1.4 GUI

The Interface Manager is the GUI that helps the user quickly create instances of the classes just described and then also uses their member functions to execute the steps of Operation 1 discussed in Chapter 3. Figure 4-5 illustrates the GUI which was designed using the NX UIStyler tool.


Interface Manager

Joint Definitions

Joints List: J0, J1, J2, [✓ J3]

Joint Name: J3

Interface Type: Conical [✓ Cylindrical]

Flange Direction: Forward [✓ Backward]

Boss Orientation: Inner [✓ Outer]

Station (x): 48.000

Radius (R): 30.300

Angle: 45.00

Bolt Diameter (D): 0.3750

Bolt Count: 40

Row Count: 1 [✓ 2]

Part Definitions

Parts List: P1, P2, [✓ P3], P4

Joints List: J0, [✓ J1], [✓ J2], J3

Name: P3

[✓] In a sub-assembly named: SubAssm1

Fwd Joint Position: Inner [✓ Outer]

Material: METAL [✓ COMPOSITE]

Aft Joint Position: Inner [✓ Outer]

Figure 4-5: Interface manager GUI

There are two collapsible sections in the Interface Manager. The first section is called Joint Definitions. The Joints List contains the name of each of the `JointObjects`. Next to the Joints List is the Joint Name box and the Add and Delete

Buttons. Below them, are the radio boxes and input boxes to define all of the necessary inputs for the `JointObjects`. At the bottom of the first section is the illustration of the current `JointObject` orientation.

The second section of the Interface Manager is the Part Definition section. It contains a similar list box for the Parts and also a multi-selection box where the user can specify which joints are associated with each part. If the part needs to be in a sub-assembly, the user can check the box below the lists and give the sub-assembly a name. The radio boxes below the check box are where the user designates the part position on the interface and the part material. Again, illustrations of the selected joints are displayed at the bottom.

The class definition of the GUI is shown below

```
class InterfaceManagerUI{
public:
      //NX class methods (OMITTED)
      //my class methods
      void CreateAssemblyAndControlStructure();
      void ResetJointImage();
      void WriteOutputFile();
    //------------------ UIStyler Callback Prototypes ---------------//
      (PROTOTYPE VARIABLE LISTS HAVE BEEN OMMITED FOR BEVITY)
    NXOpen::UIStyler::DialogState MakeMe_cb(…);
    NXOpen::UIStyler::DialogState action_AddJoint_act_cb(…);
    NXOpen::UIStyler::DialogState action_Delete_act_cb(…);
    NXOpen::UIStyler::DialogState ListActivated(…);
    NXOpen::UIStyler::DialogState RadioChangeDir(…);
    NXOpen::UIStyler::DialogState RadioChangeOr(…);
    NXOpen::UIStyler::DialogState RadioChangeType(…);
    NXOpen::UIStyler::DialogState MultiListJointsAcivated(…);
    NXOpen::UIStyler::DialogState PartsListActivated(…);
    NXOpen::UIStyler::DialogState SubAssmChanged(…);
    NXOpen::UIStyler::DialogState action_AddPart_act_cb(…);
    NXOpen::UIStyler::DialogState action_DeletePart_act_cb(…);
    NXOpen::UIStyler::DialogState OK_cb(…);
    NXOpen::UIStyler::DialogState Cancel_cb(…);
private:
    //Dialog Objects (OMITTED)
    //My variables
    std::vector<JointObject *> JointObjects;
    std::vector<JointDerivedPart *> Parts;
};
```

The `InterfaceManagerUI` class has two member variables in addition to the dialog objects: the `JointObjects` vector, and the `Parts` vector. These are collections of the class objects described in sections 4.1.1 to 4.1.3.

A callback is a class function that is executed following a specific action from the user. The associations between the callbacks and the GUI objects are illustrated in Figure 4-6.

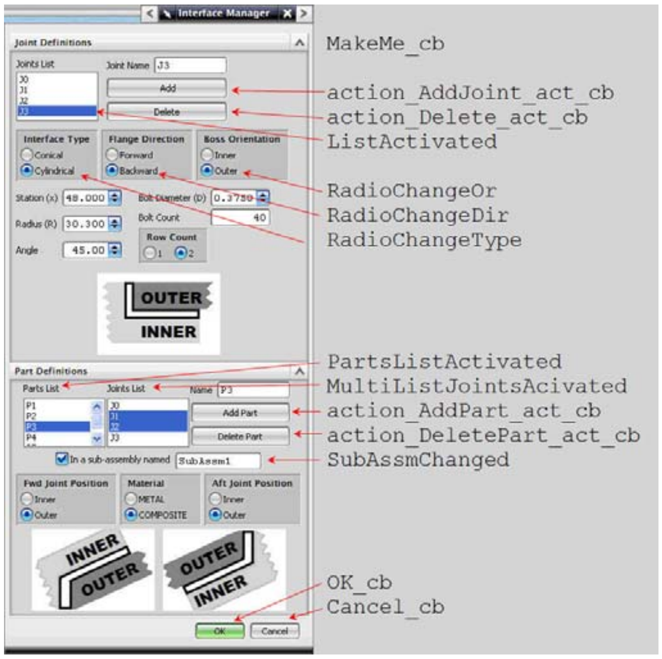


Figure 4-6: GUI callbacks

`MakeMe_cb` is executed when the GUI is first opened. It parses a file of defaults looking for the keywords “Joint”, “ConicalJoint”, and “Part”. When it finds a keyword, it creates the appropriate object using the default constructor, calls its `ReadMe` function which initializes the member variables, then adds the object to the collection and to the lists.

The `action_AddJoint_act_cb` callback instantiates a new `JointObject` using the current values of the input objects and adds it to `JointObjects` and the Joints list. It also verifies that the name of the object being added is not already in the list.

The `action_Delete_act_cb` callback finds the name in the Joints List that matches the current value in the Joint Name box and removes it from the lists and removes the associated object from the collection.

The `ListActivated` callback finds the JointObject in the collection whose name matches the selected name in the list and populates all of the dialog objects with values retrieved from the object, including the illustration path.

`RadioChangeDir`, `RadioChangeOr`, and `RadioChangeType` each call the `ResetJointImage` function, which resets the path of the illustration to match the current inputs.

The `PartsListActivated` callback finds the JointDerivedPart object in the collection that matches the selected name and highlights the items in the Joints List that match one of the JointObjects in the JointDerivedPart. It also populates the values of the other dialog objects with values retrieved from the JointDerivedPart object. If only one joint is associated with the JointDerivedPart object, then the second image and Joint Position Radio Box are hidden.

The `MultiListJointsAcivated` callback sets the left image according to the selected joint whose axial position is most forward, and the right image according to the second joint if two joints are selected. If only one item is selected, the image on the right and the Aft Joint Position Radio Box are hidden.

The `action_AddPart_act_cb` and `action_DeletePart_act_cb` callbacks work the same as the callbacks to add and delete Joints. However, it also verifies that either one or two joints are selected.

The `SubAssmChanged` callback toggles whether the sub-assembly Name Box is available.

If `Cancel_cb` is executed, then nothing happens and the dialog closes.

If `OK_cb` is executed, then the following code is executed.

```
this->WriteOutputFile();
this->CreateAssemblyAndControlStructure();

for(int i=0;Parts.size();i++)
{
      Parts[i]->CreatePart();
      Parts[i]->MakeWaveLinks();
      Parts[i]->SketchCrossSection();
}
```

The `CreateAssemblyAndControlStructure` and `WriteOutputFile` member functions are summarized below.

```
void InterfaceManagerUI::WriteOutputFile(){
      ofstream fout(//file path and name of default file);

      for(int i=0;i<JointObjects.size();i++)
      {
            JointObjects[i]->WriteMe(fout);
      }

      for(int i=0;i<Parts.size();i++)
      {
            Parts[i]->WriteMe(fout);
      }

      fout.close();
}
```

```
void InterfaceManagerUI::CreateAssemblyAndControlStructure(){
      // Create Assembly Part File
      // Create expressions for global variables
      //    (axial clearance and radial clearance)
      // Make and open Control Sketch


      // Make expressions and Geometry for each Joint Object
      // Keep track of which Joint Objects are conical
      std::vector<int> ConIndxs;
      for(int i = 0;i<JointObjects.size();i++)
      {
            JointObjects[i]->MakeExpressions();
            JointObjects[i]->MakeGeometry();
            if(JointObjects[i]->IsConical)
                  ConIndxs.push_back(i);
      }


      if(ConIndxs.size()==2) // if there are two conical joints
      {
            //constain conical flanges to be parallel
            //Create a dimension between the conical flanges to space
            //them apart by a default for the interstage thickness
      }

      //update and close control sketch

      //Create hole sketches
      for(int i = 0;i<JointObjects.size();i++)
      {
            JointObjects[i]->MakeHoleControls();
      }

}
```

In summary, Operation 1 completes the following steps:

1. Creates the assembly and component files
2. Creates the assembly control structure
3. Creates the associative links from the assembly to each child
4. Starts sketching the child cross sections

Figure 4-7 demonstrates what an example part cross section would look like after Operation 1 is complete. The thicker lines are the sketch lines in the part. The darker thin lines and points are the constraints and the circular lines from the hole sketches. The lighter thin lines are the control structure lines.

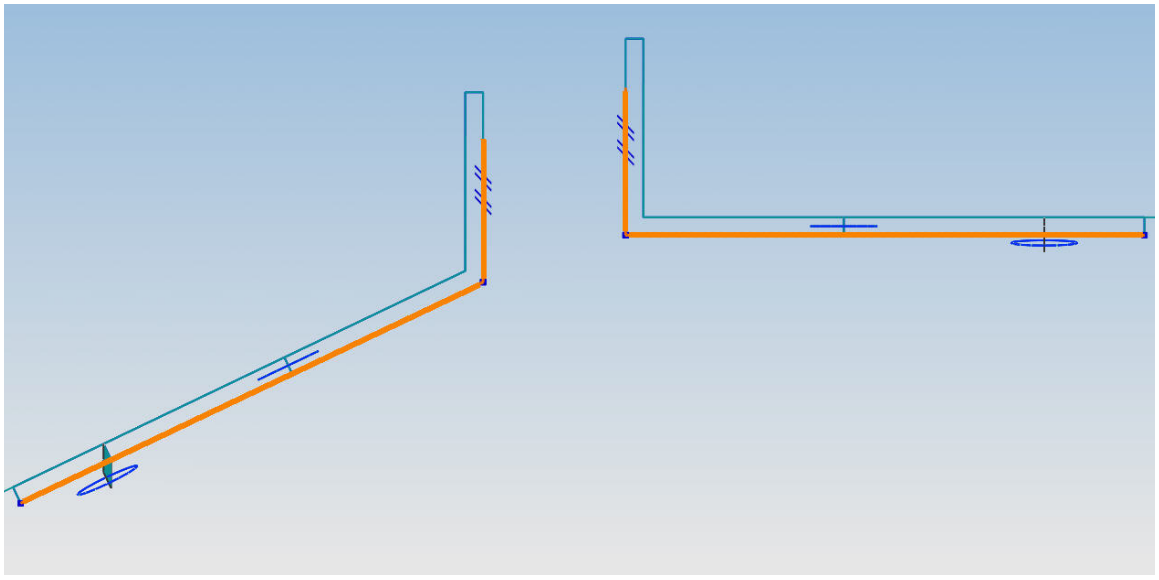


Figure 4-7: One component cross section

4.2 Operation 2: Detail features

Operation 2 contains four main functions. Each one traverses the entire assembly and performs the same routine for each component. Below is pseudo code that demonstrates how to traverse the assembly.

```
void Traverse(component1){
      SetWorkComponent(component1); //make component1 the active part
      subParts = component1->GetChildren();
      if(subParts.size==0) //ignore assembly parts
      {
            //call function to be executed on each part
      }
      for(all subParts)
      {
            Traverse(subParts[i]);
      }
}

int main(){
      //get top level assembly part
      Traverse(TopPart);
}
```

The four routines called by the traversing function are `Revolve`, `DetailFeatures`, `Extrude`, and `Pattern`.

4.2.1 Revolve

The `Revolve` routine simply queries the active part and finds the sketch feature named "CS". It then uses the "CurveFeatureRule" to add all of the sketch curves to the revolve section. This enables the function to work without knowing any of the names of the curves in the sketch. The axis of revolution is always the x-axis and the angle of revolution is always 360.

4.2.2 Detail features

The `DetailFeatures` routine queries the attributes in the active part to determine the number of joints in the part, their types, and their names (as described in section 4.1.3). Then the routine finds the edges of the part body that intersect the key vertices of each joint and creates a chamfer or fillet for each one. The key vertices and their associated operations are illustrated in Figure 4-8 and the example code is included following the image.

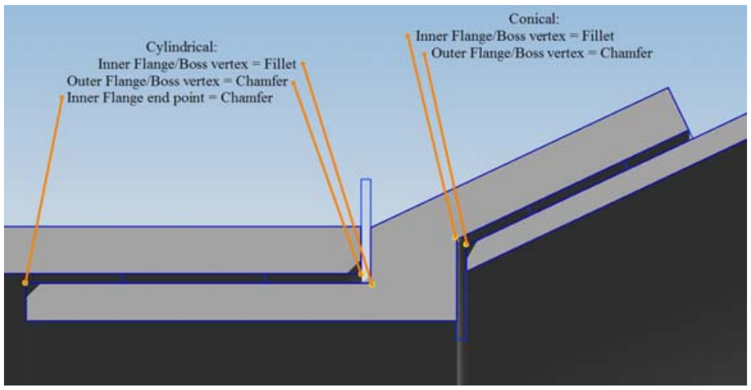

Cylindrical:
Inner Flange/Boss vertex = Fillet
Outer Flange/Boss vertex = Chamfer
Inner Flange end point = Chamfer
Conical:
Inner Flange/Boss vertex = Fillet
Outer Flange/Boss vertex = Chamfer

Figure 4-8: Fillet and chamfer rules

```
void DetailFeatures(){
  . . .Get Part Body. . .

  //Get number of joints from part attributes
  std::string numJoints = workPart->
      GetStringAttribute("JOINTS").GetText();
  int n = atoi(numJoints.c_str());
  //Get top level control sketch
  Sketch *ControlSketch(dynamic_cast<Sketch *>(displayPart->
      Sketches()->FindObject("SKETCH_CONTROL_STRUCTURE")));

  for(int j =1;j<=n;j++) //for each joint in the active part
  {
      //get joint id
      std::string JointID = workPart->
            GetStringAttribute("JOINT"+stringify(j)).GetText();
      //get joint type and separate the first 3 characters
      std::string JointType = workPart->
      GetStringAttribute("JOINT"+stringify(j)+"_TYPE").GetText();
      std::string shape = JointType.substr(0,3); //CON or CYL
      . . . get lines from ControlSketch . . .

      //get all edges
      std::vector<Edge *> bodyedges = theRevFeat->GetEdges();
      for(int i=0;i<bodyedges.size();i++) //for all edges
      {
            double distVertOut= GetMinDist(lineVertOut,bodyedges[i]);
            double distHorzOut=GetMinDist(lineHorzOut,bodyedges[i]);
            //if current edge intersects outer flange/boss vertex
            if((distVertOut<0.001)&&distHorzOut<0.001)
            {
                  ChamferEdge(bodyedges[i]);
                  continue;
            }
            double distVertIn = GetMinDist(lineVertIn,bodyedges[i]);
            double distHorzIn = GetMinDist(lineHorzIn,bodyedges[i]);
            //if current edge intersects inner flange/boss vertex
            if((distVertIn<0.001)&&distHorzIn<0.001)
            {
                  BlendEdge(bodyedges[i]);
                  continue;
            }
            double distXEnd = GetMinDist(lineXEnd,bodyedges[i]);
            //if current edge intersects inner flange end point
            if((distXEnd<0.001)&&distHorzIn<0.001)
            {
                  if(shape=="CYL") //only for cylindrical joints
                  {
                        ChamferEdge(bodyedges[i]);
                        continue;
}}}}
```

4.2.3 Extrude

The `extrude` routine searches through all of the "linked curve" features and finds each of the circles from the control structure. For each circle it finds, it calls the `ExtrudeFastenerHole` function which extrudes the hole and names the expressions and features according to the joint name and row number of the matching arc in the top level assembly. Code for both functions is included below with explanatory comments.

```
void Extrude(){
  . . . get part features (myFeatures) . . .
  for(int i=0;i<myFeatures.size();i++)
  {
   std::string type = myFeatures.at(i)->FeatureType().GetText();
   if(!type.compare("LINKED_CURVE"))// all waved control structures
   {
     Features::CompositeCurve *myCompCurve
         (dynamic_cast<Features::CompositeCurve *>(myFeatures.at(i)));
     for(int j=1;j<=10;j++) //each curve in the wave link feature
     {
        Arc *arc1(dynamic_cast<Arc *>(myCompCurve->
            FindObject("CURVE"+stringify(j))));
        Arc *nullArc(NULL);
        if(arc1!=nullArc) //each control curve that is a valid arc
        {
          double deltaAngle = arc1->EndAngle() - arc1->StartAngle();
          if(deltaAngle==2*PI) //if arc is a complete circle
          {
            ExtrudeFastenerHole(arc1);
}}}}}}
```

```
void ExtrudeFastenerHole(Arc *arc1){

        . . . create the extrude feature (called feature1). . .

 // get the collection of sketches from the top level assembly
 SketchCollection::iterator sket_it = displayPart->Sketches()->begin();

  //cycle through the top assembly sketches
  for(;!(sket_it==displayPart->Sketches()->end());sket_it++)
  {
      std::vector<NXObject *> sketchgeometry =(*sket_it)
            ->GetAllGeometry();
      Arc *arc2(dynamic_cast<Arc *>(sketchgeometry[0]));
      Arc *nullArc(NULL);
      if(arc2!=nullArc)  //get the valid arc
      {
            //get the assembly arc that matches the input arc
            Point3d currentCenter = arc2->CenterPoint();
            if(fabs(currentCenter.X - arc1->CenterPoint().X) < 0.1)
            {
                  std::string assmArcName = sketchgeometry[0]
                        ->Name().GetText();
                  //JointID = the arc name minus "HOLEARC"
                  std::string JointID =
                        assmArcName.substr(assmArcName.size()-7);
                  feature1->SetName(JointID+"_HOLE");
                  break;
}}}}
```

4.2.4 Pattern

One of the limitations in the NX Open C++ library is its inability to create feature patterns. Because of this, the `Pattern` routine uses the C language API and converts between the C "tag" objects and the C++ class objects. There are some objects however, that cannot be converted to tags. The new version of the Hole feature in NX 5 Release 3 could not be converted to a tag and this is the reason extruded sketches were used to create the fastener holes. There were other problems associated with the Pre-NX5-Release 3 Hole feature as well.

The `Pattern` routine first finds each of the extrude features in the active part whose name ends in "_Hole". For each one, it extracts the name of the joint object from the extrude feature name (the portion before "RX_Hole"). Then it creates a circular pattern referring to the expressions in the top level assembly which were created during Operation 1 as explained in section 4.1.1. More specifically, it sets the instance quantity to "myAssembly::*JointName*_N" and the angle to "360/myAssembly::*JointName*_N". The routine also renames the expressions created for the pattern feature to be meaningful. Example code for finding the extrude features and creating the patterns is as follows.

```
                //// FINDING THE EXTRUDE FEATURES ////
. . . get all of the features from the active part (myFeatures) . . .
for(int i=0;i<myFeatures.size();i++){
      std::string type = myFeatures.at(i)->FeatureType().GetText();
      //for all extrude features
      if(!type.compare("EXTRUDE"))
      {
        Features::Extrude *myExtrude =
           dynamic_cast<Features::Extrude *>(myFeatures.at(i));
        std::string name = myExtrude->Name().GetText();
        if(name.size()>5) //prevents crashing on non-named features
        {
          // if name ends in “_Hole”
          if(!name.substr(name.size()-5).compare("_HOLE"))
          {
            // get the next feature as well
            i++;
            Features::Extrude *myExtrude2 =
                dynamic_cast<Features::Extrude *>(myFeatures.at(i));
            // call the pattern routine for both features
            PatternFastenerHoles(myExtrude,myExtrude2);
}}}}
```

```
         //// CREATING THE PATTERN AND RENAMING EXPRESSIONS ////
void PatternFastenerHoles(Features::Extrude *R1extrudeFeature,
                        Features::Extrude *R2extrudeFeature){

  . . .  convert feature objects to tags and add to feature_list . . .
            . . .  create other input data structures . . .

  //extract joint ID from extrude feature name
  ExtrudeName = R1extrudeFeature->Name().GetText();
  //Joint name = extrude name minus “RX_HOLE”
  FirstJointName = ExtrudeName.substr(0,ExtrudeName.size()-7);

  //create the expressions for the instance number and spacing angle
  number_str = "myAssembly::"+FirstJointName+"_N";
  Ang_str = "360/myAssembly::"+FirstJointName+"_N";

  // make the pattern feature with other inputs
  UF_MODL_create_circular_iset(... ,number_str,Ang_str,feature_list);

  //get iterator of the expression collection in the active part
  NXOpen::ExpressionCollection::iterator exp_it =
      workPart->Expressions()->begin();

  //cycle through all expressions
  for(;exp_it!=workPart->Expressions()->end();exp_it++)
  {
      name = (*exp_it)->Name().GetText();
      RHS = (*exp_it)->RightHandSide().GetText();
      //if right hand side matches number_str
      //and name does not match “JointName_N”
```

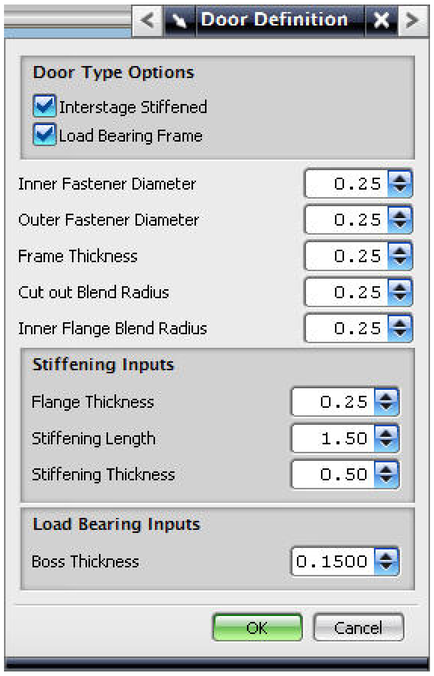
```
       if(!RHS.compare(number_str)&&(name.compare(FirstJointName+"_N")))
       {
        //rename expression from p123 to "JointName_N"
        workPart->Expressions()->Rename((*exp_it), FirstJointName+"_N");
        break;
       }
   }

   //the rename function reorders the collection of expressions,
   //so after the first expression is renamed, the loop was restarted
   exp_it= workPart->Expressions()->begin();
   for(;exp_it!=workPart->Expressions()->end();exp_it++)
   {
       name = (*exp_it)->Name().GetText();
       RHS = (*exp_it)->RightHandSide().GetText();
       if(!RHS.compare(Ang_str)
             &&(name.compare(FirstJointName+"_FastAng")))
       {
        //renames p123 to "JointName_FastAng"
        workPart->Expressions()->
             Rename((*exp_it),FirstJointName+"_FastAng");
        break;
}}}
```

4.3 Operation 3: Insert door

Operation 3 allows the user to insert an access door on the interstage including the door frame and the stiffening alterations on the interstage body. There are two sub tasks involved in Operation 3. The first creates the door and door frame components and stiffens the interstage. The second creates the fastener holes for all three components.

4.3.1 Operation 3a: Components

When Operation 3a is executed, a GUI is opened to retrieve inputs from the user for defining the type of door frame and whether the interstage should be stiffened (See Figure 4-9). The GUI also retrieves inputs for the fastener sizes, the corner radii, the thicknesses of the frame and the stiffened cross section. Using dialog callback functions, the Load Bearing Inputs section and the Stiffening Inputs section is hidden if the associated check boxes are unchecked. The inputs from the GUI are stored as global variables so they can be accessed by any of the functions within the program. Once OK is clicked, the dialog closes and the user is asked to select the sub-assembly into which the door and door frame components are then added. Expressions are also created in the components using the input variables.


Door Definition

Door Type Options

- [✓] Interstage Stiffened
- [✓] Load Bearing Frame

Inner Fastener Diameter	0.25
Outer Fastener Diameter	0.25
Frame Thickness	0.25
Cut out Blend Radius	0.25
Inner Flange Blend Radius	0.25

Stiffening Inputs

Flange Thickness	0.25
Stiffening Length	1.50
Stiffening Thickness	0.50

Load Bearing Inputs

Boss Thickness	0.1500

Figure 4-9: Door definition GUI

Since this operation will be executed for each door, it must use a naming convention that provides unique object and expression names for each door. An expression in the interstage sub-assembly keeps track of the number of doors and an appropriate prefix is added to each name i.e. “Door1”, “Door2” etc. A global variable is also created so that each sub-function in the application can reference the number of doors as needed.

Interstage

As described in Chapter 3, this operation requires geometric inputs from the user. There must be a sketch in the interstage component that represents the cutout region. The sketch must be created on a plane that is tangent to the surface of the interstage and must contain a closed, concave loop of lines. The second geometric input is the line in the cross section that will be revolved to create the IML surface. When the operation begins, the user is also prompted to select the interstage component and its sub-assembly to ensure that the correct part files are modified.

The first step in altering the interstage is to extrude the cutout sketch and apply blends to its extruded edges. The extrude function, like the revolve routine discussed in section 4.2.1, uses the "CurveFeatureRule" to add all of the lines from the selected sketch to the section definition of the extrude function. This eliminates the need to name the sketch lines. After the sketch is extruded into a sheet body, the blend function needs to identify the correct edges on the sheet body. This is done by comparing the direction of each edge to the direction of the extrude feature. Sample code of this procedure is summarized below.

```
  //// COLLECTING EDGES WHOSE DIRECTION MATCHES EXTRUDE DIRECTION ////
std::vector<Edge *> edges4;
//loop though all edges in the feature
for(int i=0;i<allEdges.size();i++)
{
      // get vertices of current edge
      allEdges[i]->GetVertices(vertex1,vertex2);
      //calculate magnitude of edge vector
      double mag = sqrt(pow(vertex1->X - vertex2->X,2)
            +pow(vertex1->Y - vertex2->Y,2)
            + pow(vertex1->Z - vertex2->Z,2));
      //calculate unit edge vector
      Vector3d edgeUnitVect((vertex1->X - vertex2->X)/mag,
            (vertex1->Y - vertex2->Y)/mag,
            (vertex1->Z - vertex2->Z)/mag);
      //compare edge vector to extrude vector
      if(fabs(edgeUnitVect.X - extrudeVect.X)<.0001)
      {
            //add current edge to collction if vectors match
            edges4.push_back(allEdges[i]);
      }
}
```

The next step in Operation 3a is to create the offset surfaces from the IML surface and from the cutout surface. If the user selects the non-stiffened interstage option, then the only offsets that are needed are for the surfaces at the beginning of the flange and at the outer mold line (OML). Otherwise, offsets are needed to represent the entire boundary of the stiffening body that will be united to the interstage body.

The only difficulty in creating the offset surface features is verifying that the offset directions are correct. In order to do this, one offset surface is created and its position is compared to the position of the source surface. If the position is incorrect, the offset feature is edited to flip the offset direction. A Boolean variable keeps track of whether the first offset was flipped or not and applies the same flip condition to all subsequent offsets. These segments of the code are shown below.

```
          ////VERIFY THAT OFFSETS ARE IN CORRECT DIRECTION////
bool FlipOffset = false;    //boolean variable to track flip condition

//get maximum radius of the outermost vertex
//of the line used to create the IML surface
double IMLR1 = sqrt(lineIR->StartPoint().Y*lineIR->StartPoint().Y
      + lineIR->StartPoint().Z*lineIR->StartPoint().Z);
double IMLR2 = sqrt(lineIR->EndPoint().Y*lineIR->EndPoint().Y
      + lineIR->EndPoint().Z*lineIR->EndPoint().Z);
double IMLR = max(IMLR1,IMLR2);

//get outermost vertex on offset surface
double offsetR,R1,R2;
offsetR=R1=R2= 0;
//get offset surface edges
std::vector<Edge *> offsetEdges = OMLOffset->GetEdges();
//loop over all edges
for(int i =0;i<offsetEdges.size();i++)
```

```
{
      //get vertices of currect edge
      offsetEdges[i]->GetVertices(vertex1,vertex2);
      //calculate radii of vertices
      R1 = sqrt(vertex1->Y*vertex1->Y+vertex1->Z*vertex1->Z);
      R2 = sqrt(vertex2->Y*vertex2->Y+vertex2->Z*vertex2->Z);
      //if either radius is the new maximum
      if(max(R1,R2)>offsetR)
      {
            offsetR=max(R1,R2); //set offsetR equal to it
      }
}
//if offsetR is less than IMLR, reverse the direction of the IML
if(offsetR<IMLR)
{
      ReverseOffsetDirection(FlipOffset);
}
```

The offset surfaces represent all of the boundaries for the stiffening region that are normal to either the cutout surface or the IML surface. For the rest of the boundaries, a routine is called to generate a swept surface using intersection curves as guides. Here is an illustration of the inputs and results for this routine followed by its pseudo-code.

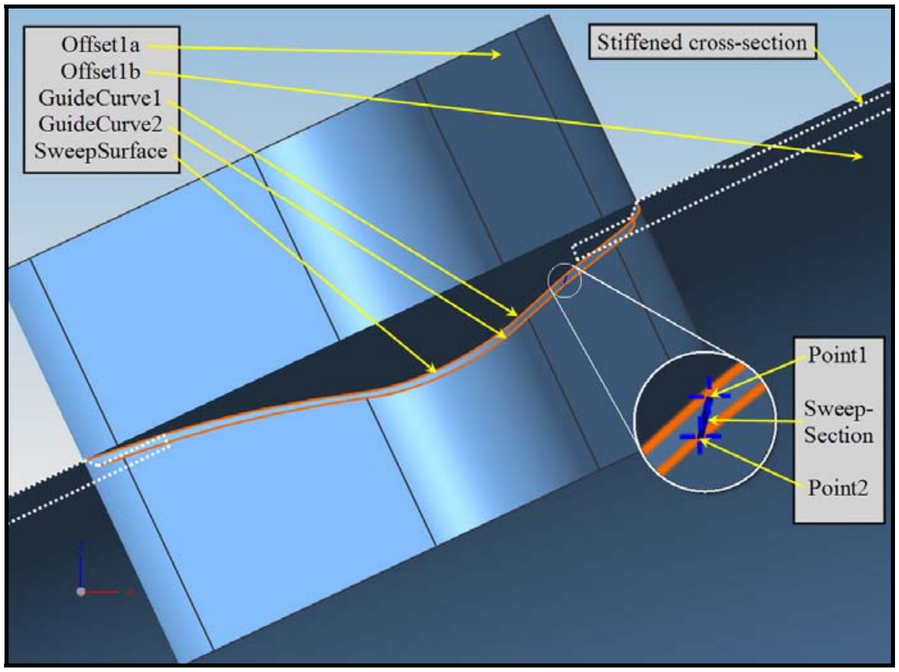


Figure 4-10: Inputs and results of swept surface routine

```
SweptSurface(Offset1a,Offset1b,Offset2a,Offset2b){
      GuideCurve1 = Intersection(Offset1a,Offset1b);
      GuideCurve2 = Intersection(Offset2a,Offset2b);
      Point1 = Intersection(GuideCurve1,XZPlane);
      Point2 = Intersection(GuideCurve2,XZPlane);
      SweepSection = Line(Point1,Point2);
      SweptSurface = Sweep(SweepSection,GuideCurve1,GuideCurve2);
}
```

When creating the sweep feature, the guide curves must be pointing in the same direction. To verify this, compare the signs of the Y and Z components of the guide direction vectors. If either of the component signs do not match, then execute the `ReverseDirectionOfClosedLoop` API command.

Now that there are surfaces which bound the entire stiffening region, they must be trimmed to exclusively represent the boundary surfaces of the stiffening body as discussed in Chapter 3. Except for finding the region points, the trimming operation is straightforward. The function inputs include the set of surface bodies to be trimmed, the set of faces that will trim them, and the set of points representing the regions to either be retained or discarded. The bodies and faces have already been created at this point and just need to be grouped into sets. The easiest way to obtain the set of region points is to use the vertices of the edges on the source bodies because it is always known whether the regions containing the edges should be retained or discarded. Here is the example code of finding the region points.

```
                    //// FINDING REGION POINTS ////

//if region point can be any vertex on the source body
Point3d Pnt1(0,0,0);
Point3d Pnt2(0,0,0);
SourceFace->GetEdges()[0]->GetVertices(&Pnt1,&Pnt2);
trimmedSheet = TrimSheet(Body,Face,Pnt1,bool_keep,"featureName");

//if region point must be the innermost vertex on the source body
std::vector<Edge *> ex_edges = SourceFace->GetEdges();
```

```
Point3d Pnt1 = GetMinREdgePoint(ex_edges);
trimmedSheet = TrimSheet(Body,Face,Pnt1,bool_keep,"featureName");


//the function for getting the innermost vertex
//it is trivial to modify this function to return the outermost vertex
Point3d GetMinREdgePoint(std::vector<Edge *> edges){
  //std::vector<Edge *> edges = trimmed_cutOut->GetEdges();
  double R = 100000;
  double tempR1,tempR2;
  Point3d minPoint;
  Point3d vertex1(0, 0, 0);
  Point3d vertex2(0, 0, 0);
  for(int i = 0;i<edges.size();i++)
  {
      edges[i]->GetVertices(&vertex1,&vertex2);
      tempR1 = sqrt(vertex1.Y*vertex1.Y+vertex1.Z*vertex1.Z);
      tempR2 = sqrt(vertex2.Y*vertex2.Y+vertex2.Z*vertex2.Z);


      if(tempR1<R)
      {
        minPoint.X  = vertex1.X;
        minPoint.Y  = vertex1.Y;
        minPoint.Z  = vertex1.Z;
        R = tempR1;
      }
      if(tempR2<R)
      {
        minPoint.X  = vertex2.X;
        minPoint.Y  = vertex2.Y;
        minPoint.Z  = vertex2.Z;
        R = tempR2;
      }
  }
  return minPoint;
}
```

The last steps in altering the interstage component are also straightforward. The interstage body should first be trimmed by the offset surfaces to remove the cutout region and any region where the stiffening portion is thinner than the body. Figure 4-11 shows the two bodies before and Figure 4-12 shows the bodies after the trimming operation. The trimmed boundary surfaces of the stiffening body should then be sew together and united with the interstage body through a Boolean operation (see Figure 4-13).

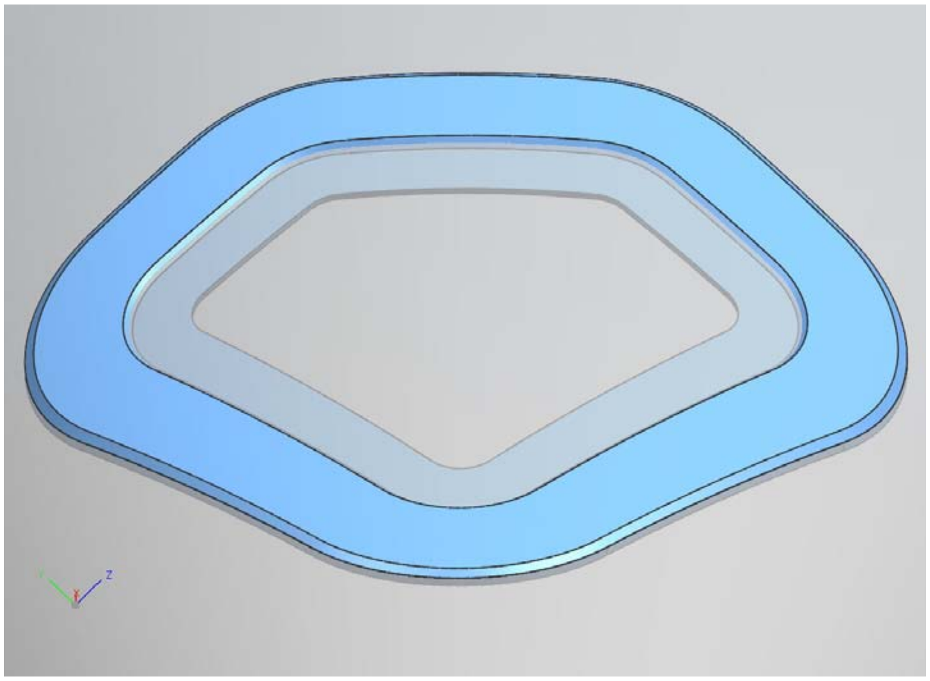


Figure 4-11: The interstage body (partially transparent) and the stiffening body before trimming

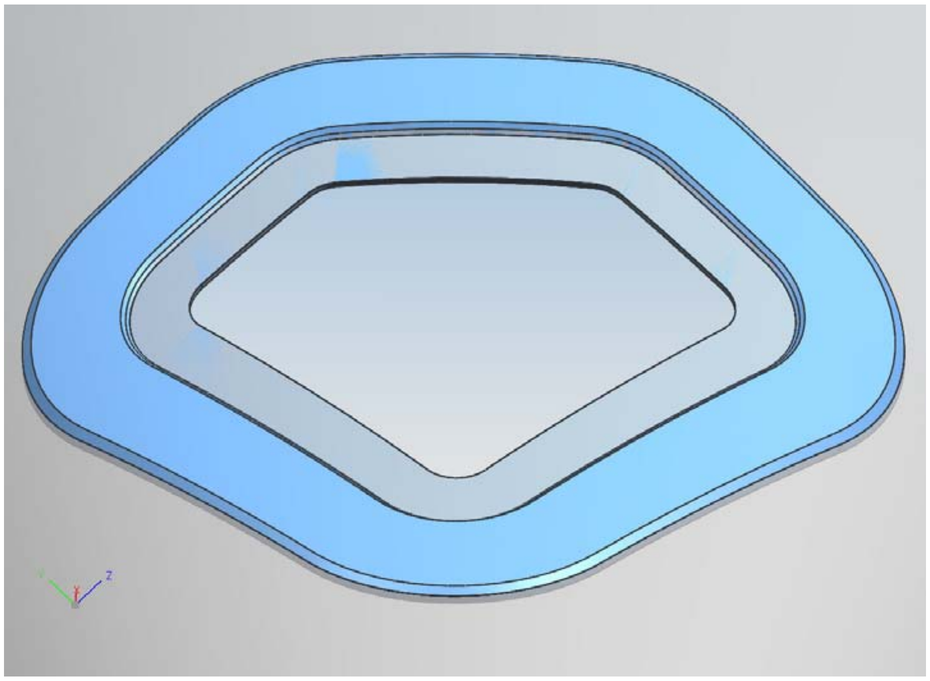


Figure 4-12: The trimmed interstage body and the stiffening body

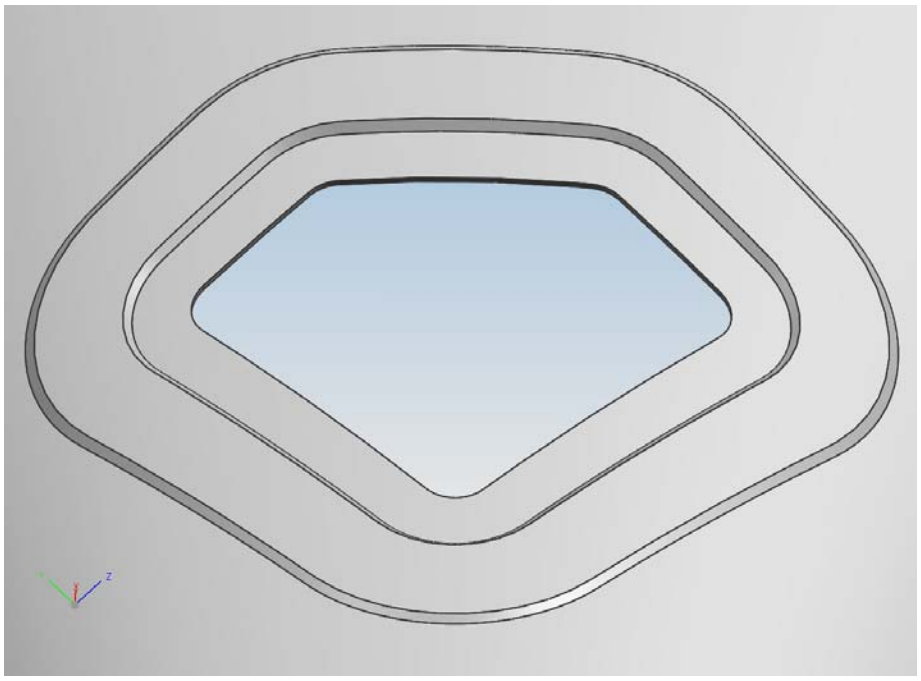


Figure 4-13: United interstage body and stiffening body

Frame

The first step in creating the frame component body is wave linking the key surfaces from the interstage component. This is another capability that is lacking in the NX Open C++ API. Creating the wave linked faces is easily accomplished in the C-language API though and does not need much discussion other than to mention that the surfaces that should be wave linked are 1) the IML surface, 2) the cutout surfaces, 3) the outer flange surface, and 4) the surfaces around the beginning of the flange (See Figure 4-14).

The offset surfaces on the door frame are created in the same manner as discussed for the interstage offset surfaces. First, there are offsets created from the IML surface and from the cutout surface to provide clearances between the frame and the interstage.

Depending on the type of frame specified by the user, other offset surfaces are created to bound all of the surfaces on the boundary (See Figure 4-15).

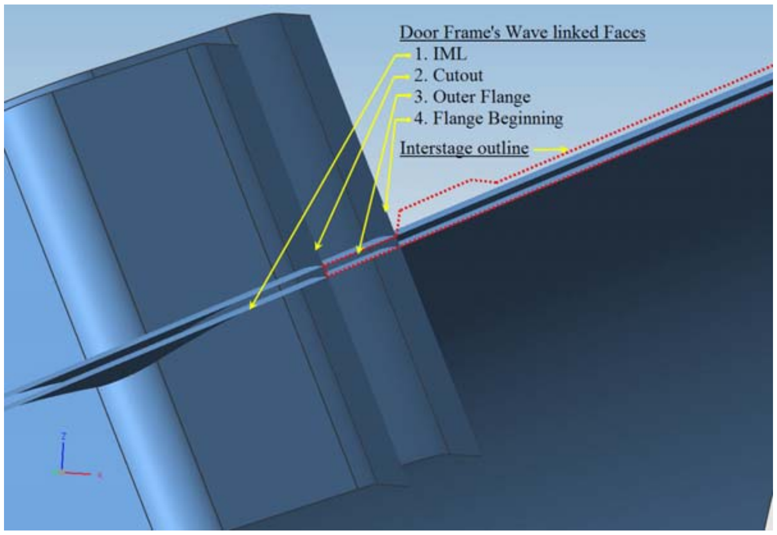


Figure 4-14: Wave linked faces in the door frame

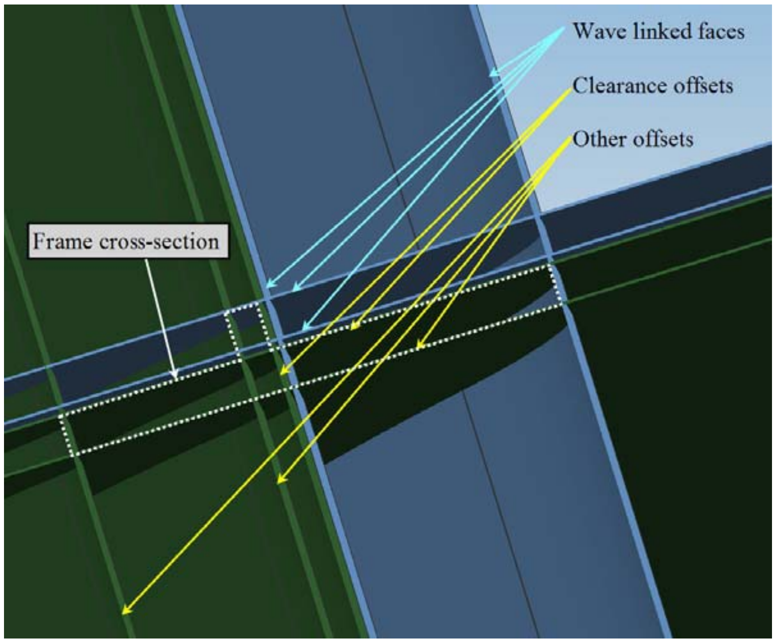


Figure 4-15: Frame offset surfaces

One issue comes up when creating an inward facing offset with a value that is larger than the blend radius of the original cutout blends. As illustrated in Figure 4-16, which is aligned with the extrude direction, the *inner flange offsets*' blended faces would fail and the linear faces would cross each other. To correct this issue, offsets are only applied to the linear faces and then face blend features are created between adjoining surfaces.

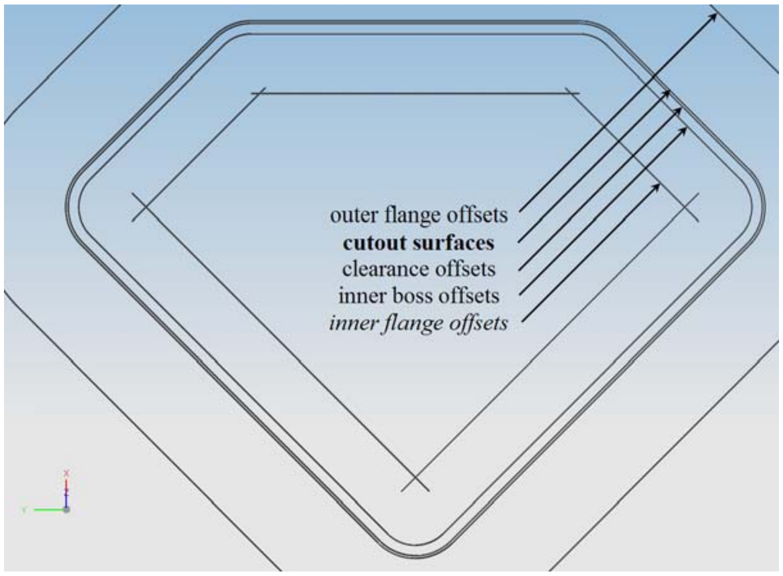


Figure 4-16: Large inward facing offsets need face blend feature

For the face blend features to work properly, the faces must be ordered by proximity. This was accomplished by first collecting the top edge of each *inner flange* offset surface, and then finding the edge whose vertex is closest to one of the vertices on the first edge. The indexes of the faces are added to a list and the process repeats with each new edge while ignoring the indexes that have already been listed. After all the faces have been added to the list, a face blend feature is created between each face and the face listed next in the list. Finally a face blend feature is created between the first and last face in the list. The code for creating the ordered list of indexes and for creating the face blend features is shown below.

```
        //// ORDERING FACES AND INSERTING FACE BLEND FEATURES ////
                . . . get top edges of offset faces . . .
//get ordered list of indexes
std::vector<int> SortedIndexes;
SortedIndexes.push_back(0);
for(int i = 0;i<topEdges.size()-1;i++)
{
      GetClosestEdge(topEdges,SortedIndexes,SortedIndexes.back());
}
//Blend each face to its neighbor
for(int i=0;i<InnerFlangeFaces.size()-1;i++)
{
      FaceBlend(InnerFlangeFaces[SortedIndexes[i]],
            InnerFlangeFaces[SortedIndexes[i+1]],"InnerFlangeBlendR");
}
FaceBlend(InnerFlangeFaces[SortedIndexes[0]],
      InnerFlangeFaces[SortedIndexes.back()],"InnerFlangeBlendR");
```

```
               // FUNCTION FOR GETTING NEXT CLOSES EDGE //
void GetClosestEdge(std::vector<Edge *> edges,
                     std::vector<int> &SortedIndexes,
                     int index){
  //get the vertices of the key edge
  Point3d v1(0, 0, 0);
  Point3d v2(0, 0, 0);
  edges[index]->GetVertices(&v1,&v2);
  double MinDist = 1000;
  int closestEdgeIndex;

  for(int i=0;i<edges.size();i++) //loop over all edges
  {
      // determine whether current index is already in the sorted list
      bool ignore = false;
      for(int j = 0;j<SortedIndexes.size();j++)
      {
        if(SortedIndexes[j]==i)
              ignore=true;
      }

      // if it is not then...
      if(!ignore)
      {
        // get the vertices on the current edge
        Point3d tempv1(0, 0, 0);
        Point3d tempv2(0, 0, 0);
        edges[i]->GetVertices(&tempv1,&tempv2);
       //calculate distance from each current vertex to each key vertex
        double dist1 = sqrt( pow(v1.X-tempv1.X,2)
            + pow(v1.Y-tempv1.Y,2) + pow(v1.Z-tempv1.Z,2) );
        double dist2 = sqrt( pow(v2.X-tempv1.X,2)
            + pow(v2.Y-tempv1.Y,2) + pow(v2.Z-tempv1.Z,2) );
        double dist3 = sqrt( pow(v1.X-tempv2.X,2)
            + pow(v1.Y-tempv2.Y,2) + pow(v1.Z-tempv2.Z,2) );
        double dist4 = sqrt( pow(v2.X-tempv2.X,2)
            + pow(v2.Y-tempv2.Y,2) + pow(v2.Z-tempv2.Z,2) );
        //reset closestEdgeIndex and MinDist if closer edge is found
        if (dist1<MinDist){
            MinDist=dist1;
            closestEdgeIndex=i;
        }
        if (dist2<MinDist){
            MinDist=dist2;
            closestEdgeIndex=i;
        }
        if (dist3<MinDist){
            MinDist=dist3;
            closestEdgeIndex=i;
        }
        if (dist4<MinDist){
            MinDist=dist4;
            closestEdgeIndex=i;
  }}}
  SortedIndexes.push_back(closestEdgesIndex);
}
```

Figure 4-17 illustrates the results after creating face blends on the *inner flange offsets* from Figure 4-16. If the offset surfaces were originally ordered as shown, the `SortedIndexes` vector would end up being [0,1,3,4,2] and face blends would be created between surfaces 0 and 1, 1 and 3, 3 and 4, 4 and 2, and finally 0 and 2.

After the offset surfaces and face blend features have been created, the surfaces are trimmed and sewn together using the same methods discussed for the interstage (See Figure 4-18 and Figure 4-19)

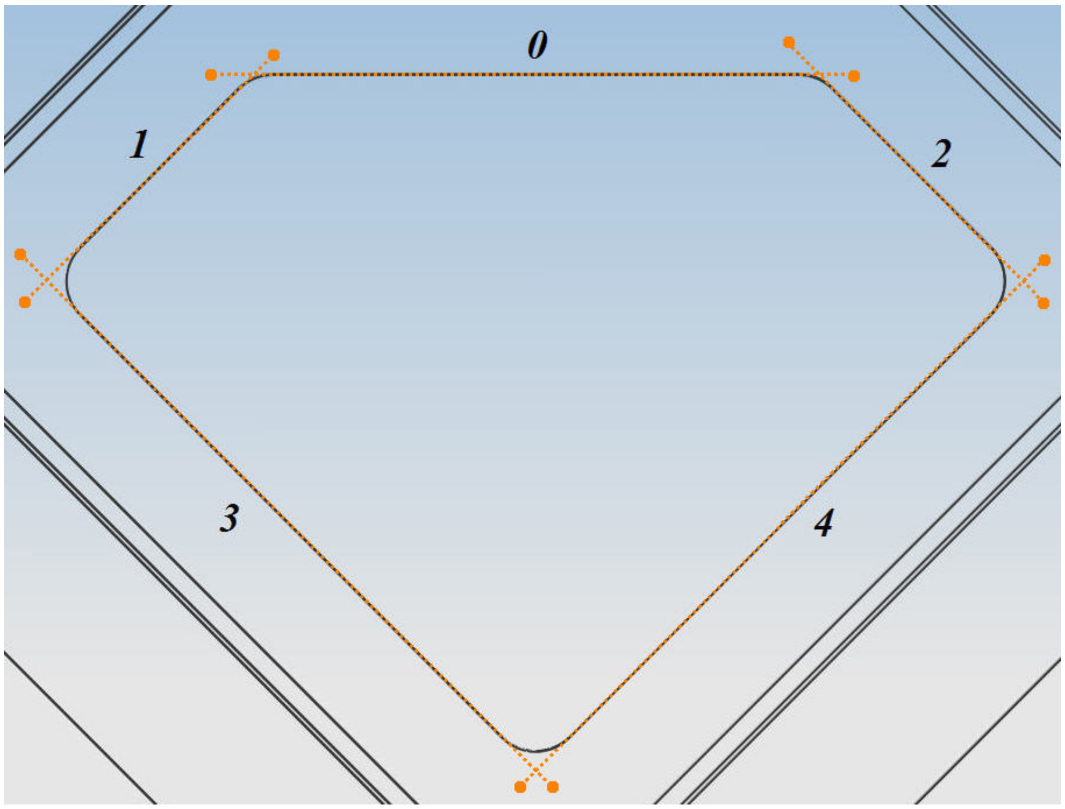


Figure 4-17: Face blend example

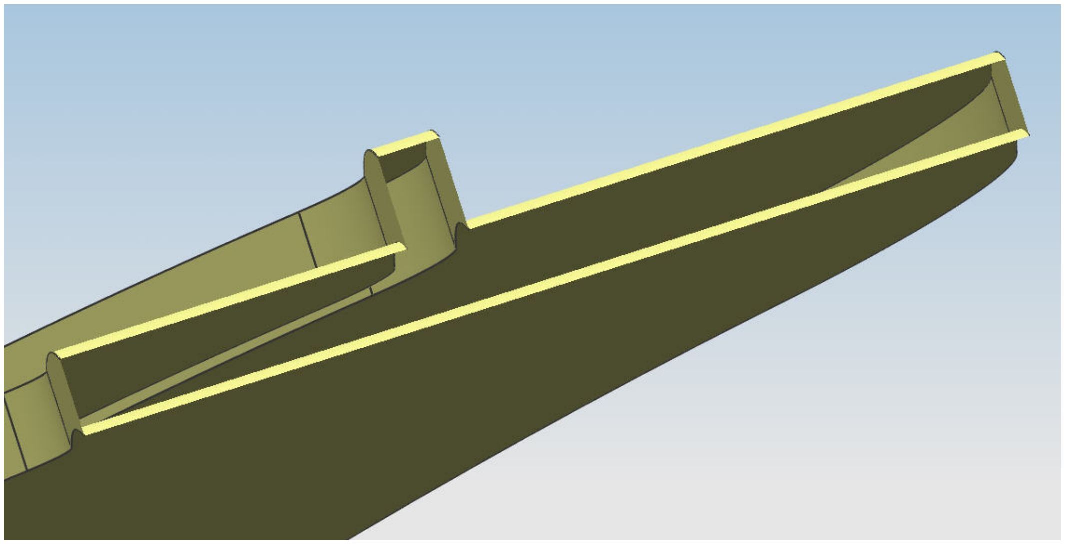


Figure 4-18: Trimmed surfaces on frame

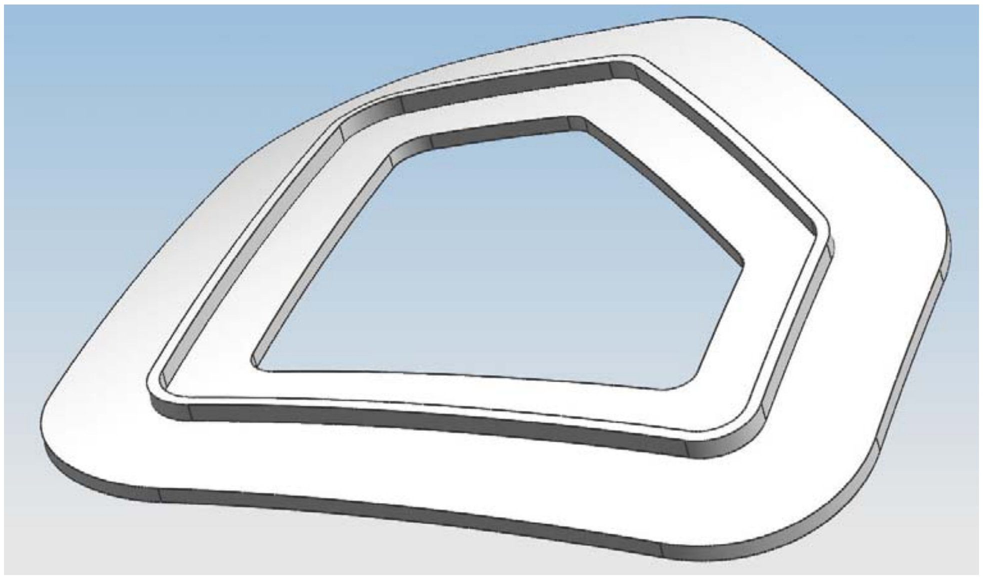


Figure 4-19: Door frame surfaces sewn into solid body

Door

There were only a couple additional issues in creating the door component and they both involved the wave link features. First, it should be noted that the specific surfaces wave linked to the door component depend on which type of door frame was selected by the user. If the user selected a non-load-bearing door frame, then the cutout surface from the interstage should be linked to the door component instead of the inner boss offset surfaces from the frame component.

The second issue was that a separate feature is created in the part history tree for each wave linked face and offset surface patch in the frame component. In other words, if the original door cutout sketch included 4 lines, then there would be separate offset surface features and wave link features for each of the 4 faces plus one for each of the 4 blended faces. This requires the function that creates the door component to know how many surface features it needs to link to.

This issue was resolved by outputting the number of sides from the function that creates the door frame component and inputting it into the function that creates the door component. Thus, the door function was able to link the correct number of surfaces based on the number of sides in the cutout sketch.

Results

The completion of Operation 3a results in a modified interstage having a cutout of user-specified shape with or without a stiffened cross section. It also results in either a load-bearing or non load-bearing door frame following the arbitrary shape and a door component that also fits the frame with tolerances. Operation 3a also creates all of the necessary associative links between the three components and the parameters that will be used in Operation 3b to insert fastener holes of the specified size. Figures 4-20 through 4-23 show top views and section views of two door examples with different user options.

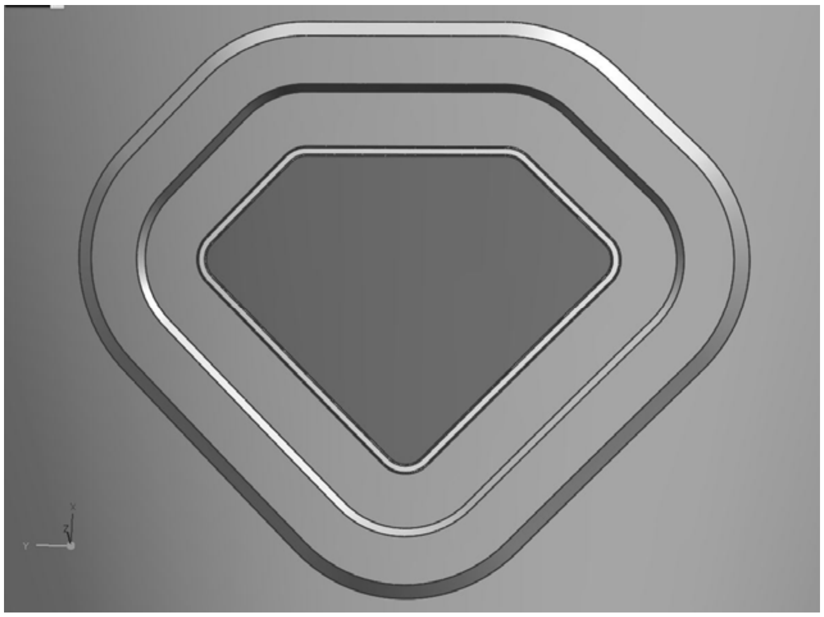


Figure 4-20 Top view of stiffened load-bearing door configuration

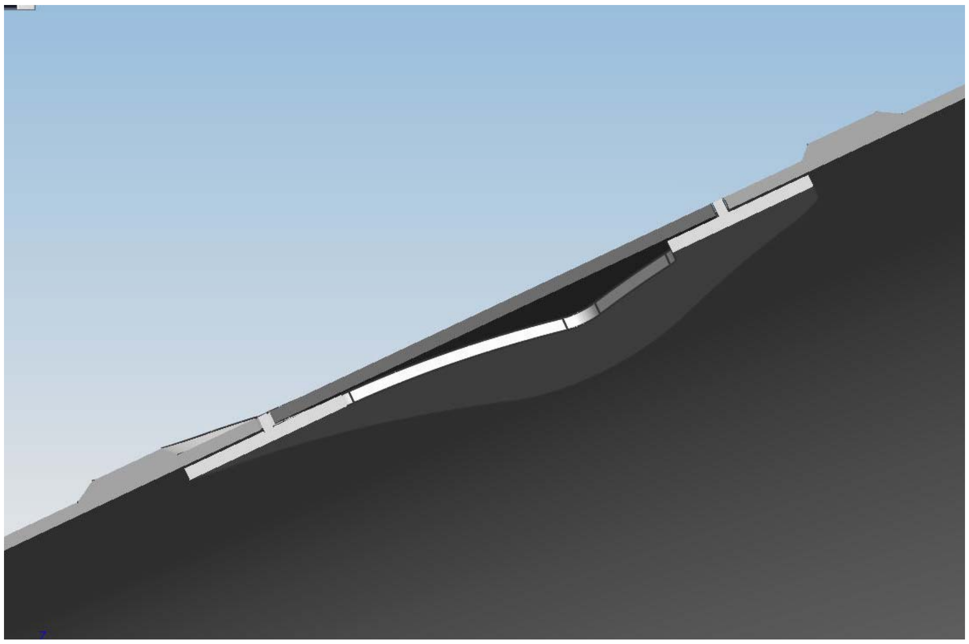


Figure 4-21: Section view of stiffened load-bearing door configuration

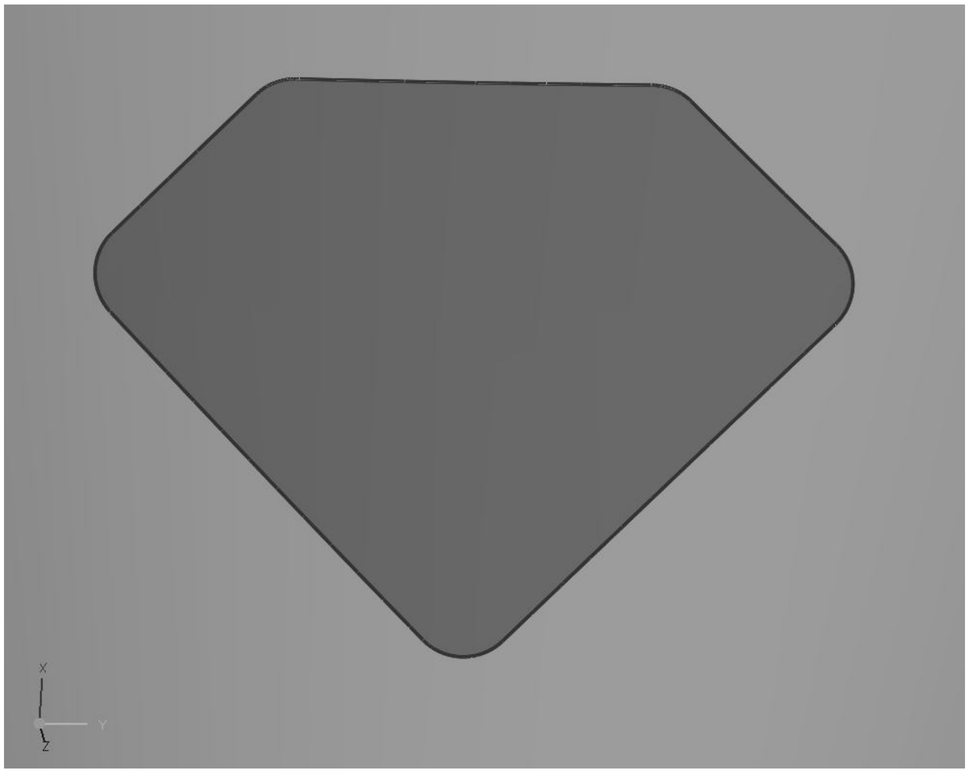


Figure 4-22: Top view of un-stiffened, non load-bearing door configuration

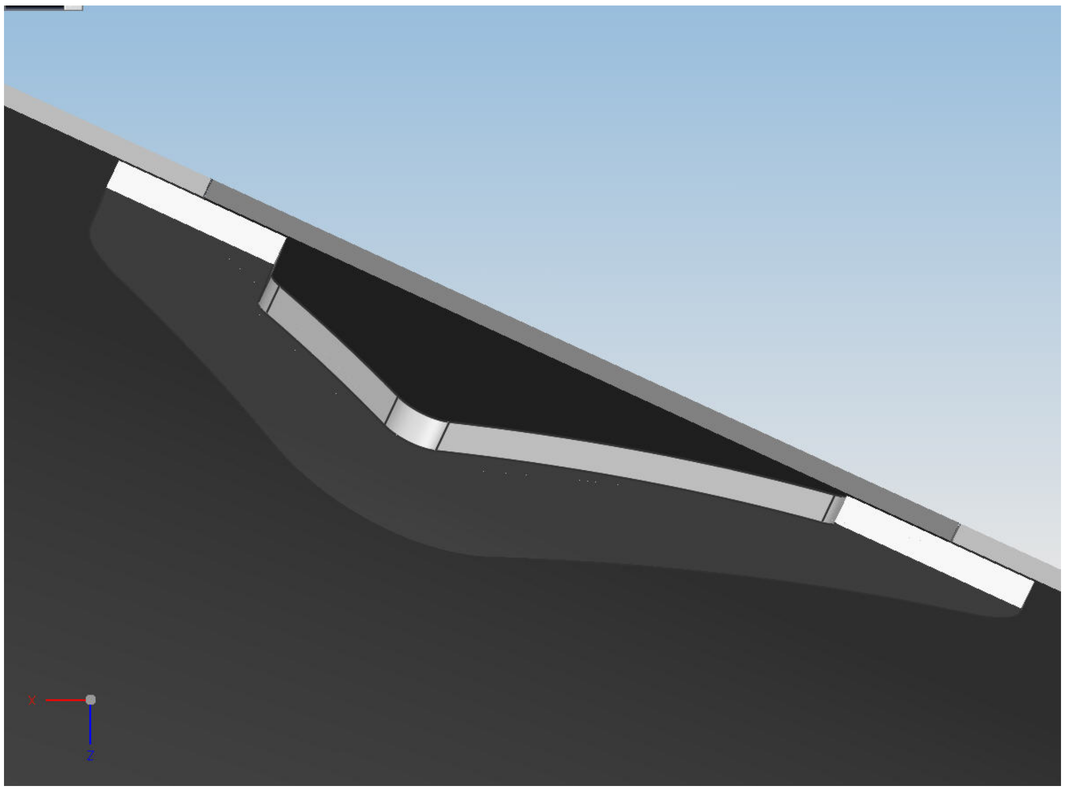


Figure 4-23 Section view of un-stiffened, non-load bearing door configuration

4.3.2 Operation 3b: Door fasteners

Operation 3b relies on Operation 3a to create the part expressions and part bodies. It is separated as a unique application so that the fastener details can be inserted into the model when the designer wishes. This enables the designer to alter the design and make modifications to the components before including the full details of the fastener holes.

There are a few over-arching routines utilized in Operation 3b that are repeated for each set of fastener holes. They are 1. MakeOffsetHoles, 2. CreateSimpleHole, 3. WaveLinkPoint, and 4. ProjectPoints. The pseudo-code below summarizes how these routines are executed in the main operation.

```
void do_ugopen_api()
{
       . . . Get Door Component (gDoor_comp), Door Frame Component
      (gFrame_comp), and Interstage component (gInt_comp) from user
                            selections. . .

  //Fasteners between interstage and door frame
  SetWorkComponent(gInt_comp);
  MakeOffsetHoles(... outer flange inputs ...,interstagePoints);

  SetWorkComponent(gFrame_comp);
  std::vector<Point *> FrameLinkedPoints; //output vector
  for(interstagePoints)
  {
      WaveLinkPoint(interstagePoints[i],gFrame_comp,FrameLinkedPoints);
  }
  CreateSimpleHole(FrameLinkedPoints,... outer flange inputs ...);

  //Fasteners between door frame and door
  MakeOffsetHoles(... inner flange inputs ...,framePoints);

  SetWorkComponent(gDoor_comp);
  std::vector<Point *> DoorLinkedPoints; //output vector
  for(framePoints)
  {
      WaveLinkPoint(framePoints[i],gDoor_comp,DoorLinkedPoints);
  }
  Features::ProjectCurve *projFeat = ProjectPoints(DoorLinkedPoints);
  CreateSimpleHole(projFeat,... inner flange inputs ... );
}
```

MakeOffsetHoles

As described in Chapter 3, the first step in adding fastener details is creating an offset curve of the cutout edge on the flange surface. The easiest way to find the correct edge and face is to prompt the user to select them manually. Since this was a challenging aspect of the program, the example code is included below. The MaskTriple object is the key to ensuring the correct type of object is selected. Other object type definitions can be found in “uf_ui_types.h”.

```
                   //// Get Selection From User ////
UI *theUI = UI::GetUI();
Selection *thisSelection = theUI->SelectionManager();
thisSelection->initialize();

// set selection mask to only allow the correct type of object
std::vector<Selection::MaskTriple> mask_array;
Selection::MaskTriple mask1 = Selection::MaskTriple::MaskTriple
```

```
  (UF_solid_type,UF_all_subtype,UF_UI_SEL_FEATURE_ANY_EDGE);
                                      // or ..._ANY_FACE...
mask_array.push_back(mask1);
NXObject *selectedObject1;
Point3d cursor(0,0,0);

// get the selection from the user
Selection::Response response1 = thisSelection->SelectObject
  ("Select the forward outer flange edge","", SelectionScopeWorkPart,
  SelectionActionClearAndEnableSpecific,false,false,mask_array,
  &selectedObject1,&cursor);
// verify the response and convert the retrieved object
if(response1==Selection::ResponseObjectSelected)
      Edge *cutoutedge = dynamic_cast<Edge *>(selectedObject1);
```

This method was also used in the main function to obtain the door and door frame components as well as the interstage component. The prefix used in naming objects and retrieving objects from the existing components is extracted from the name of the selected door component.

After retrieving the edge and face from the user, the offset in face feature is created with the offset distance being a function of the diameter for the current set of fasteners. The diameter is retrieved from existing expressions in the part. As with the other offset features, the direction must be verified. In order to do this, the user is asked to select the forward most edge of the cutout, and then the routine finds the forward most vertex on the offset curves. If the forward most vertex on the offset curves is not more forward than the selected edge, then the offset direction is reversed.

The next step in the MakeOffsetHoles routine, is creating evenly spaced points along the offset curves. Because the NX feature that adds arrays of points to a curve is non-associative, individual Smart Point features need to be created at specific parameter values along the curves. For each curve segment of the offset feature, the length of the curve is queried, and the number of points is calculated based on the hole diameter and the curve length to leave a certain spacing between each hole.

Since the curve segments share vertices, only every other curve should have points at its ends. The number of points (N) on a curve including the endpoints is calculated by Equation 4.1 and the number of points (N) on a curve excluding the endpoints is calculated by Equation 4.2

$$N = \mathrm{ceil}(L/(3.5 * D)) + 1 \qquad \textbf{(4-1)}$$

$$N = \mathrm{ceil}(L/(3.5 * D)) - 1 \qquad \textbf{(4-2)}$$

where L is the length of the curve, D is the fastener diameter, and ceil represents the ceiling function. Since N needs to be a discreet value, the spacing will not be an even multiple of the diameter, but is within 5-10% of 3 diameters.

In order to distinguish which curves should have points at the ends, the curves in the offset feature are ordered by length. The longest half of the curves (the sides of the cutout) will have points at their ends and the shortest half (the corners of the cutout) will not. As each point is created it is added to a collection. Then once all of the points have been added to each curve of the feature, the CreateSimpleHole routine is called on the collected points.

CreateSimpleHole

The CreateSimpleHole routine creates the NX 5-Release 2 version of hole feature. This new hole feature is able to create holes which are normal to the nearest solid surface at the specified points. This routine allows for two different methods of specifying the input points. It can take in either a vector of points, or a feature which contains points. The code for adding the points to the “HolePackageBuilder” as it is called in the C++ API is included below.

```
if(FeaturePoints) //if points are input from a feature
{
  std::vector<Features::Feature *> features1(1);
  features1[0] = feat; //the input feature
  FeaturePointsRule *featurePointsRule1 = workPart->ScRuleFactory()->
      CreateRuleFeaturePoints(features1);

  holePackageBuilder1->HolePosition()->AllowSelfIntersection(true);

  std::vector<SelectionIntentRule *> rules2(1);
  rules2[0] = featurePointsRule1; //select all points of the feature
  NXObject *null (NULL);
  Point3d helpPoint1(0.0, 0.0, 0.0);
  holePackageBuilder1->HolePosition()->AddToSection(rules2,
      null,null,null,helpPoint1,Section::ModeCreate, false);
}
else //if points are input from a vector
{
  Xform *nullXform(NULL);
  Point *point2;
  for(int i=0;i<points.size();i++){
    point2 = workPart->Points()->CreatePoint(points[i], nullXform,
      SmartObject::UpdateOptionWithinModeling);
    holePackageBuilder1->HolePosition()->AddSmartPoint(point2,
      0.00095);
  }
}
```

WaveLinkPoint

The WaveLinkPoint routine is called in a loop to create the associative copies of the fastener location points in the mating component. It reads in one point, the target component, and the collection of wave linked points. Each time it is called it adds the linked point to the existing collection. Here is the code for the routine.

```
void WaveLinkPoint(NXOpen::Point *point1,
                   Assemblies::Component *comp1,
                   std::vector<Point *> &WavedPoints)
{
  //declare object tags
  tag_t  point,link_point, feat, comp, xform, target_part;
  tag_t  target_object = NULL_TAG,
  //get tags from input objects
  point = point1->GetTag();
  comp = comp1->GetTag();

  if ((point != NULL_TAG)&& (comp != NULL_TAG)) //if inputs are valid
  {
    target_part  = UF_ASSEM_ask_prototype_of_occ(comp);
    ensure_part_fully_loaded(target_part);
    target_object = UF_OBJ_cycle_all( target_part, target_object);

    if (UF_ASSEM_is_occurrence(point)){
      UF_SO_create_xform_assy_ctxt(target_part,
          UF_ASSEM_ask_part_occurrence(point), comp, &xform);
      point = UF_ASSEM_ask_prototype_of_occ( point );
    }
    else{
       UF_SO_create_xform_assy_ctxt(target_part, NULL_TAG,comp,&xform);
    }

    UF_WAVE_create_linked_pt_point(point,NULL_TAG,target_object,&feat);
    UF_WAVE_ask_linked_feature_geom(feat, &link_point);
    //convert the point tag back to a C++ object
    TaggedObject *pnt = NXOpen::NXObjectManager::Get (link_point);
    Point *point1(dynamic_cast<Point *>(pnt));
    //add the point object to the collection
    WavedPoints.push_back(point1);

  }
}
```

ProjectPoints

The ProjectPoints routine takes in a vector of points and projects them to a surface selected by the user. It then returns the projection feature so that it can be passed into the CreateSimpleHole routine. The portion of code that adds the points to the feature section is shown below

```
Xform *nullXform(NULL);
Point *point1;
for(int i =0;i<pnts.size();i++) //pnts = input vector
{
 point1 = workPart->Points()->CreatePoint(pnts[i], nullXform,
      SmartObject::UpdateOptionWithinModeling);
 projectCurveBuilder1->SectionToProject()->AddSmartPoint(point1,0.001);
}
```

Summary

After Operation 3b is complete, the fasteners holding the frame to the interstage and the fasteners holding the door to the frame are completely defined and linked associatively guaranteeing that the components mate properly. Figures Figure 4-24 through Figure 4-27 show the same components from Figures 4-20 through 4-23 with the fastener details added.

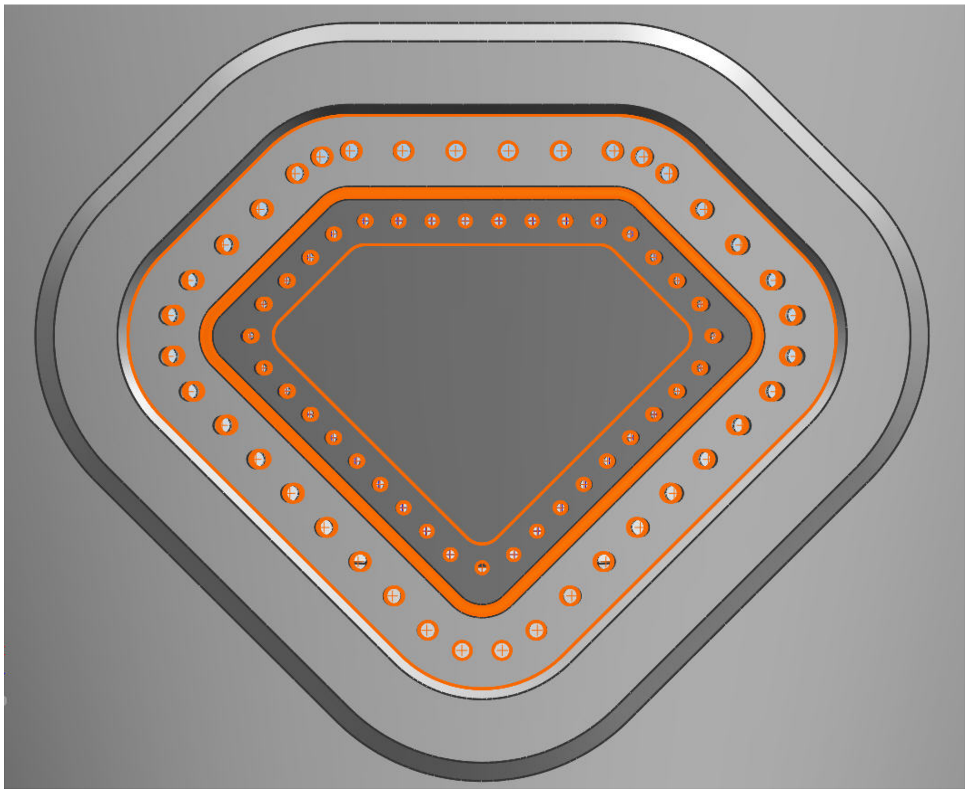


Figure 4-24: Top view of fastener details for a stiffened, load-bearing configuration.

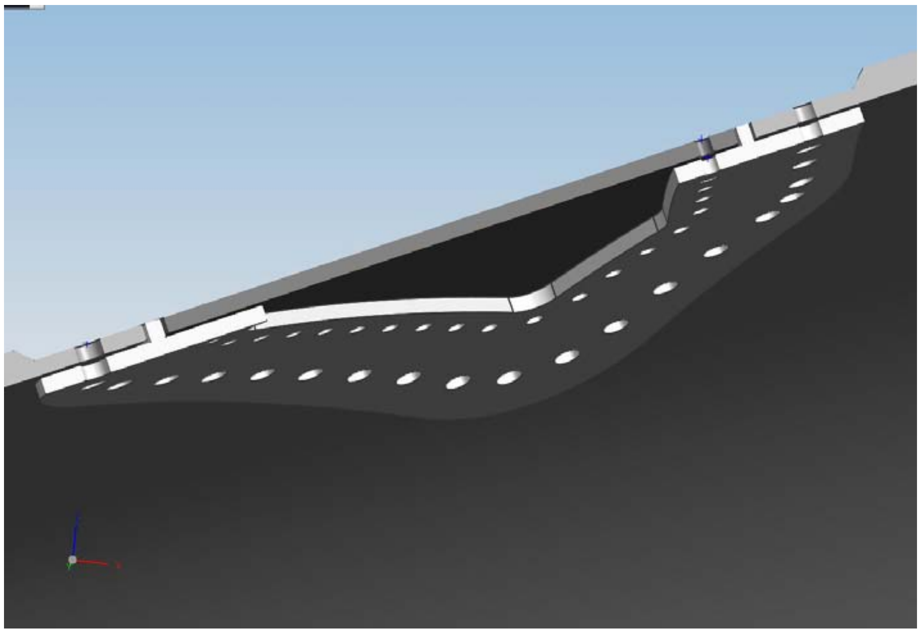


Figure 4-25: Section view of fastener details for a stiffened, load-bearing configuration

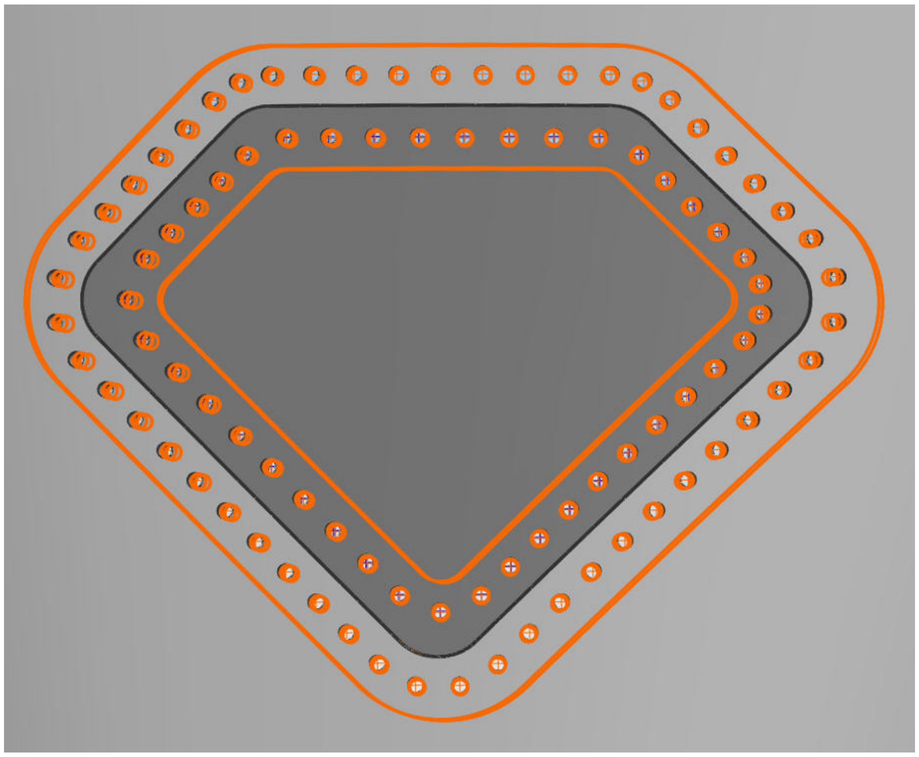


Figure 4-26: Top view of fastener details for an un-stiffened, non load-bearing configuration.

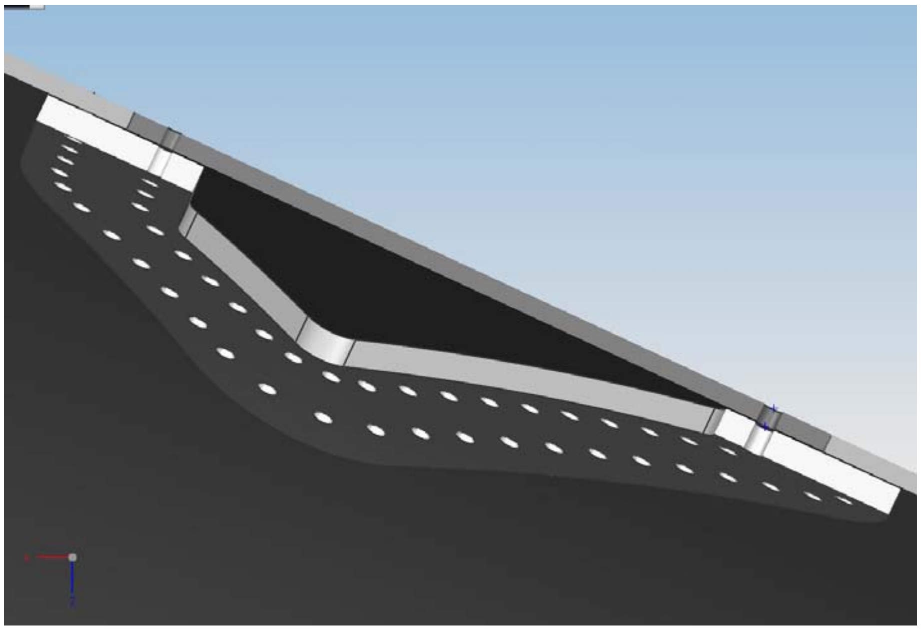


Figure 4-27: Section view of fastener details for an un-stiffened, non load-bearing configuration

5 Results

As stated in Chapter 1, the objectives of this thesis are:

- Create a framework of intelligent, high-level, operations that can be used to quickly design a wide range of rocket interstage components and assemblies.
- Show that these features/operations decrease the design time without impeding innovation.

To determine whether these objectives were met, and to what extent they were or were not successful, three elements of the objectives will be evaluated:

1. the range of designs supported by the framework,
2. the time savings observed, and
3. The proportion of the design left open to the engineer.

5.1 Range of supported designs

As a theoretical framework, the methods developed in this thesis will work for any interstage assembly design. In practice, there are certain limitations. The implementation described in Chapter 4 is capable of supporting any interstage design, subject to these limitations:

Interface Manager Limitations

- The interfaces must be either cylindrical or conical bolted flanges
- The fastener pattern must be either a single row or a double offset row pattern
- The distances from the fastener centerlines to the flange edges and between the fastener rows are predetermined but may be changed later.
- The axial position of any joint must be at least nine diameters from the origin.
- The minimum radius for any interface is three inches.
- An assembly component must contain either one or two interfaces.

Component Cross Sections Limitations

- All cross sections must contain closed loops and must not self-intersect.
- The thickness of any bolt flange must be less than five inches.
- All cross sections must extend at least the entire length of the control structure’s flange.

Detail Features Limitations

- The chamfer and fillet dimensions cannot be pre-specified. They may only be changed after running the application.

Insert Door Limitations

- The cutout sketch must be on a plane that is tangent to the interstage surface at the center of the cutout.
- The door cutout sketch must be a closed loop with no convex regions and no self-intersections
- The cutout region’s minimum width must be at least ten times the inner fastener diameter
- There space between the cutout sketch and the interstage flange on both sides of the cutout must be at least five outer diameters plus the stiffening length
- The stiffening cross section around the cutout is predetermined
- There are only two door frame cross sections to choose from

Door Fasteners Limitations

- The offset distances from the flange edges and the spacing between fasteners is predetermined
- There is only one row of fasteners per flange

To illustrate the spectrum of designs that are still possible, consider the total number of possible combinations of interfaces for one component (N_c). Let N_t be the total number of available interface types and N_i be the maximum number of interfaces associated with one component. N_c can then be calculated by

$$N_c = \sum_{j=0}^{N_i - 1} N_t^{\,N_i - j} \,. \qquad 5\text{-}1$$

which accounts for components with any number of joints from 1 to N_i.

For the current implementation of the `InterfaceManager` framework, $N_t = 8$ and $N_i = 2$. Therefore, for this implementation of the framework, $N_c = 8^2 + 8 = 72$. This means that there are 72 distinct joint combinations that can be used to create each component. Furthermore, the assembly can include any number of components and the variations of cross sections for each component is unlimited.

The framework can be expanded to include practically any type of interface found on rocket interstages, and to allow more than two interfaces per component. Thus the framework has the potential to cover the entire range of designs for interstage configurations. The method used to build this framework is also general enough to work for other product types. With some additional development, extruded cross sections and access doors on any input surfaces would also be possible.

5.2 Time savings

To determine the value of the proposed methods, three test subjects were asked to perform a set of modeling tasks using both the traditional approach and the approach implemented in this thesis. Each task correlated with one of the method's main Operations. For each test subject, the number of key-strokes and mouse clicks, and the completion time was recorded for each method. The results from each task were compiled to estimate the time and effort required to model an entire interstage assembly.

The test subjects were graduate engineering students with two to three years' experience using NX. Therefore the test subjects' completion times are most likely longer compared to those of more experienced engineers.

5.2.1 Task 1: Interface manager

In task 1, the test subject must define the control structure for one cylindrical interface and one conical interface including the hole location sketches. He must then add a new component to the assembly, create the linked geometry features, and create a fully constrained sketch of the part cross section. Table 5-1 lists the results of the three test subjects for both methods and the comparison between the two methods. There was a testing error so some data for test subject 3 is unavailable.

Table 5-1: Task 1 completion statistics

Test Subject	**Traditional Method**			**Proposed Method**			**Percent Difference**		
	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time
1	864	717	45.27	84	99	5.48	90.60%	86.19%	87.89%
2	1236	947	61.78	89	95	4.67	92.80%	89.97%	92.45%
3	error	error	41.73	54	148	6.87	—	—	83.75%
Average	**1065**	**832**	**49.59**	**75.67**	**114**	**5.64**	**91.7%**	**88.1%**	**88.0%**

5.2.2 Task 2: Detail features

In task 2, the subject must make a revolve feature from the cross section, add chamfer and blend features, and create the hole extrudes and patterns. Results from the three test subjects are listed below in Table 5-2.

Table 5-2: Task 2 completion statistics

Test Subject	Traditional Method			Proposed Method			Percent Difference		
	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time
1	220	201	14.28	0	2	0.12	100%	99.00%	99.18%
2	72	178	10.48	0	3	0.10	100%	98.31%	99.05%
3	31	199	9.68	0	2	0.10	100%	98.99%	98.97%
Average	107.6	192.7	11.48	0	2.3	0.11	100%	98.77%	99.07%

5.2.3 Task 3: Insert door

In task 3, an interstage component is provided as well as a sketch of the door cutout. The subject is asked to make the cutout in the interstage, and create the door frame and door component bodies. The simplest door configuration was selected, so the results listed below in Table 5-3 will be conservative.

Table 5-3: Task 3 completion statistics

Test Subject	Traditional Method			Proposed Method			Percent Difference		
	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time
1	205	635	28.80	0	14	1.37	100%	97.80%	95.25%
2	208	900	41.22	10	16	1.52	95.19%	98.22%	96.32%
3	404	641	46.50	16	19	2.10	96.04%	97.04%	95.48%
Average	272.3	725.3	38.8	8.7	16.3	1.7	97.08%	97.68%	95.69%

5.2.4 Task 4: Door fasteners

Finally, the test subject must add details for fasteners. For the traditional method, the user is only required to create five holes on one side of the interstage cutout and the mating holes for the door frame. The thesis method still creates the complete set of holes for all three components, so the data listed in Table 5-4 represents the results after having been multiplied by eight to compensate for this difference.

Table 5-4: Task 4 completion statistics

Test Subject	Traditional Method			Proposed Method			Percent Difference		
	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time (min.)	Key-strokes	Mouse Clicks	Time
1	456	784	42.4	0	13	0.87	100%	98.34%	97.96%
2	312	736	38.53	0	14	1.17	100%	98.10%	96.97%
3	240	672	37.07	0	15	1.30	100%	97.77%	96.49%
Average	336	730.7	39.3	0	14	1.11	100%	98.07%	97.14%

5.2.5 Entire assembly

The results from the three test subjects can be used to extrapolate an estimate for the effort required to create the parametric models for an entire interstage assembly. The layout of a fictitious interstage assembly is depicted in Figure 5-1. It includes four interfaces, five components, and will have two access doors. Table 5-5 lists the estimated average results for this configuration based on these assumptions:

1. Creating the four interfaces will take twice as much effort as measured in Task 1.
2. Inserting the detail features on all five components will require five times the effort measured in Task 2.
3. Inserting the doors and their fasteners will require twice the effort measured in Tasks 3 and 4.

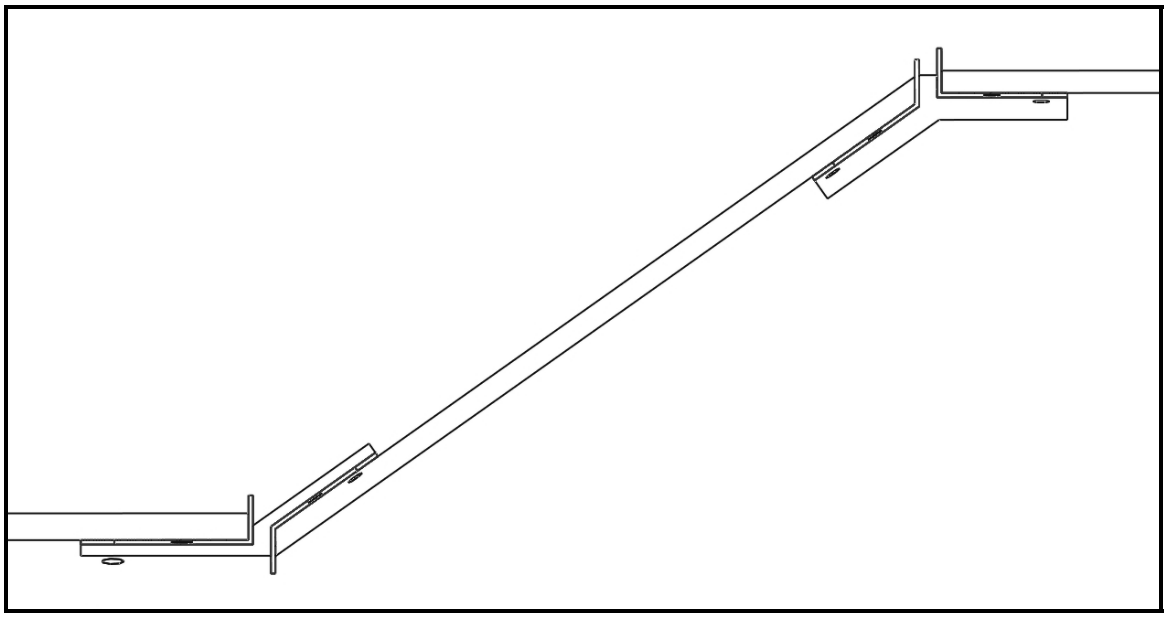


Figure 5-1 Interstage assembly configuration

Table 5-5: Estimated results for entire assembly

	Key-strokes	**Mouse Clicks**	**Time (min.)**
Traditional Method	3885.0	5539.3	313.0
Proposed Method	168.7	300.3	17.4
Percent Difference	95.66%	94.58%	94.45%

5.3 Openness of design

There are several major aspects of the application that allow flexibility in the design. The `InterfaceManager` is the primary one. It allows the engineer to define any number of component interfaces, and provides a substantial number of interface types to choose from as mentioned in section 5.1. It also lets him/her create an assembly with any number of components. Another key to providing openness is the fact that the designer creates the component cross section sketches. This permits virtually unlimited variations in the design of the part bodies. This is especially valuable for the actual interstage component since many different types of cross sections are used.

The `InsertDoor` operation also gives the designer a large amount of freedom. Since the cutout region is also sketched by the engineer, the shape of the door is almost entirely up to him or her. The only restrictions are that the cutout sketch be a closed loop with no convex portions. The disadvantage of the current implementation of the InsertDoor operation is that the topology of the interstage stiffening cross section is hard-coded. Additional development should focus on allowing a more flexible definition method for the stiffening cross section.

Overall, the fact that the models are entirely parametric means that any of the dimensions and expressions can be changed to suit the specific needs of the designer. Therefore, even the portions of the geometric design that are hard coded into the application, such as the flange length proportions or the chamfer and fillet sizes, can be modified after running the application.

To quantify the level of openness provided by the application, each of the primary decisions which must be made by the designer has been assigned a score from 0 to 3 with the following significances.

0. The designer cannot make changes to the decision
1. The designer can choose from a finite set of options
2. The designer can modify the parameters of the decision
3. The designer can change anything about the decision within normal design limits

Table 5-6 lists descriptions of each of the primary decisions and its openness score.

Table 5-6: Openness scores for primary design decisions

Decision	Score
Assembly Layout	
number of components	3
topology of each part's cross section	3
Dimensions of each part's cross section	3
Chamfers and fillets	2
Joining method for each part-to-part interface	
Interface type	1
Interface position	3
fastener size	2
fastener pattern	1
dimensions of the joint	2
Doors	
Number of doors	3
Door shape and size	3
Door position	3
Stiffening cross section	1
Door frame cross section	1
Joining methods for door components	
Fastener size	2
Fastener pattern	1
Dimensions of the joint	2

Several design decisions received low openness scores. It would be possible to increase the low scores with further development and research. Several suggestions for such future work will be given in Chapter 6.

5.4 Discussion of results

The primary difficulty in creating parametric design tools is balancing the tradeoffs between speed and design freedom. The results presented in Chapter 5 have shown that the methods developed in this thesis are able to decrease the required time and effort by more than 90% while still leaving a large majority of the primary decisions open to the designer.

Although these methods have been developed specifically for designing rocket interstage assemblies, they have potential application in any assembly dominated by similarly oriented 2½ dimensional components, i.e. uniform cross sections that are either extruded in the same direction or revolved around the same axis. The door insertion and fastener methods can also be applied in other disciplines such as pressure vessel design.

A primary advantage of these high-level programmatic operations over other design automation tools, such as UDFs, is that these methods are able to operate on multiple components. Therefore, they can create the inter-part associativities and expressions that are necessary in parametric assembly modeling. In addition, they are able to generate much larger sets of geometry since UDFs cannot use their own entities as inputs to their other features e.g. A UDF would not be able to contain an offset surface feature and a feature that trims said offset surface, since the user would not be able to identify the input surface to the trim feature.

Other important advantages of the methods presented here are that they drastically reduce user error and can be executed by novice engineers, or even technicians. During testing, many of the manual operations had to be repeated or corrected because the wrong input geometry was selected, or because input values were wrong. Programmatic methods do not have these problems. There were still some user errors while testing the programmatic methods, but they were usually due to unclear instructions and were much less frequent.

One disadvantage of the author's methods, is that they do not currently provide special functionality for updating the geometry after the model is changed. For most of the geometry, this is handled automatically because standard NX features are used. However for features that depend on custom calculations, such as the smart point features that are inserted based on the curve lengths, the standard NX update algorithms would not suffice. If the curve lengths change, the number of smart points would not. Custom updating routines would also be able to ensure that offset directions do not flip, and that the trimming regions remain correct.

The author's methods also lack special deleting functionality. The operations can be undone immediately after execution, but if a user wanted to remove a door later in the design process, for example, he or she would have to delete all of the components, features, and expressions manually. This would be tedious and error-prone.

6 Conclusions

The objectives of this thesis were to show that high-level, product type-specific operations can accelerate the design of a wide range of rocket interstage components and assemblies and that these operations will decrease the design time without impeding innovation.

In Chapter 3, a method was developed to define the assembly layout using a framework of C++ classes and user interfaces called the `InterfaceManager`. While this theoretical framework was capable of supporting any assembly layout, the framework that was implemented in Chapter 4 was limited to eight types of interfaces. Chapter 5 demonstrated that the `InterfaceManager` still supported a very large number of interface combinations even with these limitations and was able to create the interfaces around 90% faster than by using the traditional method. From these results we can conclude that product type-specific operations can greatly reduce modeling time of assembly layouts and can be flexible enough to support wide spectrums of designs.

Chapter 3 also discussed methods for creating the detailed features on each part in the assembly including the revolve features, chamfers, fillets, and hole patterns. These methods, as developed in Chapter 4, resulted in more than a 98% reduction of modeling time and effort. Since the inputs for these detail features were defined by the `InterfaceManager`, no effort was required of the user to detail the parts. These results prove that CAD design can be streamlined extensively using high-level operations.

The methods from Chapter 3 that add access doors with fasteners to the interstage were also successfully implemented in Chapter 4. After analyzing their performance, Chapter 5 proved that these methods also resulted in excellent time savings while leaving most of the major design decisions open to innovation.

To summarize, this thesis has shown that CAD modeling can be extremely streamlined through the use of high-level, product type-specific operations. It has shown that such high-level operations can work for a wide range of components and assemblies and can be created in a way that leaves the majority of the primary design decisions open to the user. The methods developed in this thesis have also reduced modeling time and effort by at least about 90%. Recommendations will now be given for researchers interested in continuing similar work.

6.1 Recommendations

The results presented in Chapter 5 show that these methods have excellent potential but there are still many improvements that can be made to increase the scope of supported designs and the functionality of the operations. Implementing the following recommendations would increase the openness scores as discussed in section 5.3.

The largest limitation of the current method is that it currently only applies to revolved parts. The `InterfaceManager` framework should be extended for use with extruded parts which also have 2 ½ dimensions. Extending the methods to fully three dimensional products would require much more research but would be very valuable.

The `InterfaceManager` should also be extended to include additional interface types. For interstage assemblies alone, there are several more interfaces that are commonly used, such as manacle joints and weldments.

Along with these improvements to the `InterfaceManager`, there are several areas of needed improvement to the `InsertDoor operations.` Supporting extruded parts would require the `InsertDoor` operation to work on any input surface. This improvement would be trivial. Research should also be focused on developing a more general method for defining the stiffening cross section around the door cutout. If the operation required the user to create a sketch of the desired cross section, it could investigate the sketch geometry and determine which offset features would be needed as well as their distances. It would also be able to determine the correct trimming features.

Most of the operations could be improved if further researchers developed special updating routines and deletion routines. The `DoorFasteners` operation could update the fastener patterns when the door size changes. The `InsertDoor` operation could ensure correct offset directions and trimming regions during update cycles. A `DeleteDoor` operation would be very useful during the design cycle so that users would not have to manually delete every part, feature and expression.

7 References

Lendermann, C. (2005). Associative parametric CAE methods in the aircraft pre-deisgn. *Aerospace Science and Technology*. Vol. 9, No. 7. pp 641-651.

Hoffman, C.M, Joan-Arinyo, R. (1998). On User-defined Features. *Computer Aided Design*, Vol. 30, No. 5. pp 321-352.

Elliott, J. (2004). *An automated approach to feature-based design for reusable parameter-rich surface models.* M. S. Thesis, Brigham Young University

Bidarra, R., Idri, A., Noort, A., Bronsvoort, WF. (1998). Declarative user-defined feature classes. *CD-ROM Proceedings of the 1998 ASME Design Engineering Technical Conferences*, 13–16 September, Atlanta, GA, USA, New York: ASME.

Shah, J.J., Ali, A., Rogers, M.T. (1994). Investigation of declarative feature modeling. *Proceedings of the ASME 1994 Computers in Engineering Conference*, ASME, NewYork, Vol. 1, pp. 1-11.

Tang, M., Wen, Y., Mi, X.,Dong, J.; (2001). Parametric modeling with user-defined features. *Computer Supported Cooperative Work in Design, The Sixth International Conference on*, 12-14 July, pp 207 – 211.

Lamarche, B., Rivest, L. (2007). Dynamic Product Modeling with Inter-Features Associations: Comparing Customization and Automation. *Computer-Aided Design & Applications.* Vol. 4, No. 6. pp 877-886.

Jankowski, G. (2005). Solid Thinking: Using Functional Features to Build Plastic Parts. *Cadalyst* Nov. 15, 2005. Retrieved on 12/10/07 from: http://manufacturing.cadalyst.com/manufacturing/article/articleDetail.jsp?id=197017

Huh, Y.and Kim, S. (1991). A knowledge-based CAD system for concurrent product design in injection moulding. *International Journal of Computer Integrated Manufacturing*.Vol. 4, No. 4. pp. 209 – 218.

Ong, S.K., Prombanpong, S., Lee, K.S. (1995). An object-oriented approach to computer-aided design of a plastic injection mould. *Journal of Intelligent Manufacturing*. Vol. 6. pp 1-10.

Delap, D., Hogge, J., Jensen, C. (2006). CAD-centric creation and optimization of a gas turbine flowpath module with multiple parameterizations. *Computer-Aided Design & Applications*. Vol. 3, Nos. 1-4, pp 175-184.

Danjou, S., Lupa, N., Koehler, P. (2008). Approach for Automated Product Modeling Using Knowledge-Based Design Features. *Computer-Aided Design & Applications*. Vol. 5, No. 5, 2008, pp 622-629.

Emch, F. (2002). Impact of System-Level Engineering Approaches on the Airframe Development Cycle Via Integration of KBE with CAD Modeling and PDM. *RTO AVI Symposium.* April 2002.

Mosca, F., Di Martino, C., Aleixos, N. (2001). Complex CAD project management by the means of designing criteria control tools. Deployment of a vehicle gearbox archetype with the aid of WAVE by UNIGRAPHICS. *XII ADM International Conference*. September 2001.

Ma Y.-S. et. al. (2007). Associative assembly design features: concept, implementation and application. *The International Journal, advanced manufacturing technology.* Vol. 32, No. 5, 2007, pp 434-444.

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