2017-04-01

Hybrid State-Transactional Database for Product Lifecycle Management Features in Multi-Engineer Synchronous Heterogeneous Computer-Aided Design

Devin James Shumway
Brigham Young University

Hybrid State-Transactional Database for Product Lifecycle Management Features in Multi-Engineer Synchronous Heterogeneous Computer-Aided Design

Devin James Shumway

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

ABSTRACT

Hybrid State-Transactional Database for Product Lifecycle Management Features in Multi-Engineer Synchronous Heterogeneous Computer-Aided Design

Devin James Shumway
Department of Mechanical Engineering, BYU
Master of Science

There are many different programs that can perform Computer Aided Design (CAD). In order for these programs to share data, file translations need to occur. These translations have typically been done by IGES and STEP files. With the work done at the BYU CAD Lab to create a multi-engineer synchronous heterogeneous CAD environment, these translation processes have become synchronous by using a server and a database to manage the data. However, this system stores part data in a database. The data in the database cannot be used in traditional Product Lifecycle Management systems. In order to remedy this, a new database was developed that enables every edit made in a CAD part across multiple CAD systems to be stored as well as worked on simultaneously. This allows users to access every action performed in a part. Branching was introduced to the database which allows users to work on multiple configurations of a part simultaneously and reduces file save sizes for different configurations by 98.6% compared to those created by traditional CAD systems.

Keywords: Interoperability, Hybrid State Transactional Database, Database, Revision History, Configuration Management, Feature Level History, CAD Translation, Heterogeneous CAD, CAD Features, Neutral Format, Design History, Multi-User CAD, Collaboration, Neutral Parametric Canonical Form, Design Intent

ACKNOWLEDGMENTS

I would like to express thanks to my family for their support and motivation. I would like to thank my advisers Dr. John Salmon, Dr. Steve Gorrell and Dr. Alan Parkinson for their support and mentorship in completing my thesis. I give thanks to the Interop team in the BYU CAD Lab: Eric Bowman, Josh Coburn, Dan Staves, Robert Freeman, Scott Christensen and Logan Hill for making this research possible. I express thanks as well to those that helped edit papers including, Jon Sadler, Ariana Pedersen, Jacob Tovar, Jordan Oldroyd, Spencer Brunell, Ian Freeman, Brandon Davis, and Alex La. As well as the rest of the BYU CAD Lab for their research and contributions to the field of collaboration in engineering that provided a foundation from which to develop this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

GUI	Graphical User Interface
GUID	Globally Unique Identifier
NPCF	Neutral Parametric Canonical Form
NPDB	Neutral Parametric Database
2D	Two-Dimensional
3D	Three-Dimensional
API	Application Programming Interface
IGES	Initial Graphics Exchange Specification
STEP	Standard for the Exchange of Product Model Data
HSTDB	Hybrid State Transactional Database

CHAPTER 1. INTRODUCTION

1.1 Problem Statement

In 1999 a study performed by the National Institute for Standards and Technology made an estimate that within the automotive supply chain, imperfect interoperability created costs between \$1.02 billion and \$1.05 billion dollars per year. They estimate that 86% of these costs originate from repairing unusable data [2]. This repairing of data is largely made up of the changes needed to be made to CAD files transferred between the designers and manufacturing facilities. Through a CAD Lab internal study, with different aerospace partners, we have seen that these problems also exist in the aerospace industry. I also believe that these trends exist in many other industries not explored in this research.

Due to the number of CAD applications that are used in a manufacturing supply chain, companies are forced to interface with differing CAD file formats other than the CAD system that they use in-house. In the past, drafting has been the main method of communication between designers and manufacturers, and rigorous drafting standards have been put in place; however, even with these standards in place close communication between designers and manufacturers is necessary [3]. Currently, conversion between CAD file formats is done using translation practices, namely the International Graphics Exchange Standard (IGES) or the Standard for the Exchange of Product Model Data (STEP). These systems have their benefits but each has limitations as well. IGES represents only geometric data contained within the model, and while geometric data translation is necessary, more data is typically desired. IGES only translates Boundary Representation (BREP) data and CAD features such as associativity are lost. With STEP, current research is ongoing with Solid Model Construction History (SMCH) which stores BREP data and construction history to not only keep the geometry but also to preserve design intent. This leads to large file sizes [4], in comparison with standard part files, due to storing additional information. SMCH also results in part loading failure rates as great as 50% [5]. These solutions, IGES and STEP, are incomplete and are at best a short-term fix to a larger problem which will continue to grow as products and manufacturing distribution chains get more complex.

In order to allow a better, more seamless transition between CAD systems, a CAD plugin named “Interop” was developed at BYU [6]. This plugin works with existing CAD systems to help translate data in real time so that users can synchronously be working on the same part in multiple CAD systems. As this level of interoperability becomes a possibility with increasing speed of internet connections and computer processors, a need arises for a neutral format to store the CAD file data. In order to address this the Neutral Parametric Canonical Form (NPCF) was created. This allows users to open a CAD package of their choice, and if they have the correct plugin to translate data from the neutral format to their CAD package. Then they can open the part files and associated features locally. The process used to create the NPCF follows the recommendation outlined by The Neutral Modeling Command method [7]. The client uses the application programming interface (API) of each CAD system to create a collaborative heterogeneous CAD solution by storing the data of CAD features in a database named the Neutral Parametric Database (NPDB). This database contains tables which store the feature data in its associated NPCF. Each database table for a feature contains a unique identifier, a state of the feature (creation, modification, deletion), and the parameters of the feature.

However, there are limitations with the current implementation of the NPCF and NPDB. Due to the nature of Product Lifecycle Management (PLM) systems, which store and manage CAD information as well as any other information needed in the design process, there is no way to store the NPDB and its associated part and feature data. A user could store the data but would need to perform a database operation to generate the scripts for the database and store those scripts in PLM. This adds an additional step to restore multi-user functionality. On part open, a user would have to download the database script from the PLM system and generate a database from it before the user could start working with other users.

Another way to interact with PLM is to download the NPDB files and save off the binary part files typical to that CAD system. Finally, they could store the binary part files in the PLM system. This method also has a drawback as it loses the synchronous interoperability provided by the NPDB and associated plugins.

The next step in extending the NPDB is to incorporate PLM features into the NPDB to allow for greater control at the part level for the data needed throughout the production process. Since the PLM system cannot be interfaced directly, some of the features it provides would be beneficial to have directly in the CAD environment. For the purposes of this thesis, PLM will specifically refer to the systems that attempt to manage CAD system data from a part management perspective.

The selected features that are being integrated into the NPDB are revision history and configuration management. These features allow for greater control of parts and part structures, a logical place to incorporate PLM features. Revision history also allows for previous versions of the part to be saved which will allow for greater design transparency and provides the groundwork for many additional PLM features, one of which is configuration management.

1.2 Summary of Research Objectives

The goal of this research is to provide a Hybrid State Transactional Database (HSTDB) that allows not only for part geometry and feature data as provided by the NPCF, but to also allow Revision History and Configuration management. The goal can be seen in Table 1.1 showing the HSTDB with all data being stored and accessed by all users.

Table 1.1: Comparison with proposed HSTDB and current methods to show the goals of the research.

Translation Comparison	IGES	STEP	NPCF	HSTDB
Part Geometry	✓	✓	✓	✓
Feature Data	X	X	✓	✓
Revision History	X	X	X	✓
Configuration Management	X	X	X	✓

The proposed structure for the HSTDB can be seen in Figure 1.1. New tables that will be added to the current NPDB can be seen in yellow for configuration management tables and green for revision history tables.

The research will be completed and validated when

1. The database has been expanded to store all revisions of a part

2. A user can interact with the new database information to view previous part states

3. A user is able to make and load different configurations of a part through branching

4. The multi-user and multi-CAD package aspects of Interop have been maintained

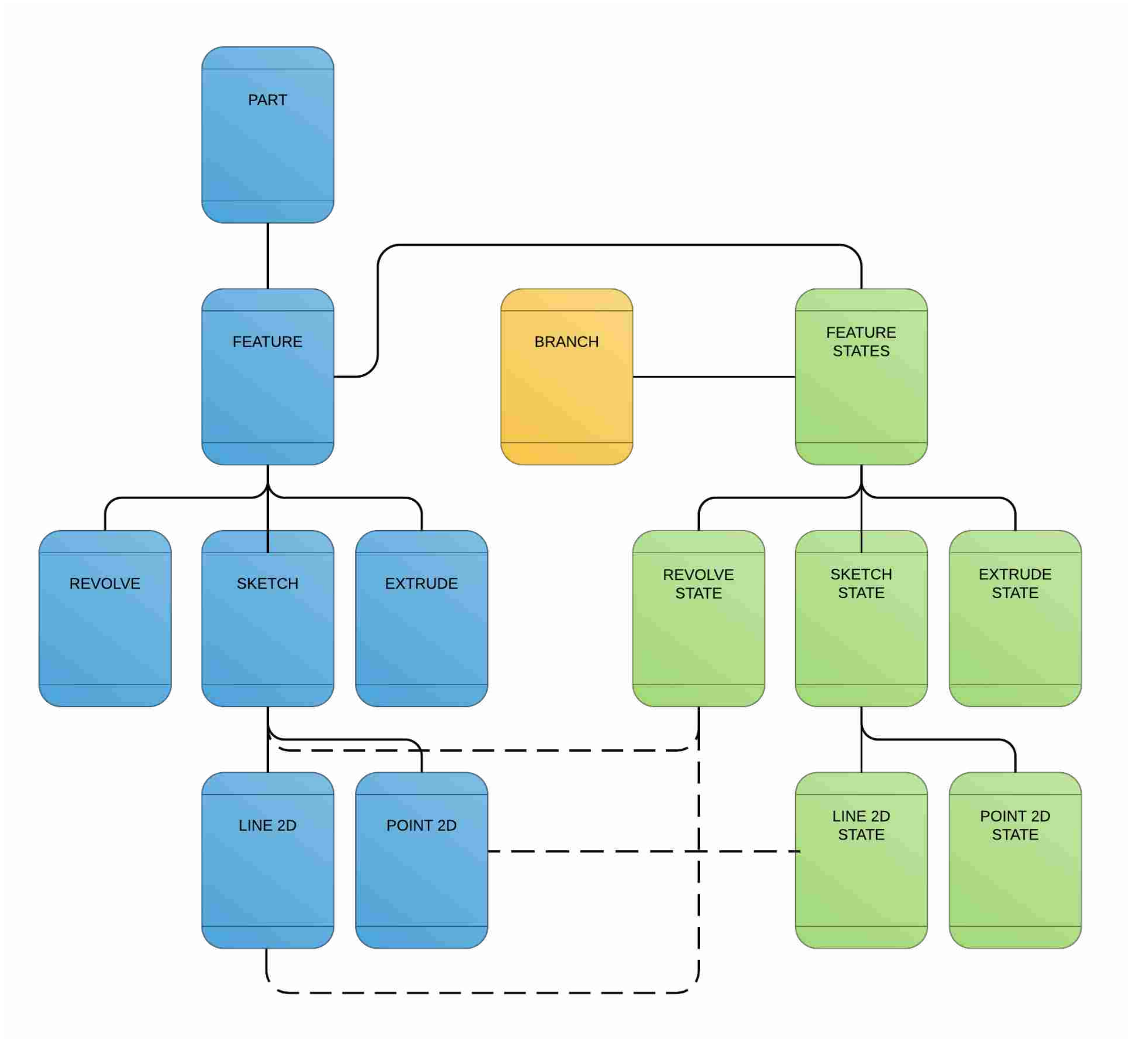


Figure 1.1: Proposed database structure with current NPDB in blue, new revision history tables in green and new configuration management tables in yellow.

Objective 1 will be discussed in Chapter 3, where the changes made to the NPDB in order to create the HSTDB will be explained. Chapter 4 will discuss Objective 2 and how a user can view the new data stored in the database. Chapter 4 will also discuss how Objective 4 was maintained with revision history. Chapter 5 will discuss Objective 3 as well as Objective 4 and how they were implemented on the client side for configuration management. Background for all of the topics that make up this research will be discussed in Chapter 2. Finally results from this research will be discussed in Chapter 6.

CHAPTER 2. BACKGROUND

This research is based off of principles taken from five different subjects.

1. Single-User CAD
2. Multi-User Synchronous CAD - which involves multiple users interacting with the CAD file in real time.
3. Multi-User Synchronous Heterogeneous CAD - Which is similar to multi-user synchronous CAD but involves the use of multiple CAD systems interacting with the same CAD file in real time.
4. Revision History - Which allows the user to see all changes made to the part.
5. Configuration Management - Which is the storage of different configurations of an assembly with different parts being referenced by different assemblies.

Each of these subjects contribute to the understanding that went into the creation of the Hybrid State Transactional Database (HSTDB). Figure 2.1 shows the timeline of when the different subjects that comprise this research were introduced. This timeline includes PLM for revision history and configuration management and solid model construction history as a stepping point for Interop.

For the purposes of this thesis a part will refer to a CAD part that is one single object that when manufactured would represent one single object. An example of a part can be seen in Figure 2.2 where a part can be seen in the tree on the left being represented by a name such as “Main Upright”. An assembly will refer to the entire collection of parts that make up a product, Figure 2.2 shows a full assembly of a pipsqueak engine. A sub-assembly will refer to a collection of parts that are a stand alone design but exist within a larger assembly as well. Figure 2.2 does not contain any sub-assemblies; however the highlighted parts could be made into a sub-assembly that would represent the shaft of the part. A feature will refer to one CAD operation such as a sketch, extrude,

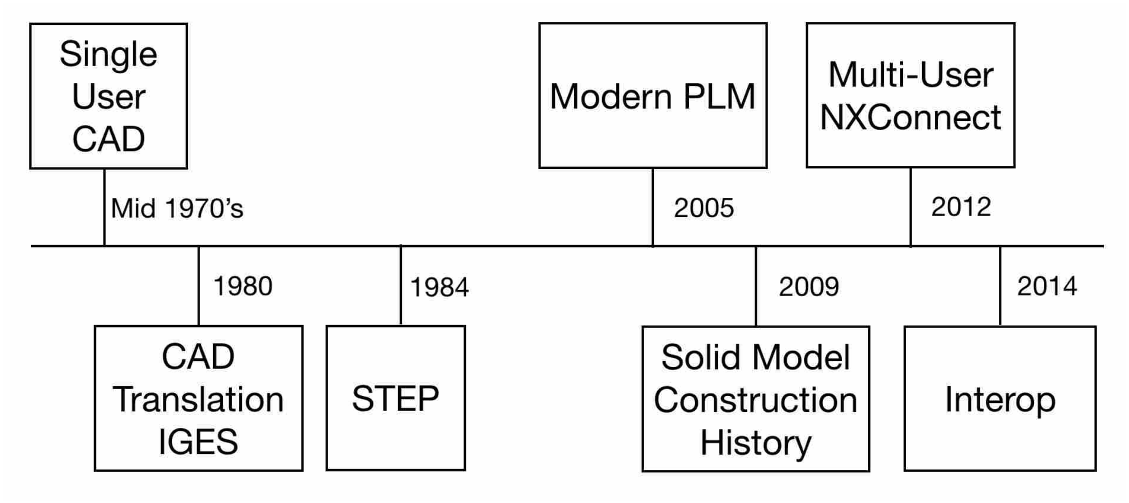


Figure 2.1: Timeline of the different applications on which this research is based.

2D point, revolve, fillet, line, etc. From Figure 2.2 the blue lines that can be seen located on the bottom of the assembly represent a sketch feature that is made up of smaller line and point features. From a hierarchical view an assembly contains sub-assemblies which contain parts which contain features.

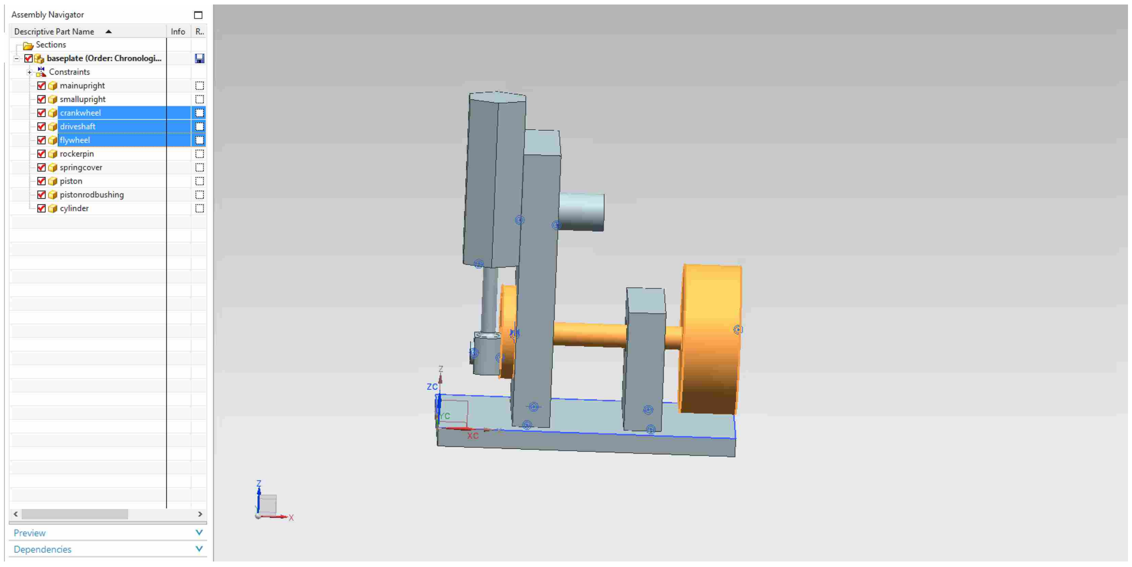


Figure 2.2: Assembly of a pipsqueak engine with multiple parts and features.

2.1 Single-User CAD

The current state of single-user CAD is a diverse collection of many single-user CAD packages where each user is allowed to open and edit a part only if that part has not been opened by another user. In order to facilitate this process, CAD packages use a PLM tool such as Siemens Teamcenter to allow users to check-out a part and then save and check-in that part after they are finished working on it [8]. This allows for transference of ownership from one user to the next depending on which user has the file checked-out. The benefit of this system is that it allows multiple users access to any given part file or assembly file. The downside of this system is that when one user is working on a part, other users cannot edit that checked-out file because they do not possess ownership at that current time. In order to allow users to view part files without ownership Siemens has created a lightweight 3D viewing file format that is created on check-in called JT files [9]. This provides a multi-user feel to the PLM system by allowing users to view the current state of the file before it was checked-out but does not allow users to view changes being made in real time.

There are many CAD systems that fall under the umbrella of single-user CAD including Siemen's NX, Dassault's CATIA, and PTC's CREO. These three systems comprise the focus of this thesis and were included in the Interop platform created in the CAD Lab at BYU as discussed in the following section.

2.2 Multi-User Synchronous CAD

In order to address the issues mentioned under the Heterogeneous CAD section, the BYU Center for e-Design developed NXConnect [10]. This system is a thin-server, thick-client system that utilizes the API of Siemen's NX CAD package to create messages for each operation performed on a client and then send the messages to each other user that has the part or assembly file open. This design allows for a synchronous CAD plugin that works with Siemen's NX so that multiple users can be active in the same part simultaneously. NXConnect performs operations on the clients machine allowing the server to focus on sending and receiving messages from the individual clients and redirecting the messages to the appropriate clients [11].

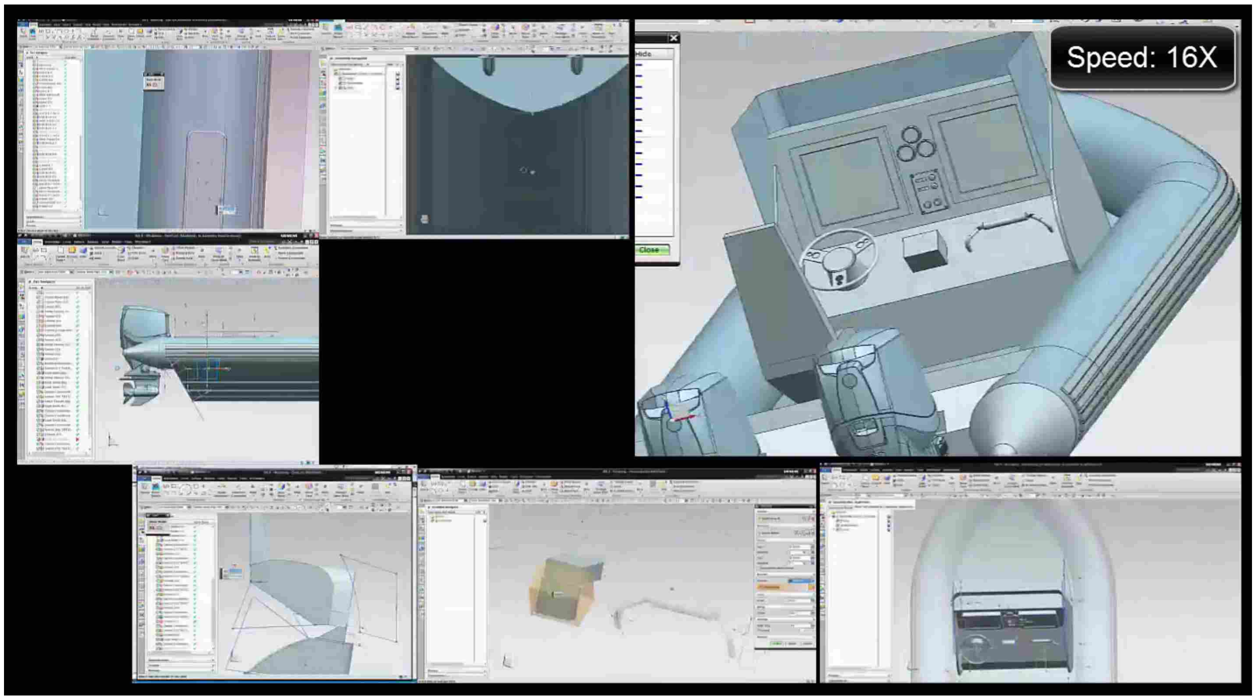


Figure 2.3: Example of six users working simultaneously in NX to create a Zodiac CZ7.

Figure 2.3 shows six users working in NXConnect on a Zodiac CZ7 watercraft. As a users create changes, those changes are added in all other users' CAD assemblies. In Figure 2.3 the users all have the same part open and they are able to see what the actions performed by other users. In order to not have inconsistencies and errors on different users' computers, no features are applied while a user has a feature dialog window open. Feature messages are also not applied at the assembly level, if the user only has a part which exists in the assembly open, but not the assembly itself to prevent part corruption. Not applying messages that are at the assembly level if the user is in the part level helps explain what is happening in the bottom left image of Figure 2.3. The user in the Figure has a part open but not the whole assembly so changes to the Zodiac CZ7 are not displayed in that view. The image in the top right of Figure 2.3 shows a client that has been opened to show the current state of the assembly. This client receives messages only and acts as a viewer in displaying the functionality of the NXConnect plugin.

2.3 Multi-User Synchronous Heterogeneous CAD

After years of research into multi-user CAD, the BYU team developed the neutral parametric canonical form (NPCF). The NPCF helps to solve some of the translational problems within current CAD systems by providing full geometric data through the current state-based format in which the data is stored [12]. This data is feature-specific and thus maintains design intent. The NPCF stores the feature data in a database namely the Neutral Parametric Database (NPDB). The NPCF also leverages the knowledge gained in creating NXConnect to provide the data in the NPCF to every CAD package. This is done by writing a plugin in the same multi-user synchronous format as described in section 2.2 for NXConnect. At this time plugins have been written for multiple CAD packages which makes it heterogeneous in nature. This led to the naming of this type of CAD as Multi-User Synchronous Heterogneous (MESH) CAD.

The current state of the NPCF stores the feature data for every feature supported by the Interop plugin in a CAD file, part, or assembly. These features are stored in the database and every client attached to the database can access the data. The NPCF relies on the client's computer to translate the data from the NPCF to create command messages that create the features using the API of the client receiving the message from the NPCF. This allows any user to not only access the data in the database but to push data to and receive from other clients. This results in a thick-client, thin-server architecture, similar to that of NXConnect, that supports a multi-user synchronous collaborative environment where multiple users can model in the same part and assembly. Unlike NXConnect, this is done in NX, CATIA and CREO. The three CAD programs can be used simultaneously using each program's API instead of only NX's NXOpen API. The thin-server, thick-client architecture is in opposition to a thick server architecture where a server contains all of the data for the part and only sends visualization data to the clients as can be seen by the examples NetFeature [13] and CADDAC [14] which are new CAD packages that follow a thick-server architecture. This process also differs from the STEP file storage in that the current database format does not allow for the storage of the construction history, which results in a lack of full design intent being saved.

An example of MESH CAD can be seen in Figure 2.4 which is similar to Figure 2.3 in that multiple users are working on the same assembly. However, unlike Figure 2.3 multiple users can work in multiple systems at the same time with different CAD packages namely NX, CATIA, and

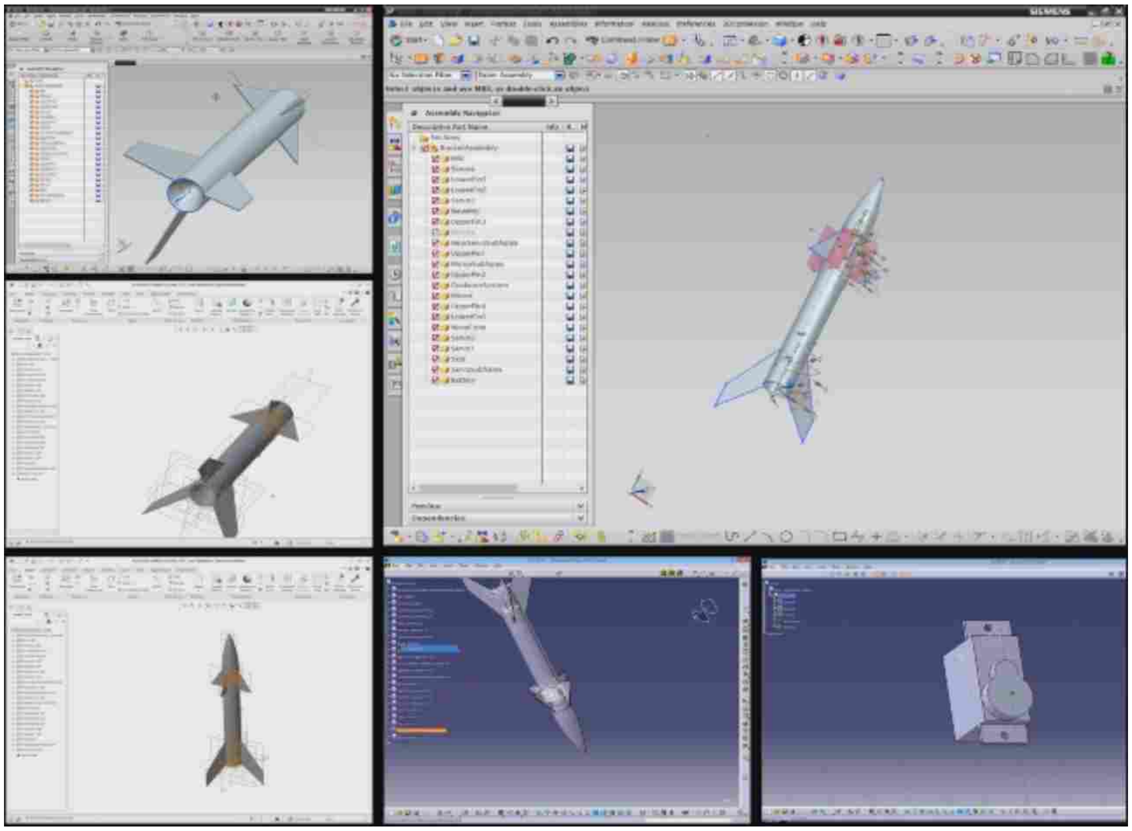


Figure 2.4: Example of six users working simultaneously in NX, CATIA, and CREO to create a model guided rocket system within MESH CAD. NX can be seen in the top left window and the large window on the right. CREO in the two left side windows under the NX window, and CATIA on the bottom with the blue background.

CREO being present in Figure 2.4 and still receive updates synchronously. In Figure 2.4, there are six users working simultaneously on a model guided rocket. Two of the users are working in NX, two in CREO, and two in CATIA. They are all working on one assembly file and as one user makes a supported feature change to any part, that change can be seen on every other screen that has the assembly open. This is accomplished by having event handles programmed into the CAD system so that when an action is performed, the program recognizes it, uses the API to grab the feature data, translates it into the NPCF, and sends that data to the server. When the server receives a message it compares the data with the format of the data associated database table, and if the data does not fit the table it is rejected by the server. If the data is not rejected, the server immediately sends the modeling command to the other users in the part.

The CATIA user in the bottom right of Figure 2.4 is working on a servo that moves one of the control flaps. To show additional functionality, this user is working directly in a part file and does not have the assembly part open. As this user makes changes to the servo, every user that has the assembly file open still receives these changes to the servo, but the user does not receive any updates to the assembly because they do not have it open. The plugins to the different CAD systems used in this example utilize the NPCF and the NPDB and comprise the program called "Interop."

2.4 Neutral Parametric Canonical Form

Interop uses the NPCF to formulate its database and allow users to work in multiple CAD systems simultaneously [6] similar to the idea postulated for neutral modeling commands by Li [15–17]. The current database format has tables for each supported feature. A simplified version of the database can be seen in Figure 2.5. The tables as can be seen in Figure 2.5 all can be linked back to a Part Table. When a user selects a part to load from the database, every feature that contains that part in its table is sent to the user. These features are then used to recreate the part on the user's local machine. When a feature is edited, the data for that feature is overwritten so that there is only one record of that feature in the database at any time. Therefore, there is no traditional interpretation of undo, redo, or history for the part. If the feature is deleted it is kept in the database but a delete flag is inserted so that the system knows the feature has been deleted and should be removed as soon as possible on the other clients.

This is similar to current CAD system files in that the database contains the information for a state and this state and associated tables are equivalent to the save file for the part. It is different in that multiple users can edit the CAD file simultaneously in multiple different CAD systems and receive the edits of all other users in the part. One benefit of this structure is quick loading, compared to the slower loading speed of other multi-user database stored systems such as NXConnect. This decrease in loading times comes from the current state of the part being stored at all times in the database so the client does not need to do the calculations for every edit to the geometry one at a time like in NXConnect. The downsides are that with the current state of Interop only the current state is saved, no previous data is stored, and only one revision of each part or feature can be saved in the database without completely remaking or copying the

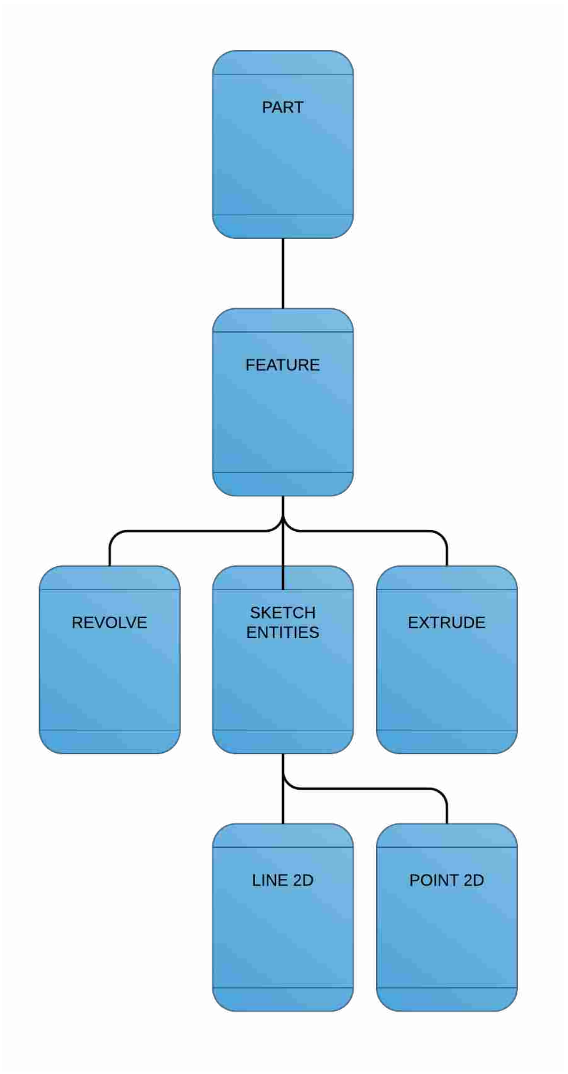


Figure 2.5: Simplified database scheme with a part table that contains the data for all of its associated features.

part. Design intent is not preserved past the current state of the part but a portion of design intent is preserved because the feature tree is maintained across CAD systems in the current implementation of Interop.

2.5 Limitations of Current System

While the NPCF helps to solve many issues surrounding CAD interoperability, it does leave the user with a deflated user interface in which they lose many of the actions that they could perform in a typical CAD system. The NPCF and its associated NPDB cannot handle the following operations in their current state; copying parts, undo, redo, revision history, branching and merging of parts and assemblies, and full configuration management. With the updates proposed for the NPDB, the groundwork for these features will be provided in the database with revision history, and configuration management implemented.

Many CAD features have not been implemented in Interop due to Interop being a research application and a proof of concept. This leaves Interop with a limited feature set, which contains sketches, extrudes and revolves. All other features found in CAD systems have not been defined in the NPCF at this time, including boundary representation (BREP) features which allow the users to create new geometry off of currently existing geometry such as creating a sketch on a face of an extrude or creating a curved edge along an existing edge of an extrude. The feature set of the NPCF will not be expanded in this study.

Most of the functions listed above are handled by PLM systems. Copying parts and revising parts are tracked using PLM systems. In current CAD PLM systems, part files are stored to represent the parts. Since the NPCF uses a database structure that stores feature data, users do not have to duplicate data in order for it to be used in multiple locations. To be used in multiple locations, the database needs to be told where to look for the data. Users can also duplicate a part for revisions without adding any more data than a new branch which contains a Globally Unique Identifier (GUID) and a name which in this case informs the database that the data will be reused.

2.6 Revision History

STEP and IGES have been created to help in the translation of data between CAD systems to create a heterogeneous environment. As seen in Table 2.1 IGES, STEP and the NPCF are able to translate the Boundary Representation (BREP) data from one system to another. But IGES and STEP fail in translating and representing feature data to another system which is the reason for the creation of the NPCF as representing feature data in real time is also what allows for MESH CAD. However, as seen in Table 2.1, there is not a single translation package that also translates revision history. The main goal behind saving revision history is to preserve design intent. Currently PLM systems store revisions by saving a binary part file of the design for each revision that is desired by the user. In order to look at previous versions of a design, one is selected and the part file is loaded for that revision. While this does provide a way to look at previous revisions of a part, and works with current PLM systems, this method does not work with the NPCF and NPDB.

Table 2.1: Comparison of translation methods and the features provided or not provided by each.

Translation Comparison	IGES	STEP	NPCF
Part Geometry	✓	✓	✓
Feature Data	X	X	✓
Revision History	X	X	X
Configuration Management	X	X	X

Due to the fact that the NPDB and NPCF are not compatible with current PLM systems, since they do not create binary part files, they cannot be integrated. In order to interact with a PLM system the NPDB data for a part would have to be saved to the binary format from one of the CAD systems accepted by the PLM system which removes the heterogeneous CAD system work-flow benefit gained from MESH CAD systems such as Interop.

2.7 Configuration Management

Configuration management is the process of making a different assembly by selecting different parts to form a different configuration and saving that to the PLM system. This allows for multiple parts to be used in different assemblies as well as using one base assembly and adding different sub-assemblies to reuse much of the work that has already been completed. A great example of that is show in Figure 2.6 which has one base chassis that is used to create a Family of Medium Tactical Vehicles (FMTV) [1]. This base chassis is used to make every single configuration of the FMTV for use in many different situations from trailers to vans. One benefit of this base chassis is that it drives costs down because effort and models can be reused such as all the parts in the chassis. Furthermore, manufacturing of that base part is done in greater numbers allowing for better economies of scale to be developed instead of having a different base for each need.

In the current state of configuration management the configurations are handled at the part level. This is how configuration management is currently handled but like revision history, since PLM deals with the binary part files from different CAD systems, this does not link into the NPDB

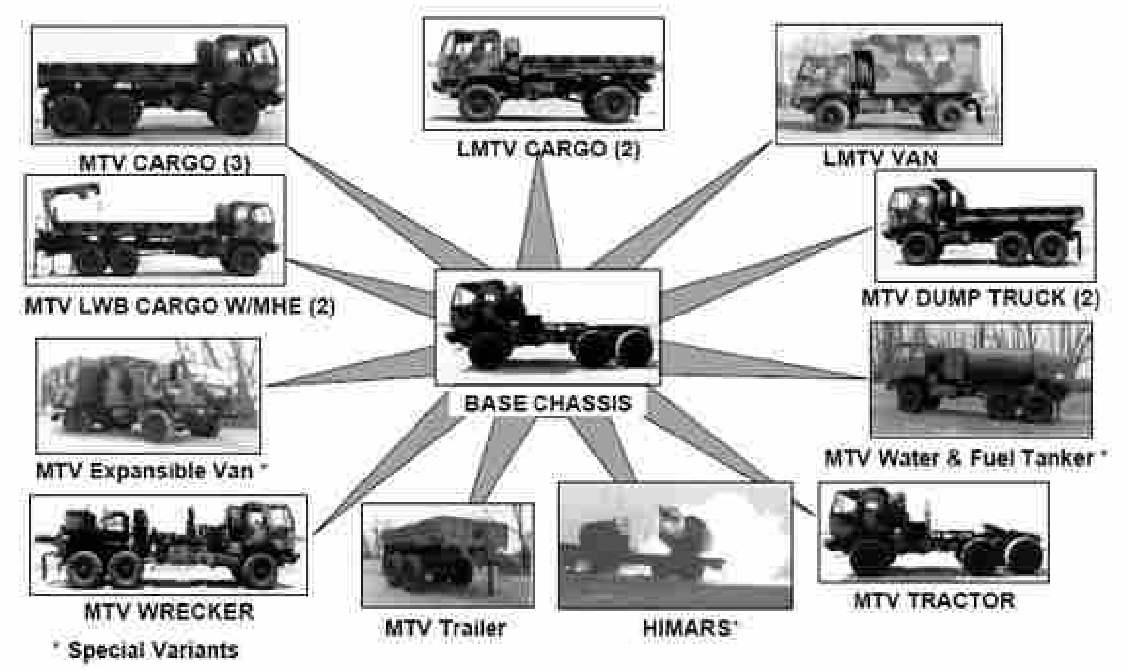


Figure 2.6: A base chassis system and all of the different configurations that create a family of medium tactical vehicles [1].

and MESH CAD environment. With the goal of revision history being feature based in the expansion to the NPDB, this allows for feature level configuration management which is made possible with this database expansion.

2.8 Neutral Parametric Database

The neutral parametric canonical form (NPCF) is a new neutral format for storing CAD information in a base mathematical definition [6]. In each CAD system a plugin is written that takes every feature that has been created and using that CAD system's API breaks the feature down into its mathematical definition. This mathematical definition for a feature is the NPCF. The NPCF has been implemented into a database format called the neutral parametric database (NPDB). Due to the NPCF being a neutral format with information to be accessed by many different CAD systems, no CAD-based PLM system exists to store NPCF information for use in Product Lifecycle Management (PLM). The data can be stored in PLM systems but many PLM features such as configuration management would not be available. Typical interaction between the CAD system and the PLM system requires the storage of binary part files such as NXs .prt files, or CATIAs .catprt files. There is no way currently to store PLM data in a synchronous, collaborative, multi-engineer, interoperable database environment. To save a part into a PLM database currently while using the NPCF and NPDB, the user must convert to the system that is interacting with their PLM service and save the file directly. Due to this conversion between systems, the benefits gained from having a multi-engineer synchronous heterogeneous database are lost because the saving process defaults to single user and is single CAD system reliant. Another issue is that currently saved CAD files are typically stored in binary and lack the ability to retain referential integrity, and therefore the part files can become corrupted. Within a database format, the product exists as a mathematical representation of the part and is inherently stable.

The current version of the NPDB stores the data for the following features; point 2D, point 3D, line 2D, line 3D, arc 2D, extrude, revolve, spline 2D, spline 3D, coordinate system (CSYS) and datum plane in their mathematical NPCF definitions. The database is set up in such a way as to store the current state of the part or assembly at all times. Every feature has a part ID that is attached to it and every feature associated with that part is loaded when opened. To ensure the proper loading order of the features, and thus the proper loading of the part, a concept borrowed from computer science called a multiton pattern is used which makes sure that every feature is loaded in the proper order [18]. This speeds up loading times compared to other multi-user CAD systems because the complexity of the loading algorithms is handled within the multiton pattern, so no computations are required for consistency, and loading proceeds quickly. The multiton pattern is a means of ensuring that there is only one object that exists for each name.

Referential integrity is the process of enforcing every database table to have all foreign keys point to a valid reference [19]. Most database engines will automatically handle references; however, this is only effective when the database has been set up in a way to handle every possible rule. Referential integrity will help the database recognize corrupted or incomplete data which in turn will be rejected from the database. Rejected data will remain on the client that attempted to send it instead of propagating through each client that is currently in the multi-user session. Further checks are implemented on the client side in each CAD system's plug-in to check that data sent to the database is consistent with the feature the user is trying to create.

CHAPTER 3. HYBRID STATE TRANSACTIONAL DATABASE

In order for data to be stored for more than just the current state, the NPDB will need to be restructured as proposed in Figure 1.1. The restructure will need to occur in such a way as to allow a list of the associated features, edits, and the feature itself to be stored. For the state system to remain intact each feature stored in the database will need to have complete data and allow the system to still load quickly without having to apply each edit sequentially. This will allow for quick loading of the part regardless of the state user is trying to load, from the first sketch or datum placed to the last edit of a feature.

The revised database structure will be validated when,

- A created feature adds a new object to the database with a new GUID and a state with a separate GUID.
- An edited feature adds a new state to the existing object with a new GUID.
- A second user can receive both creation and edits of features without duplication in the CAD file.
- Upon part load, only the most recent state of each feature is loaded.

In order to achieve the primary goal of this research, which is the implementation of configuration management and revision history into a MESH CAD environment, changes must be made to the NPDB. The current NPDB is a state-based database as described in chapter 1. In order to facilitate revision history this database structure must be changed to not only store the current state of the part but all previous states. Due to the nature of this database being state-based while also storing every feature change or transaction it will be referred to as the Hybrid State Transactional Database (HSTDB). In order to simplify naming the current state of the database before this research or NPDB will be referred to as D1 while the database with the changes made in this research or HSTDB will be referred to as D2.

If the current state of revision history used in PLM systems today were preserved in D2, a different part database would be saved off at every revision of the part and that would essentially be a new part. The saved database script is the database equivalent of saving off a binary file at different revision points as discussed in section 2.6. Saving off a database script does not utilize the power of a database and creates multiple copies of certain parts of the data. A database structure provides an opportunity to expand how many revisions are saved as well as reduce the amount of duplicated data while not significantly increasing the amount of storage space for the increased saved data.

D1 maintains referential integrity by having a linear tree structure that does not accept new data unless that data fits the database scheme seen in Figure 2.5. The state-based structure allows for comparatively fast loading with current database styled multi-user plugins such as NXConnect which have to perform each operation that was performed on the part in order, including edits and deletes, to build the part from scratch every time it is opened. In Interop, loading times have been decreased by only storing the current state of the part and so as the part is loaded it does not need to perform all edits and deletes but only adds the current geometry with its associated feature tree. These features are important and in the update to D2 they are critical to not increasing loading times.

If referential integrity were not implemented correctly data could become corrupted through database tables containing bad data. Likewise if state-based loading were not implemented then with each additional feature added to a part or assembly loading times would increase. The increased times would be a result of every single operation including edits being performed by the CAD system to load a part.

3.1 Methodology

The implementation of this database can be broken down into four key operations, which include: revision history, referential integrity, configuration management, and quick loading, and are explained in detail in the following sections. In order to achieve the desired database, Microsoft SQL Server Management Studio is used to add tables to the database and assign relationships. SQL is used as the programming language for the database. Finally, Microsoft Visual Studio is used to relate the data from the database to the local CAD system APIs using .edmx files. The .edmx files are used to view the database structure and edit the relationships represented by lines and dashed lines as well as tables for use in the local C# code files. Visual Studio will then be used to edit the C# files that interface with the CAD API and create the geometry. The full .edmx file for the new database can be seen in Appendix D.

3.1.1 Revision History

The main purpose in further developing the NPCF was to include full part history in the database. In order to achieve this, a new database structure was added to D1 or the neutral parametric database (NPDB). D2 can be seen in Figure 3.1 in comparison with D1 seen in Figure 2.5. The D2 image shows the addition of state-based tables in green that mirror the structure of the old database and are used but store state information for each feature. This is a high level image that has much of the functionality of the database removed for simplified viewing.

In order to make this change from Figure 2.5 to Figure 3.1, a State Table is added to the database through Microsoft SQL Management Studio. This new table needs to hold all of the data for each state of the feature shown in green in Figure 3.1. This table is added to the database as a child of the Object Table as shown in greater detail in Figure 3.2. This table also has a navigational property which is a relationship that links it to the Feature Table titled DBFeature in Figure 3.2. The relationship is a one-to-many relationship with one feature entry having a list of all the states associated with that feature. The reason behind this is that each feature has many states and each state is needed to preserve revision history. This small change caused a cascade of other necessary changes to the CAD plugins already included in the Interop environment. An example addition of a new feature can be seen in detail in Appendix E.

D1 had all database tables as children of the DBFeature table as seen in Figure 2.5. In order to store all of the feature data in the database, this data was moved under a new table called DBInteropState. The DBFeature tables such as Extrude and Sketch remain as children of DBFeature; however, they no longer contain the feature data as that data is stored in the matching DBInteropState table. The DBFeature table now stores a list of the states which then contain all the feature information for the state being saved.

When a user creates or edits a feature, a new state is created for that object which is a database table entry for a state as can be seen in the table located on the right side of Figure 3.2.

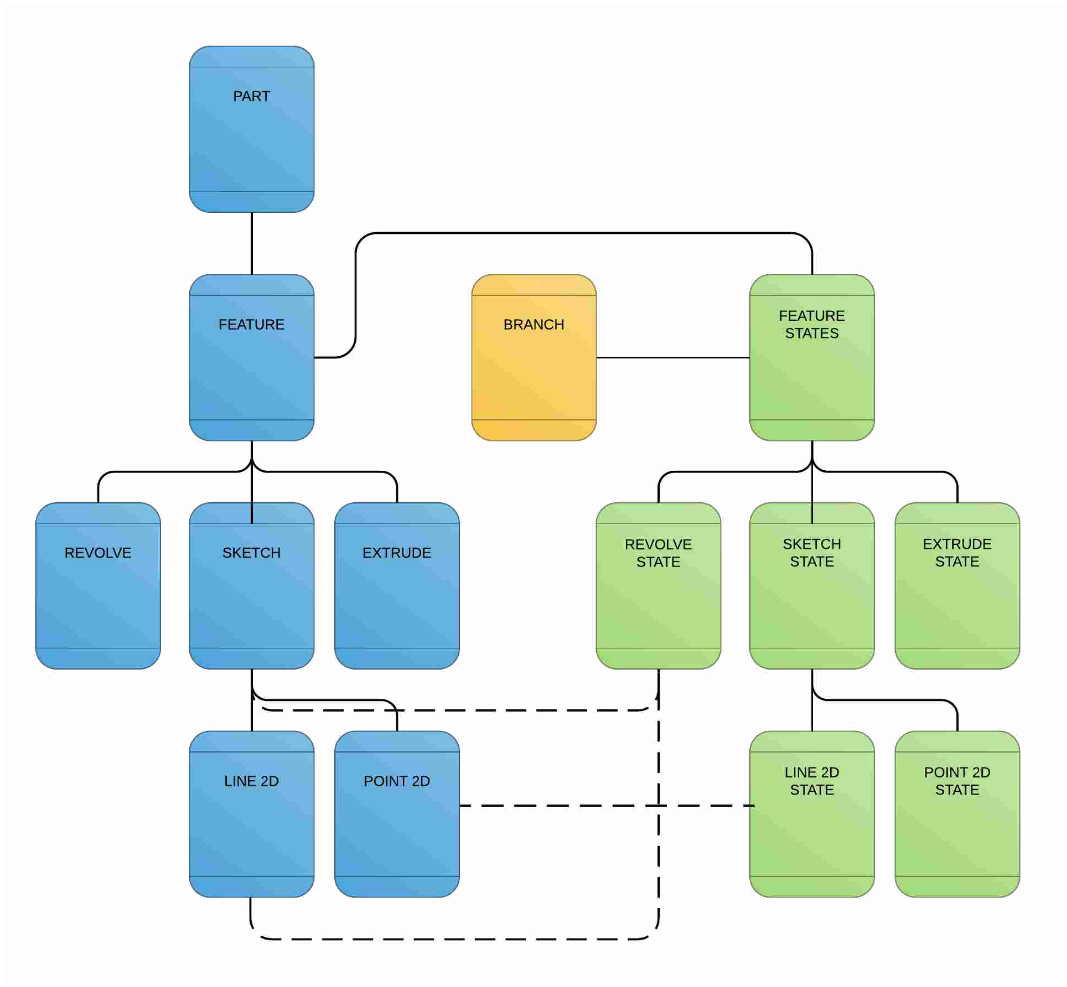


Figure 3.1: Updated simplified database showing addition of state tables as well as the branch table with some but not all relationships shown. This is in contrast to Figure 2.5 which shows the previous structure of the database.

If that state is a new state, a DBFeature is created along with the new state represented by the table located on the left side of Figure 3.2. When an edit is made, the DBFeature is first captured from the current state by following the navigational property for the state's DBFeature seen in the DBInteropState table in Figure 3.2. The edit is converted into a new state table entry and added to the DBFeature and time-stamped accordingly. The following example helps to better show how this process is achieved.

In Figure 3.3a, an extrude is added to the part based on the sketch already existent in the part. The figure also shows the data that is added to the database: a new entry to the DBFeature Table as well as a new entry in the DBExtrudeState Table. In Figure 3.3b, the extrude from 3.3a

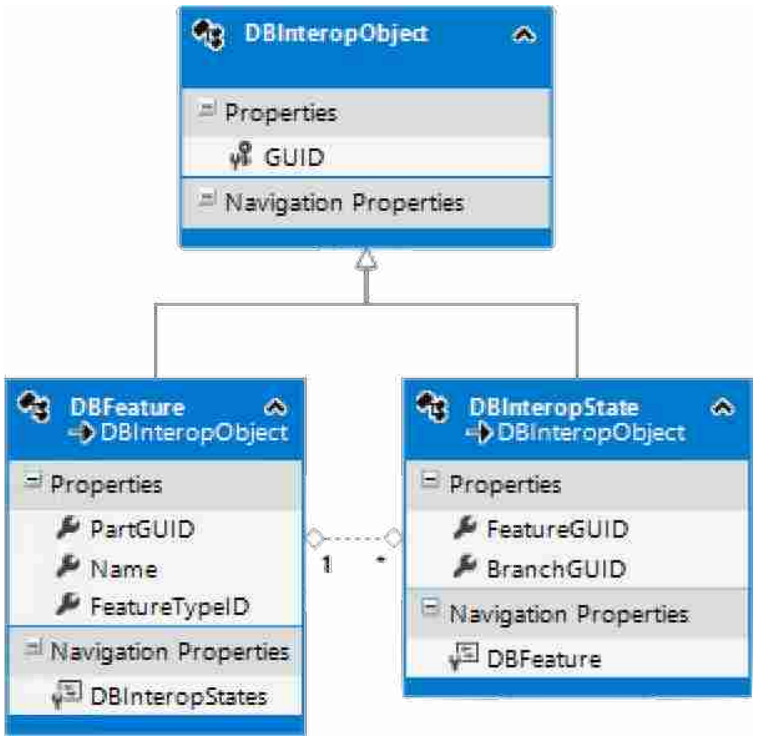
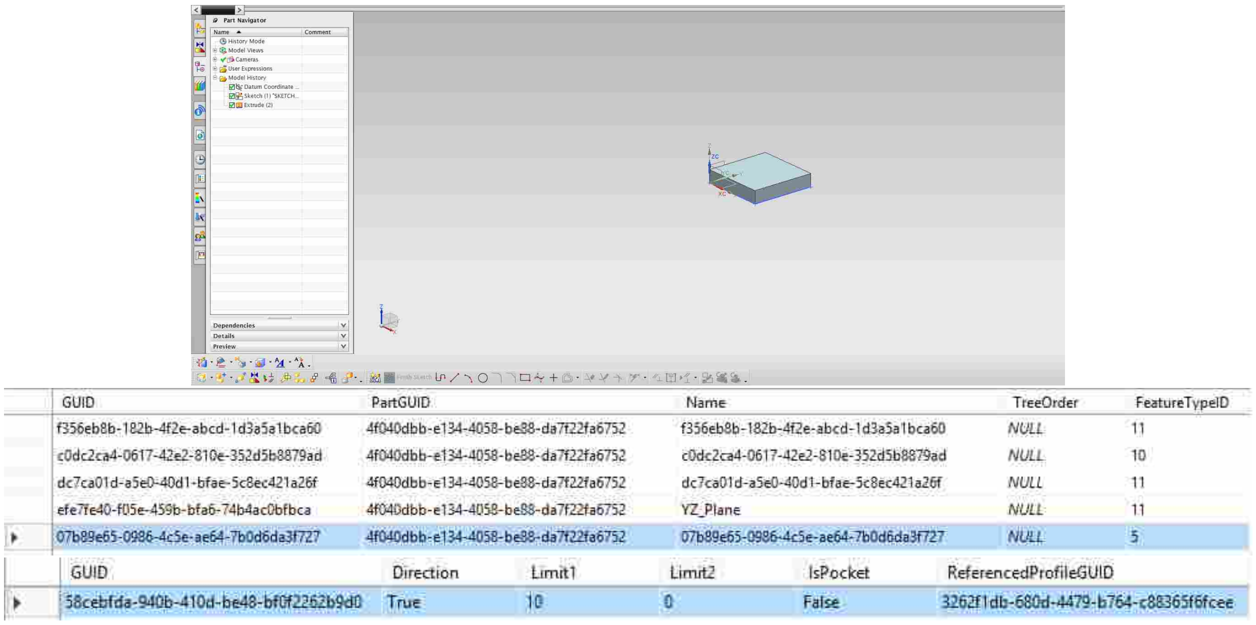


Figure 3.2: Database tables showing the relationship between DBFeature and DBInteropState with a one to many relationship.

is edited and a new entry to the DBExtrudeState Table is added. As can be seen from the figure, the table entry has the same DBFeature GUID as the DBFeature table in 3.3a. For a sketch, this process gets much longer as each feature in the sketch must go through the same process described. So if a simple box is made, ten table entries must be made: two for each point, each line, and the parent sketch which is the same process as seen for a sketch in Figure 3.3a.

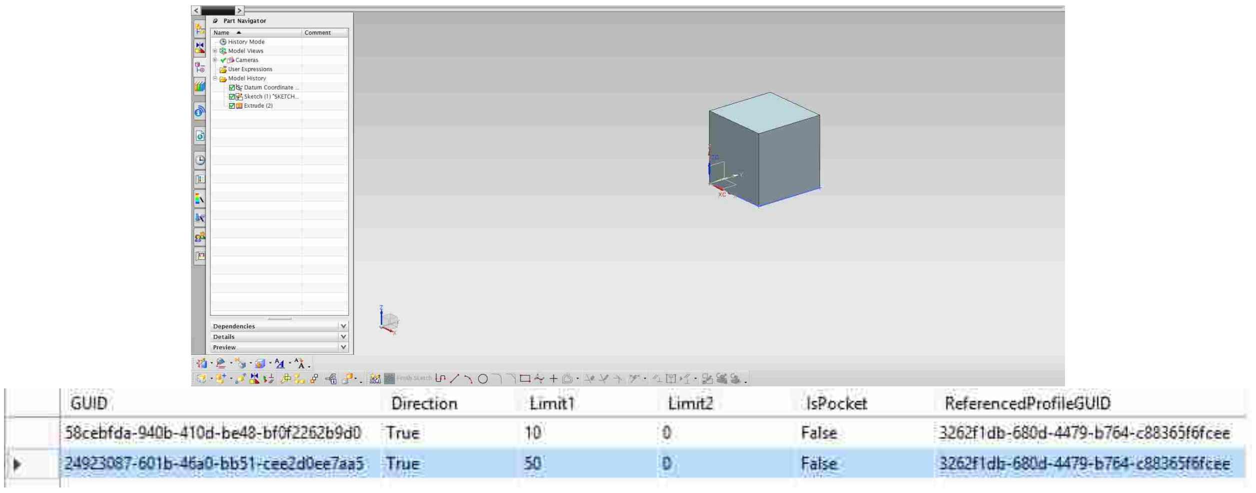
This process is captured by the system's API when a new feature is created or edited. The API recognizes that an operation has been performed and gathers the feature data. After gathering the data, the API translates the data into the NPCF and sends the data to the server. The server then communicates with the database and if the data does not fit into the database scheme it is rejected because the database enforces referential integrity as explained in section 3.1.2. When the server receives the data and it is not rejected by the database, the server sends that data to the other users in the part for implementation on their local machine. If rejected, the user that sent the data is notified that their part is now out of sync and they must reload. The process is not much different from the process described in section 2.3 other than each operation is stored in the database rather



GUID	PartGUID	Name	TreeOrder	FeatureTypeID
f356eb8b-182b-4f2e-abcd-1d3a5a1bca60	4f040dbb-e134-4058-be88-da7f22fa6752	f356eb8b-182b-4f2e-abcd-1d3a5a1bca60	NULL	11
c0dc2ca4-0617-42e2-810e-352d5b8879ad	4f040dbb-e134-4058-be88-da7f22fa6752	c0dc2ca4-0617-42e2-810e-352d5b8879ad	NULL	10
dc7ca01d-a5e0-40d1-bfae-5c8ec421a26f	4f040dbb-e134-4058-be88-da7f22fa6752	dc7ca01d-a5e0-40d1-bfae-5c8ec421a26f	NULL	11
efe7fe40-f05e-459b-bfa6-74b4ac0bfbca	4f040dbb-e134-4058-be88-da7f22fa6752	YZ_Plane	NULL	11
07b89e65-0986-4c5e-ae64-7b0d6da3f727	4f040dbb-e134-4058-be88-da7f22fa6752	07b89e65-0986-4c5e-ae64-7b0d6da3f727	NULL	5

GUID	Direction	Limit1	Limit2	IsPocket	ReferencedProfileGUID
58cebfda-940b-410d-be48-bf0f2262b9d0	True	10	0	False	3262f1db-680d-4479-b764-c88365f6fcee

(a) Extrude Creation With Data



GUID	Direction	Limit1	Limit2	IsPocket	ReferencedProfileGUID
58cebfda-940b-410d-be48-bf0f2262b9d0	True	10	0	False	3262f1db-680d-4479-b764-c88365f6fcee
24923087-601b-46a0-bb51-cee2d0ee7aa5	True	50	0	False	3262f1db-680d-4479-b764-c88365f6fcee

(b) Extrude Edit With Data

Figure 3.3: Data from creation and an edit add to the database.

than just the most recent state of the part, also no previous data is overwritten as it was in D1 on edits.

In Figure 3.4, an example of revisions saved in a current CAD system can be seen where a user manually saves off revisions which are represented by the blue bars labeled Revision 1 and Revision 2. D2 now has data that is CAD neutral and users never have to save revisions. The green lines on the right side of Figure 3.4 show all of the revisions available to the user and no saving is needed. Every step in the creation process is automatically captured in the database. This helps

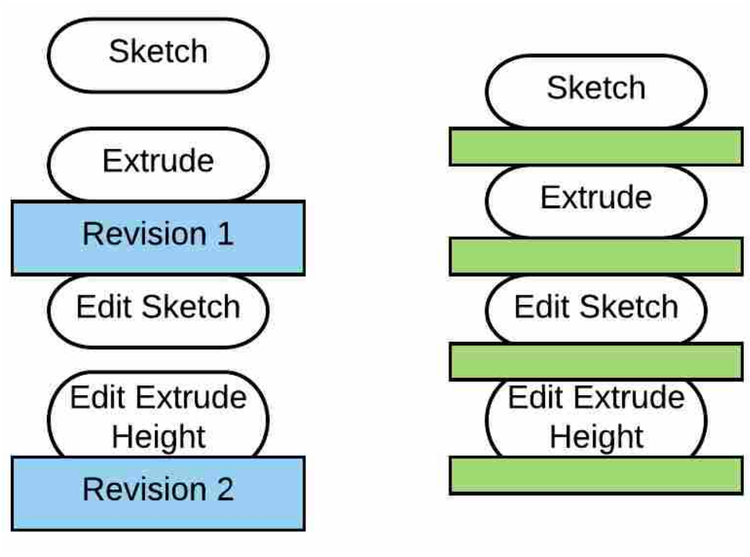


Figure 3.4: Simple example showing typical CAD systems revision history with saving in blue on the left and the new revision history without saving in Green on the right.

capture design intent as all edits to the part can be viewed chronologically. Due to the neutral format and only saving the necessary data to recreate the part the file size for the database is less than that of traditional CAD systems which will be discussed in greater detail later.

3.1.2 Referential Integrity

D2 creates a database that has an entry for each feature. That database entry has a full list of states for every edit in the parts history. Unfortunately, this method still does not fully address referential integrity. Data can be corrupted by states being linked to features that represent a feature that is not compatible. To address compatibility of features, all of the navigational properties for the objects had to be changed to point to the correct feature table. This is unintuitive for database design because all of the tables that inherit from the Feature table have no values other than a GUID which act as the primary key for the tables. These tables are simply in the database to make connections which can be seen as dotted lines in Figure 3.1. These sub tables were not necessary to set up the references needed to create the states but they were necessary to maintain referential integrity. It would have been possible to have all navigation properties point to the Feature table and control the proper association of data from the code. However, without referential integrity at the database level, data being sent to the server can be corrupted. Corrupted data can lead to features or parts that will not load resulting in lost work.

In order to maintain tighter referential integrity, a FeatureTypeID is added to the DBFeature object. The FeatureTypeID is simply an integer value that represents a feature. This FeatureTypeID acts as a check for when a client adds a state to a feature. If the FeatureTypeID matches the ID of the state object then the addition is allowed. If the FeatureTypeID does not match then the addition is not allowed and the addition is rejected from the database as it implies that the state was not created in the correct manner. This provides the code a way to check to make sure that the data being sent to the database will not be rejected by the referential integrity structure changes between Figures 3.1 and 2.5. This change to the database structure allows the programmer to check the type of the data stored in the table code-side. No tables have data other than DBFeature on the left side of the new database in Figure 3.1.

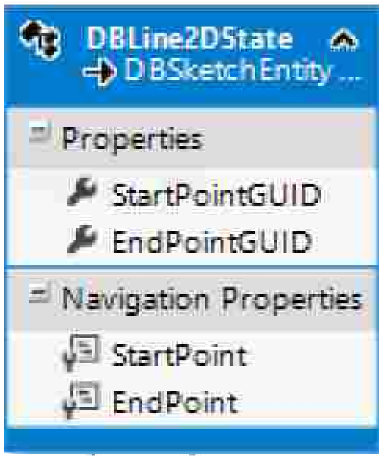
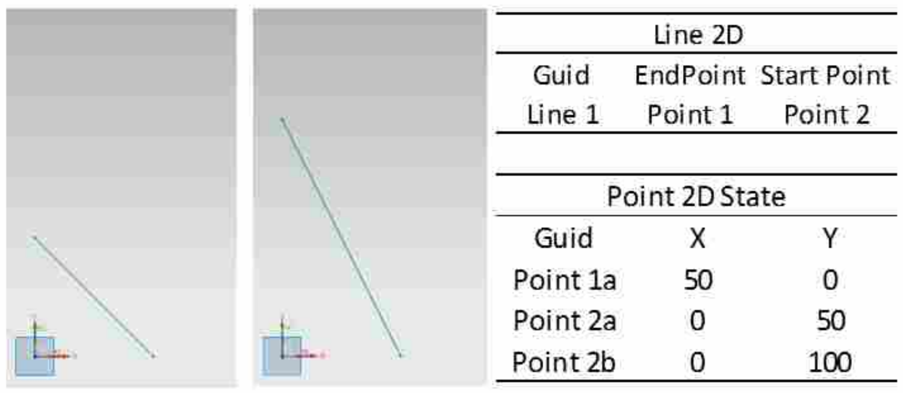


Figure 3.5: Line2DState table showing the references to Start Point and End Point.

In D2, DBInteropState is the parent table of many different state tables including a number of sketch tables such as DBPoint2DState and DBLine2DState. These tables store the state data for the points and lines that reside within a sketch. Inside the DBLine2DState, there are two navigation properties. These navigational properties are the links that show which points in D2 are the associated start and end points for that line, as can be seen in Figure 3.5. With the addition of states to D2, the endpoint and the start point can each contain a list of states. These states are contained within a DBFeature's DBInteropStates reference list. Thus, a line state has a reference to a start point, which has a list of states for every revision of that point. The challenge is that with the change in the database every reference goes back to a DBFeature object because that is what contains the state list. Although the state list does not allow for different states to be stored, from the perspective of the DBLine2DState table, there is no way to know if the reference to its start point is actually a point2D or an extrude. Therefore, this method does not maintain referential integrity.

In order to maintain referential integrity, a matching schema under DBFeature was developed that mimics the architecture under DBInteropState. For every state table, a matching DBFeature object table such as DBLine2D that inherits from DBFeature was added. This way, when a feature such as DBLine2DState references two external DBFeatures referential integrity can be enforced through the existence of DBPoint2D as a table. The DBLine2DState has two references in it that are one-to-one; they each point to a separate DBPoint2D object and each of these points has their own state tables. The correct state is chosen to represent that line at any given time. This table is nearly blank, containing an inherited global unique identifier (GUID) from the parent DBFeature that allows the table to be referenced. This forces an external reference that is contained in a state table to point to a DBFeature that is an instantiation of the feature that is actually desired, thus maintaining referential integrity.

This structure also allows the database to store a DBLine2D state that never needs to change when the referenced points are edited. It does this by allowing the points to have separate DBPoint2DState tables which can have their own states. This way, when an edit is made to a point, a state table is added to that point feature, but it is not needed to change the DBLine2D or add a state to it. In Figure 3.6 this process is shown. In the first figure of a line, the end points are at (0,50) and (50,0) respectively. In the second figure, the line has been edited to have end points at (0,50) and (100,0). The database data that is needed to represent those two different line configurations can be seen on the right of Figure 3.6. The Line 2D Table has Line 1 which contains two points: Point 1 and Point 2. The Point 2D State Table contains the initial points Point 1a and Point 2a and then the edited Point 2b which has the new coordinates. The Line 1 entry does not have to be changed because it still references Point 2 and can simply gather the data for the newest state of Point 2 which in this case is Point 2b.



Line 2D		
Guid	EndPoint	Start Point
Line 1	Point 1	Point 2

Point 2D State		
Guid	X	Y
Point 1a	50	0
Point 2a	0	50
Point 2b	0	100

Figure 3.6: Figure showing two different lines that are based on the same points with one edit and their associated database table data.

3.1.3 Configuration Management

Configuration management is the process of producing and managing different configurations of a part that all contain geometry from multiple configurations. An example of this would be a car manufacturer having two CAD assembly files for their regular and luxury models of each of their designs. These two files share a lot of the same geometry. For example, they may have the same seat belt designs and they may have the same audio system and all its associated parts. This type of data is all relevant for making the various models of the car. Currently, if a change is made in a part, a new part file is created. Continuing with the car example, if a car company desired their sport edition model to have a hood scoop, they would keep the geometry of the rest of the hood and add a scoop to the center. This results in a new part file and this new part file duplicates a lot of the data that is contained in the original part file. To implement a new database structure that would allow the user to add a hood scoop to one part file and keep the original part file for the other model, a method from computer programming was applied called branching [20]. Branching allows coders to work simultaneously where they each have the files necessary for editing but only see their edits. When the edits are completed the users can merge their code together for complete code. Merging CAD files is beyond the scope of this research and will not be implemented.

In order to keep track of branching, a branchID was added in the form of a GUID to the DBInteropState table which can be seen in Figure 3.2. A user can create a new branch manually which creates a new branchID. This branchID serves as a unique identifier for the current branch that the state is created on. Figure 3.7 shows three points being created and edited across multiple branches. This example goes to show how multiple users can work synchronously in the same base part and then take advantage of the branching functionality. The new approach to configuration management in D2 allows configurations to be handled at a feature level and not a part level.

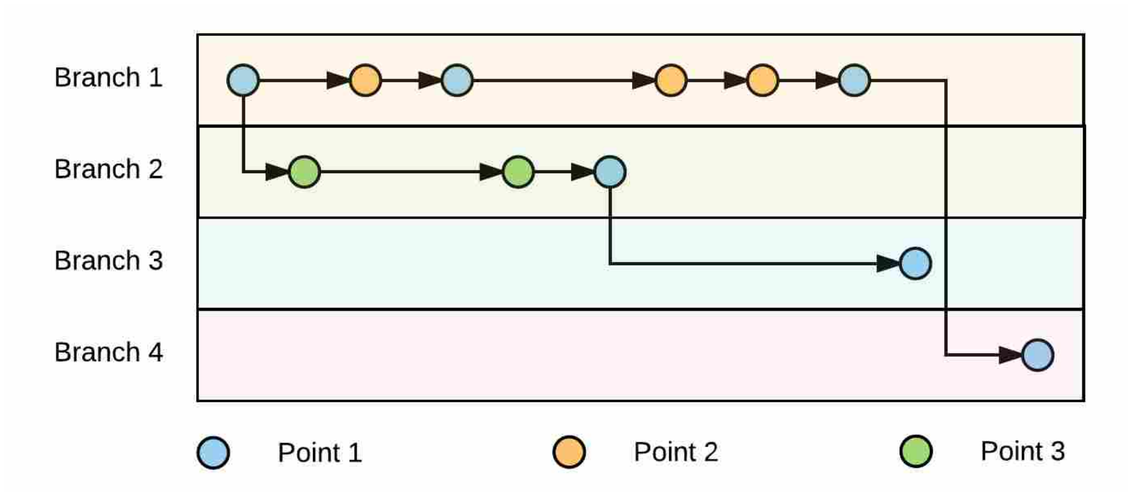


Figure 3.7: Example timeline of three points being created and edited on different branches at different times.

Figure 3.7 shows a part that has four different desired configurations labeled Branches 1 through 4. In Branch 1 multiple users start out in the part; each user can use the CAD system of their choice between NX, CATIA and CREO. One of the users creates the first point in any system while all other users receive the addition of a point in real time. At this point, a few users create a new branch which adds a DBBranch table entry for the creation of that branch. They then proceed to create a new point labeled Point 3, edit that point and subsequently edit Point 1. At the same time the users that remain in Branch 1 create a new point labeled Point 2 and edit Points 1 and 2 multiple times. At this point two different users create Branches 3 and 4 and edit Point 1 on each branch. At the end there are four different branches with three points in different configurations, multiple users are allowed in each branch at the same time on different CAD systems and when a user makes an edit on a branch other users do not receive that edit unless they are currently working on the same branch on which the edit was made.

Another example where branching may be useful would be in the case of designing a bike rim as can be seen in Figure 3.8. In this example two users enter the part and create the center and outside edge of the rim as can be seen in Figure 3.8a. One user creates a second branch of the part and adds a configuration that has a few spokes as shown in Figure 3.8b. At the same time a second user creates a configuration on a separate branch with many spokes as can be seen in Figure 3.8c. In this example both users were able to use the geometry from Figure 3.8a without saving a part file, copying that part file and uploading the part file into the database. They are also able to work in different CAD systems simultaneously with one user using NX as can be seen in Figure 3.8 and the other user using either CATIA or CREO.

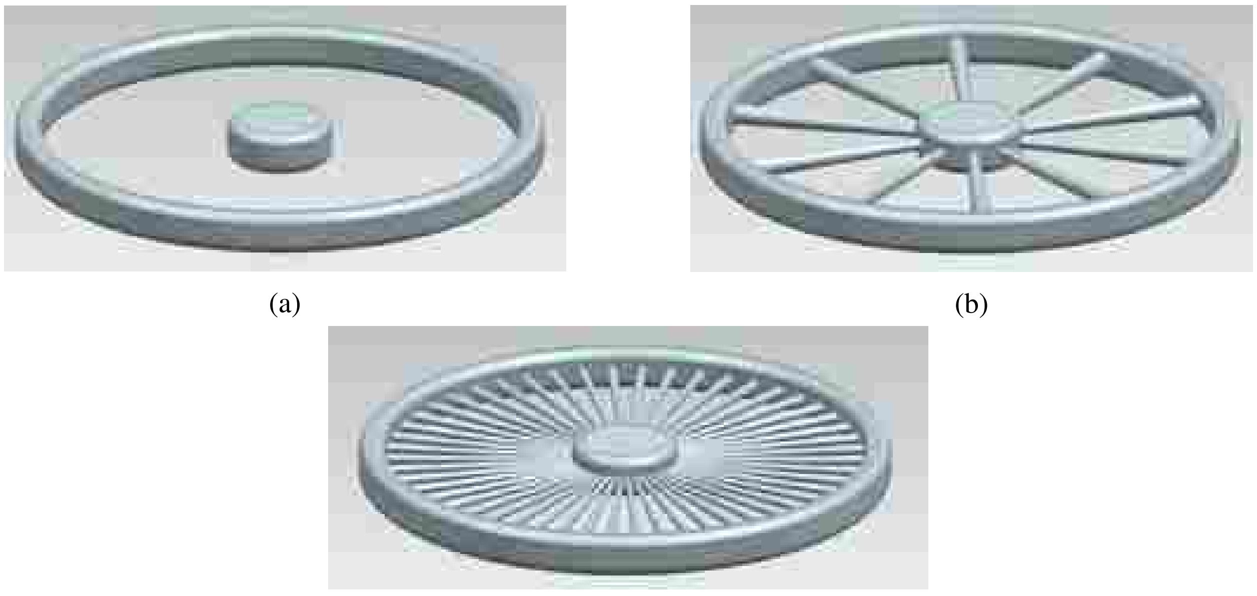

(a) (b) (c)

Figure 3.8: Branching example for a bike rim. The base features being created in 3.8a and two different configurations of the part being shown in 3.8b and 3.8c.

This method configuration management is new in that it is handled at the feature level instead of of the part level. It is also new in the fact that users can work across CAD systems simultaneously. A benefit added to the design process through the addition of this method of configuration management is that the analytics team could create a branch of the part at any point, take that part and run analyses while the design team working on the part does not need to check in the part, save, or do anything to allow the analytics team to have the most recent version of the part. They can also continue working on the part while the analysis is running. No longer are parts stuck in a silo of design or analysis but can be worked on simultaneously by both parties.

Table 3.1: Table showing the creation and edits of 3 points on multiple branches in order, according to time with a branch being created in the middle and one point being edited on the new branch.

PointID	XCoord	YCoord	BranchID
1	0	0	1
2	1	1	1
Branch 2 Created	-	-	-
1	1	4	2
2	1	2	1
3	4	3	1

Table 3.1 shows a hypothetical, incomplete list of states for a DBPoint2D feature to serve as an example of how this branching method works. This table is created as a user creates two points: one at coordinates of (0,0) and one at (1,1). This part is then branched and point one on the new branch is then changed to coordinates (1,4). The users then return to branch one and move point two to coordinates (1,2) and create a new third point at coordinates (4,3). Although this simple scenario demonstrates how this implementation works, the fundamental steps would be identical in the context of a real and more complicated scenario. If two users were actively editing this part concurrently and one was working on branch one and the other was working on branch two, then the user working on branch one at the end of this editing scenario would have three points at coordinates of (0,0) (1,2) and (4,3). The user working on branch two would have two points located at coordinates of (1,4) and (1,2). Table 3.1 is a simple chronology of events detailing what has happened in the part up to this point in time.

Table 3.2 shows the Branch history of Branch 1 while Table 3.3 shows the history of Branch 2. In order for the full history to be preserved, Branch 2 contains all of the information from Branch 1 from before the creation of the new branch as well as all additional changes made to Branch 2 after the new branch creation. As expected, Branch 1 contains all data for the changes

Table 3.2: Table showing the edits and creations that took place on Branch 1.

PointID	XCoord	YCoord	BranchID
1	0	0	1
2	1	1	1
2	1	2	1
3	4	3	1

Table 3.3: Table showing the edits and creations that took place to make up Branch 2.

PointID	XCoord	YCoord	BranchID
1	0	0	1
2	1	1	1
1	1	4	2

made on Branch 1. Since the point state table really only contains four point states, this schema has successfully created two parts without duplicating data which reduces the total storage size of the objects. A downside for this method is that the state-based loading scheme, mentioned in the introduction, does not hold and loading becomes more complicated. However, data is still not duplicated and one additional step in the loading process is added for every branch that is created in sequence to mitigate this issue.

One other benefit of this process, which is beyond the scope of this research, is the ability to keep features associative with a parent feature. In one scenario, multiple users could be editing a base feature while working in a different configuration of the part. In this scenario every feature created before the branch was created would be a collaborative feature that can be edited by every user in the part. Every feature that is created after a branch is created would be editable only by the users which have the branch on which the feature was created loaded. An example of this would be if a car rim designer wanted to create two different rims but both models needed to have the same mating mechanism for connecting to the axle of the car. In this scenario, if a user were to design the mating mechanism and then create a branch where they create the second design of the rim, they could have the mechanism be associative to the first design. Therefore, when a change needs to be made to the mating mechanism, it could be made in both designs simultaneously by only editing one of the branches. No wave linking would have to take place, no added features would be needed, and the user could choose to make that section of the part associative. The potential of this technique would require additional research into design practices and finding use cases where this could be beneficial. However, it is an interesting research topic that is now enabled through the research contributions described herein.

3.1.4 State-Based Loading

Previous attempts at multi-user CAD, such as NXConnect, implement a full history transaction database that handles loading by performing all operations performed by users after the last part save in order to create the part [21]. This is a good method of making sure that all users in the part are at the same point and have the same data after loading. The problem is that if the part has not been saved recently, then there are many operations that need to be processed and the part takes a long time to load [22]. Also, within a heterogeneous CAD environment, the CAD files cannot be saved into their respective CAD systems' part files without losing the benefits created by having the NPCF, including the fact that every time the part is loaded it would have to be translated back into the NPCF for use by the other systems. A user could have changed his/her mind multiple times about the height of an extrude, and as a result, the loading process itself would change the extrude multiple times. The benefit of D1 is that it stores the current state of the part so only one action is performed per feature and no edits are computed.

Proper application of branching is vital to implementing a fast loading scheme. The current loading scheme can always be fast because it has one state of the part, which is extracted from the database, and then constructs the geometry through the CAD system's API. With the additional information stored in the database, the client needs to know which state is being loaded and on what branch. When the part has multiple branches, the server needs to know which latest version of each feature must be loaded by the client and send that data to the client to open the child branch without errors.

The loading can become fairly complicated when multiple branches exist, but the fundamentals can be described in the terms of the example shown above in Tables 3.1, 3.2 and 3.3. In order to load Branch 1, the latest state of each feature with the branchID equal to Branch 1 would be loaded into the part and loading would be complete. Loading for a single branch is as simple as fetching the last state saved in the database.

For loading Branch 2, the schema loads all of Branch 1 up until the creation of Branch 2. It then loads all updates to those features that exist on Branch 2. The process, however, does not load the updates to those features that exist on Branch 1. Essentially, it loads Branch 1 up until Branch 2 and then switches to Branch 2 and does not execute any of the changes that are on Branch 1. Creating new states is as simple as adding a new state with the appropriate branchID for the current branch. This does not increase load times in the Interop environment.

This approach to loading is not an improvement on traditional CAD part loading for single user systems; in fact, since no binary files are used it is slower than traditional loading as the API must perform all of the operations to create the part on load. The length of time this loading takes is dependent on the number of features in the CAD system and grows as more features are added. For a simple part such as a cube the user will not notice a difference in loading times. For larger assemblies the load time would increase more but has not been tested as many larger assemblies can not be created at this time with the limited feature set present in the Interop environment. Figure 3.9 shows the loading operations needed to load a part that has a Datum Plane, a sketch and an extrude as well as edits to the sketch and the extrude. Single user CAD saves all of the data needed to load the part in the part file which allows it to only require one loading operations. NXConnect has to load every single feature as well as every single edit making a total of five loading operations. One benefit to working in a single CAD system is that part saves can be stored in the database when a user saves. This allows NXConnect to achieve the NXConnect Hybrid column in Figure 3.9 by using a combination of single user and standard NXConnect loading. For NXConnect, when a user hits the save button a copy of the part is sent to the database which allows it to act like a single user CAD system on loading up to the most recent save. After the most recent save is loaded the remainder of the part is loaded. In Figure 3.9 a user saved the part after the sketch edit so the extrude edit was a new operation. This hybrid system is possible with the Interop environment but has not been implemented. For Interop D1 and D2 the sketch and extrude features can be skipped in the loading process and the sketch edit and extrude edit can be loaded directly, reducing the number of loading operations that need to be performed. D2 is designed to maintain the state-based loading knowing which states of the part to not load, where D1 does not contain the information to load any previous revisions of the part.

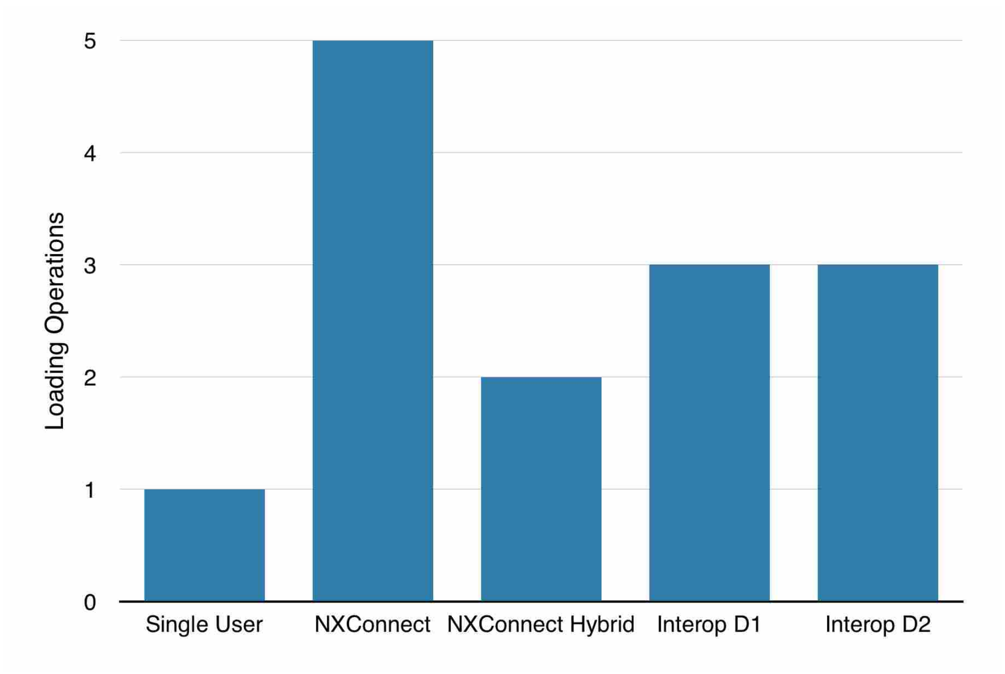
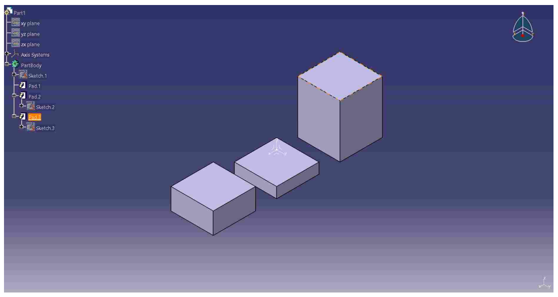


Figure 3.9: Loading Operations based on CAD Package.

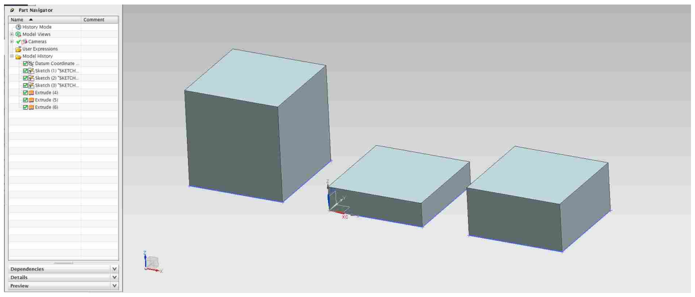
3.2 Results

As a result of this research, D1 has been expanded while still utilizing the NPCF to allow for incorporation of revision history and configuration management. These added capabilities were successfully demonstrated by a team of users by modeling each feature supported in the database in NX. The database correctly stored each state of the part as can be seen in the next chapter. Upon successful creation of each feature, users were asked to edit each feature. After successful edits were made to each feature and propagated across all clients, the database was manually checked to ensure that data was stored correctly as can be seen in Figure 3.10c. No specific model was used and users had the freedom to create features in whichever manner they saw fit. Edits were done in the same fashion allowing users to make changes as needed.

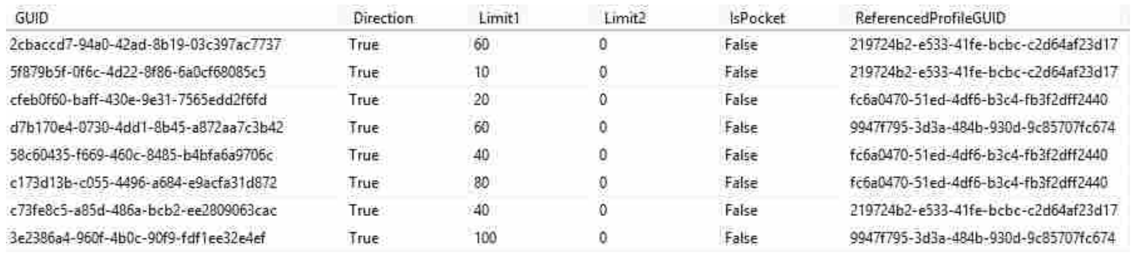
In order to test the D2 across multiple CAD systems, the extrude feature was implemented in the plugins for CATIA, CREO and NX. Three users each working in a different CAD systems at the same time on different machines modeled a simple extrude as seen in Figures 3.10a and 3.10b. The users then edited the height of the extrude multiple times as can also be seen for comparison in Figures 3.10a and 3.10b. Each client was able to pull the extrude made by the other clients and apply them on their own machine as well as update the model and receive edits on their machine. As before, in order to ensure that data was pushed for not only the creation of the extrude but also for each subsequent edit the database was checked manually and verified its implementation as can be seen in the next chapter. This validates the process because the data shows the different limits used by the multiple clients on the three different Referenced Profile GUIDs. Each state contains its identifying GUID, while each DBFeature is associated with multiple states demonstrating proper function of the database where each state should have its own GUID which is represented in Figure 3.10c in the left column.



(a)



(b)



GUID	Direction	Limit1	Limit2	IsPocket	ReferencedProfileGUID
2cbaccd7-94a0-42ad-8b19-03c397ac7737	True	60	0	False	219724b2-e533-41fe-bcbc-c2d64af23d17
5f879b5f-0f6c-4d22-8f86-6a0cf68085c5	True	10	0	False	219724b2-e533-41fe-bcbc-c2d64af23d17
cfeb0f60-baff-430e-9e31-7565edd2f6fd	True	20	0	False	fc6a0470-51ed-4df6-b3c4-fb3f2dff2440
d7b170e4-0730-4dd1-8b45-a872aa7c3b42	True	60	0	False	9947f795-3d3a-484b-930d-9c85707fc674
58c60435-f669-460c-8485-b4bfa6a9706c	True	40	0	False	fc6a0470-51ed-4df6-b3c4-fb3f2dff2440
c173d13b-c055-4496-a684-e9acfa31d872	True	80	0	False	fc6a0470-51ed-4df6-b3c4-fb3f2dff2440
c73fe8c5-a85d-486a-bcb2-ee2809063cac	True	40	0	False	219724b2-e533-41fe-bcbc-c2d64af23d17
3e2386a4-960f-4b0c-90f9-fdf1ee32e4ef	True	100	0	False	9947f795-3d3a-484b-930d-9c85707fc674

(c)

Figure 3.10: Final state of extrudes shown in both the CATIA (3.10a) and NX (3.10b) CAD clients. Database table data (3.10c) for extrudes shown in 3.10a and 3.10b as well as all edit data. Views are not from the same viewing position but the CAD parts are the same.

To test for the correct implementation of referential integrity, users were asked to constrain a sketch line to a point that was outside of the current sketch in three-dimensional space which should have a table identity of Point3D. Line2D has references for start and end points that only allow Point2D as seen in Figure 3.5. Since the point being referenced was a Point3D but the database expected a Point 2D, this data was rejected from the database and the other users in the part did not receive this edit. In a commercial CAD system, this action would be allowed, so further research is required to allow for faces and edges of objects to be used as reference objects.

Branching separate parts is functional in the database and the database knows how to handle the received information. However, no method has been instantiated in the CAD plug-ins at the moment that would allow the user to access this functionality. For testing purposes, the branchIDs were entered manually to test the functionality of the database on receiving the data. The database worked as expected in this regard rejecting bad data and storing the data correctly for multiple different branches.

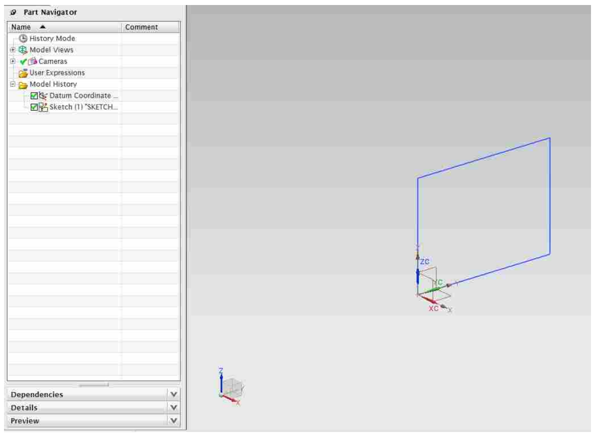
The objective of this research was to expand D1 to allow for advanced PLM features to be implemented locally and utilize the NPCF. Based on these validation tests, D1 has been correctly enhanced to D2 and is working appropriately to store all the data for creating, deleting, and editing features in a MESH CAD environment.

CHAPTER 4. REVISION HISTORY

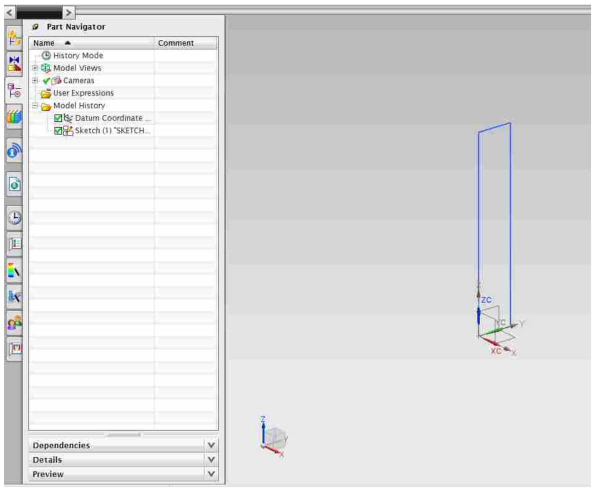
The database restructure presented in Chapter 3 is beneficial to store data, but it does not allow the user to interact with the data. If the user cannot view the data in a meaningful way, there is no way for the increase in data to be of value to the user. By providing a graphical user interface (GUI) for the user to view the list of previous revisions this allows the user to select and view the model of the previous versions of their own and others, preserving design intent by showing how features were created and in what order. Without providing access to the design history, there is no immediate benefit to storing the data in the first place. A GUI is needed for the user to be able to see the information in the database in a readable format and be able to interact with that data. An example part is shown with steps to create that part in Figure 4.1. All of the different loaded file formats can be seen in Figure 4.2. For the new database, instead of having a feature tree of data, every edit is available to the user as illustrated in Figure 4.2.

As can be seen in Figure 4.2d, the goal of revising the database is to allow the user full access to not just the current data contained in the part, but also the full history of creation so that they can return to any point in the design process. By returning to a point in the design process, the user can more fully understand why certain design decisions were made and can gain insight into how the part was created. With the changes to the database to allow for configuration management as discussed in Section 3.1.3, the user also would have the ability to save off the part at any given time and then continue working in a different direction. This capability provided to the user is discussed in greater detail in Chapter 5.

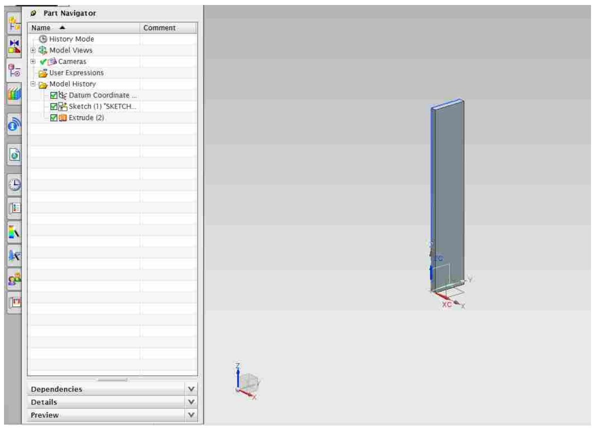
In order to allow the user to view previous revisions, a GUI will be presented to them. This GUI will show them the states of the part by accessing the states from the database. The user then has the ability to load any state and view what the part was like at that state. This will trigger a boolean flag in the client that is used to check if the client is currently in a previous state or revision of the part. If the client has checked out a previous revision of the part, then when messages are



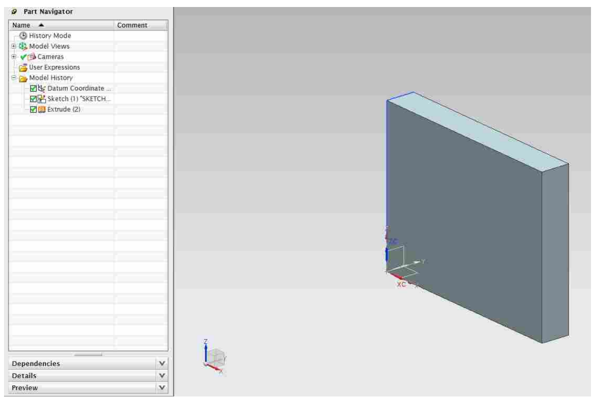
(a) Step 1: Create Sketch



(b) Step 2: Edit Sketch



(c) Step 3: Create Extrude



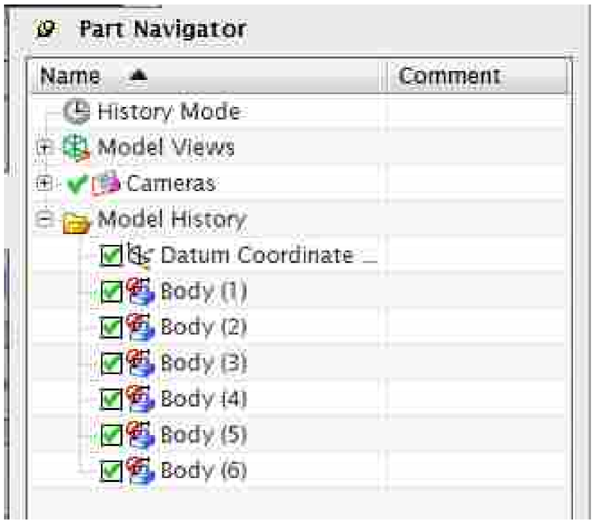
(d) Step 4: Edit Extrude

Figure 4.1: Steps to create the final part used to show different part save files.

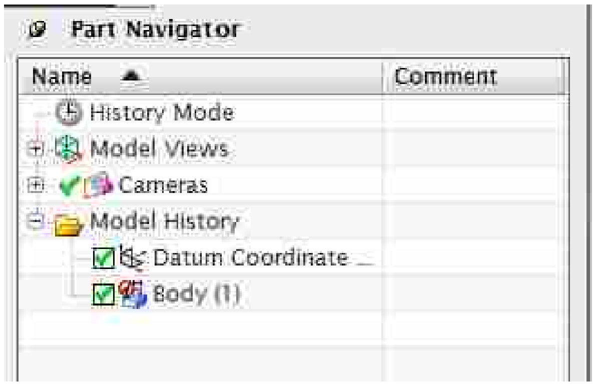
sent from the server to the client they will be ignored and not loaded. This allows the user to view the previous state of the part without changes being made to the part from other users. This also prevents errors from occurring because features needed for the incoming feature messages and edits may not be present in the loaded revision.

Revision history will be validated when:

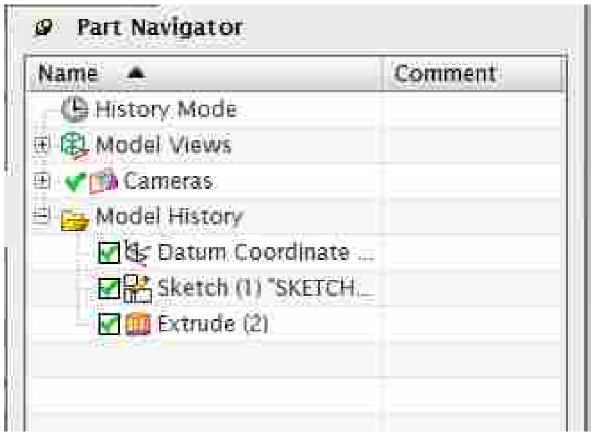
- A user can view previous states of the part in a feature list to select which to load
- A user can select a previous state of the part and load the part at that state



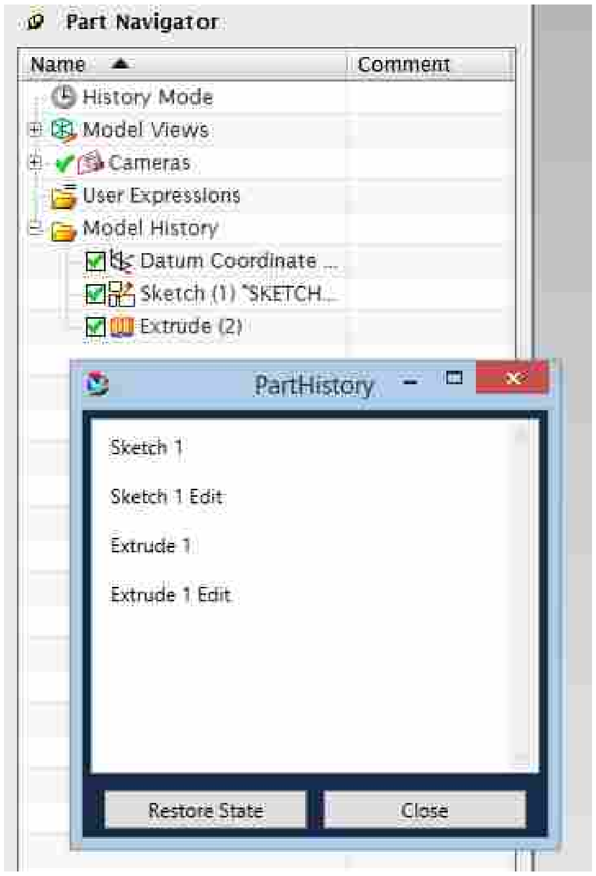
(a) Feature tree of part loaded from .igs file.



(b) Feature tree of part loaded from .stp file.



(c) Feature tree of part loaded from Interop and NX file.



(d) Feature tree of part loaded from new database file also with the part history window showing all steps in the creation process.

Figure 4.2: Feature trees from different save files.

4.1 Methodology

To create a user interface that allows the user to interact with the CAD file's revision history, Microsoft Visual Studio was used to create and edit .xaml files in both a visual manner and directly in the code. The .xaml code references functions in the C# code files which then interact with the API. For this section no changes need to be made to the database because all of the changes needed to allow for the integration of revision history were taken care of by the changes addressed in Chapter 3.

4.1.1 GUI and Loading

In order to provide the user with the revision history data a simple GUI has been developed and can be seen in Figure 4.3. This GUI has a list of all of the actions performed in the part as well as a restore state button and a close button. The list of actions in the figure are shown with names from the item they are made from. Since none of the objects have a name other than a GUID object, the name comes from the table that they are associated with. This GUI is accessed from the CAD system's menu bar, adding a new button to access the window.

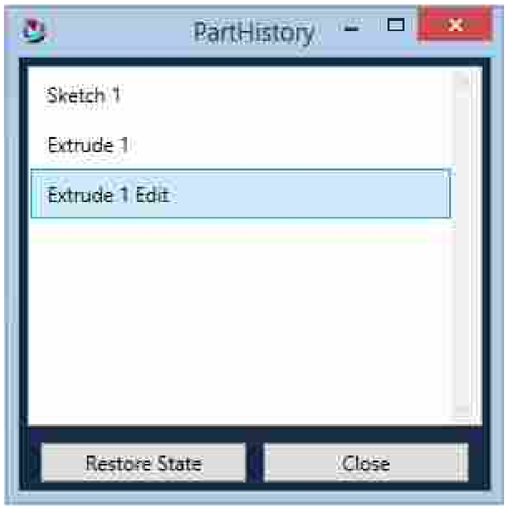


Figure 4.3: Image showing the GUI and some revisions for a part with a button to restore the part to different revisions and close the window.

In order to restore a previous state in the part, a state is selected from the list and then the restore state button is pressed by the user. If no selection is made the restore state button performs no action. If a state is selected then the state is passed back to the main Interop function and a timestamp is used to filter out unwanted features.

The part will already have all of the features loaded from the initial part open, even if it is currently in a previous state. To demonstrate the states of the features that existed in the part, the algorithm, seen in Figure 4.4, was developed to filter out all features and feature states that did not exist at the time of the revision that is being loaded. These features are filtered based on the timestamp that each feature and state have compared to the selected state's timestamp.

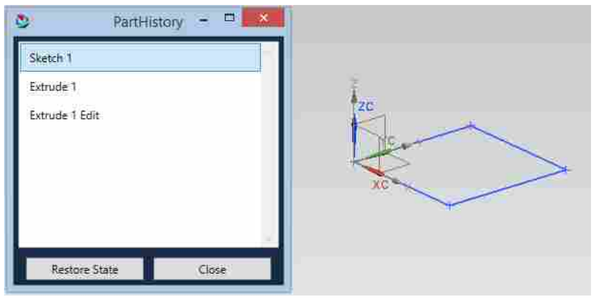
```
Data: Timestamp
Result: Previous revision loaded
for Each Feature do
    for Each Feature State Contained in the Feature do
        if (State Timestamp < Current Revision Wanted Timestamp) then
            Keep the feature to which the state belongs
        else
            Remove the state because it has not been created yet
        end
    end
end
```

Figure 4.4: Loading algorithm for a previous revision.

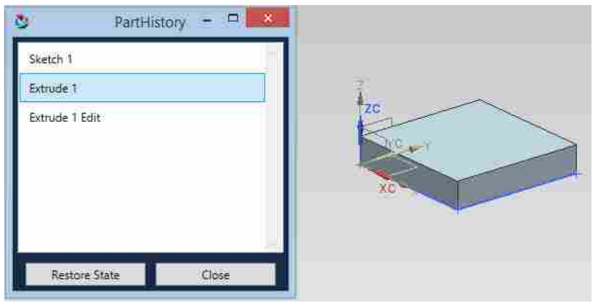
The algorithm, seen in Figure 4.4, works to load just the features and states present at the time of the create or edit to the part file. This algorithm is not branch specific so it will work with branch loading, discussed in Chapter 5. The GUI presented to the user will also be passed a branch and perform Algorithm 5.2 in order to show the revisions that are available on the current branch to the user for loading. This will preserve the design history for not just the current branch, but all subsequent branches that are created.

4.2 Results

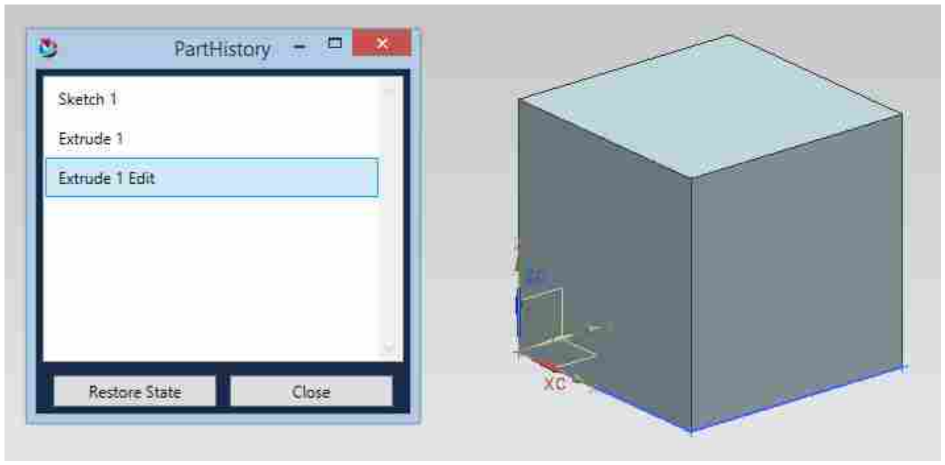
In order to test the validity of the loading method as well as the GUI, a simple part was made that contains a sketch, an extrude, and an edit to that extrude, as can be seen in Figure 4.5. The Part History window is shown in 4.5a, 4.5b, and 4.5c with the selected state being loaded in the image.



(a)



(b)



(c)

Figure 4.5: Revision History example showing the loaded revisions with the Part History Window shown with the loaded revision highlighted. In 4.5a the original sketch is loaded, in 4.5b the original extrude for the part is loaded and in 4.5c the entire part is loaded up to the most recent extrude edit. The Part History Window is the same window seen in Figure 4.3.

In order to capture the images in Figure 4.5, the entire part was made in NX and then closed without saving. The part then was loaded from the database and the GUI was opened from an NX ribbon plugin. The GUI contained the three states as desired and upon clicking on the state desired to load and then hitting the restore state button, Figures 4.5a, 4.5b, and 4.5c were taken as screen-shots. These images show that the loading worked in this instance and additional testing was performed to ensure that loading would work from any point in the revision history to any other point. It was also tested multiple times with sketches and different edits in the part to ensure a robust solution.

CHAPTER 5. CONFIGURATION MANAGEMENT

Another possibility that is opened up to the user with the changes to the database is the ability to return to any point in time or stop at any point and make a different configuration of the part. By doing so, the user makes a new branch of the part without having to create an entirely new part. However the changes to the database, detailed in Chapter 3 do not provide a GUI or loading methods to facilitate this change. The ability to create new configurations from within a part is not only a helpful feature for the user but is also beneficial in a database setup. The database allows for the new configuration to use the data from the old configuration so no data has to be duplicated.

The branching structure should allow for smaller storage size than within a PLM system. This is due to the elimination of redundant data storage created through copying parts. This allows the user to simply change the branch on which they are working and the design history will be preserved for both the first and the second branch without saving.

In order to do this, the user will be presented with a GUI that will help them create a branch. This GUI is accessed in the CAD system through the menu bar with a new button. When the user creates a branch they will no longer receive the changes made by other users who are not working on their branch, because essentially they are in a different part. The database will store which branch the user is in through a GUID and will only send the changes to the part that the user is currently in. Configuration management will be validated when:

- The user can create a new branch from within a part
- The user can load a branch without receiving any new data added to the parent branch from the time the new branch was created

5.1 Methodology

To create a user interface that allows the user to interact with the CAD file's configuration management, Microsoft Visual Studio is used to create and edit .xaml files in both a visual manner and directly in the code. The .xaml code references functions in the C# code files which then interact with the API. For this section, no changes need to be made to the database. These changes were taken care of by the changes addressed in Chapter 3. This is the same methodology described in section 4.1 but with a different display accessing different information from the database.

5.1.1 GUI

The two main features that need to be provided to the user in order to create a branching system are the ability to name and create a new branch as well as load a branch that has been created. These features can be seen in Figure 5.1. In order to select a branch that will act as the parent branch for a newly created branch, a user simply highlights the branch which they would like to have as their parent branch and then clicks the new branch button. Just as “Main” has been highlighted in Figure 5.1. To open any of the branches shown for the part a user simply selects the branch and then presses the load button.

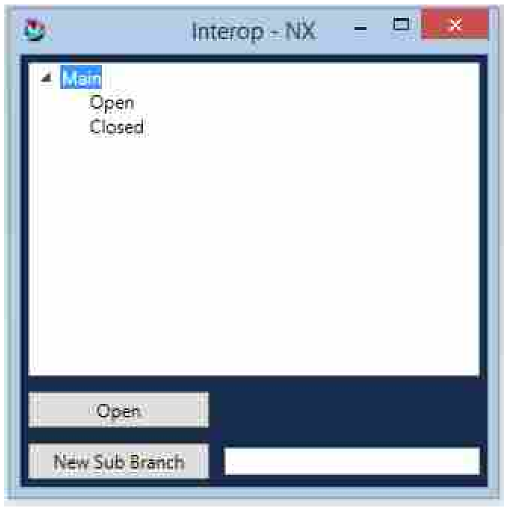


Figure 5.1: Image showing the GUI and a couple of branches for a part with a create branch button and a open button for loading a branch.

5.1.2 Part Loading and Branch Management

In order to load a branch that is selected, the GUI has to be able to communicate with the API of the local CAD system and the database. To do this the GUI passes a Branch object which can be seen in Figure 3.1. This table contains a GUID, a Name which is a string, a time-stamp of when the branch was created, and a reference to a parent branch, not shown in Figure 3.1. This data allows the Interop C# program to collect the correct branch state data from the database and use the CAD system's API to create the features.

The default behavior of Interop is to grab all of the features from the database for loading. The server always sends all of the features for the current part as messages on "part open". In order to recognize which feature states and which features to load, the algorithm, seen in Figure 5.2, was developed to remove unwanted states and features. The first FOR loop filters out unwanted features from the part by checking to see if any of the states for that feature exist on a parent branch or the current branch being loaded. The second FOR loop tells the CAD system which states to load for each feature that remains.

This loading algorithm, while only shown in pseudo-code form in Figure 5.2, can be seen fully in Appendix C. It works recursively to check every parent branch up to the main branch of the part. It does this because if any feature was created on a parent branch but not loaded on a child branch there is a chance that the part would fail to load because the correct parent feature that has the data needed for a child feature may not be there once loaded. For instance if a sketch was created on the main branch, then a new branch was created, and an extrude was made off of the sketch which exists in the part. On load, if that sketch did not load, when the system attempts to apply the extrude, the part would fail to load the reference to the non-existant sketch and part loading would fail.

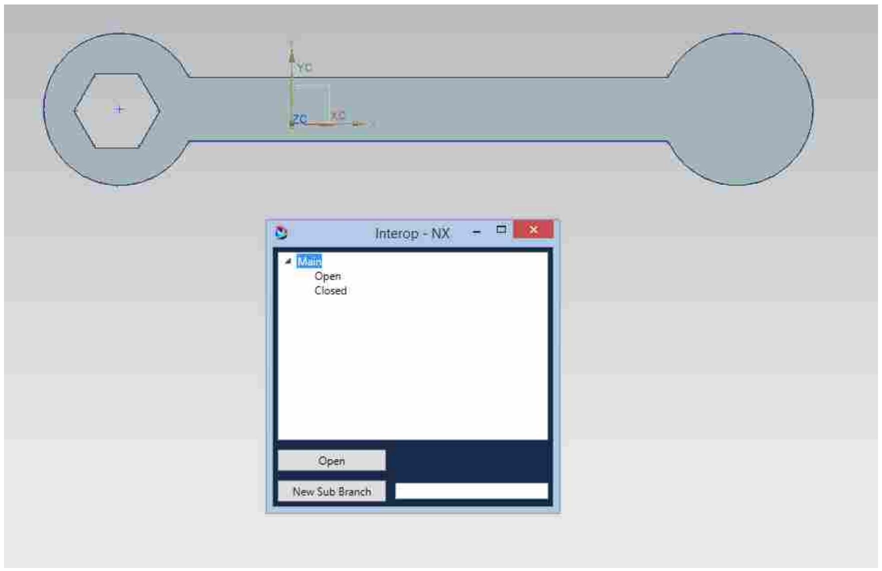
Although this algorithm is more complicated than it needs to be because the server sends more information than is needed at this time. The algorithm is in place to remedy the server implementation and it does work well for the situation of receiving all of the feature data from the server. In order for the algorithm to be improved it would have to be implemented directly into the server code where the client passes the server a Branch GUID that it desires and the server communicates with the database, gathering only the necessary branch states and parent states. The server then packages that data and sends a message back containing only the features and states relevant to the desired branch. This improved method was not implemented because altering the server code was outside of the scope of this thesis.

```
Data: All features in the part from every branch
Result: The current state of features on the branch to be loaded
for Each Feature do
    for Each Feature State Contained in the Feature do
        if (State Timestamp < Branch Timestamp) && (Is on a parent branch) then
            Keep the feature to which the state belongs
        else
            Only keep features that contain states on the current branch
        end
    end
end
for Remaining Features After Feature Removal do
    if Feature has no states on the current branch then
        Load the most recent state before the current branch was created that exists on a parent
          branch
    else
        Load the most recent state
    end
end
```
Figure 5.2: Loading algorithm for a branch.

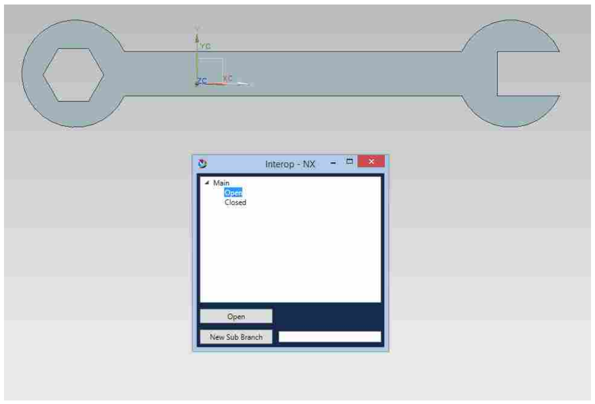
5.2 Results

5.2.1 Branching Tests

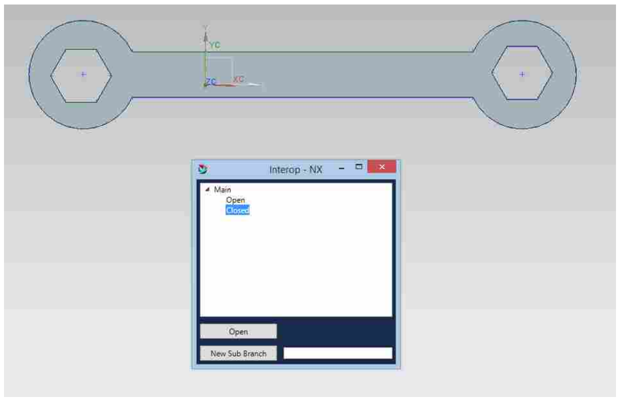
To check the branching implementation of branching, two users worked together to make two different wrenches. The wrenches were to have the same hex opening on one side of the wrench and have two different configurations where the other end was open in one, and had a different hex opening in the other. The results of this test can be seen in Figure 5.3d. One user created a sketch which had the profile of the wrench and then extruded the wrench to the proper height. The main branch which can be seen in Figure 5.3a has this configuration. At this point one user created a new branch for the open faced wrench and the second user created another branch for a closed faced wrench. They then both proceeded to create the open faced wrench which can be seen in Figure 5.3b and the closed faced wrench which can be seen in Figure 5.3c. Neither user received the edits from the other user unless they had same branch loaded. The users were able to work in the same part on different configurations.



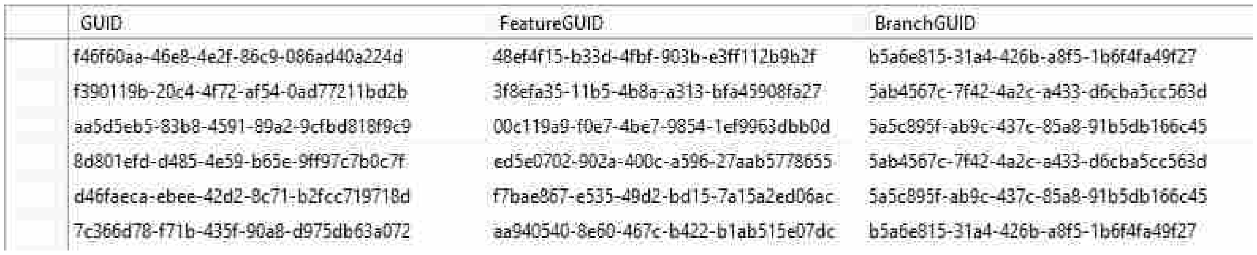
(a) Main branch with sketch and extrude providing the groundwork for the Open and Closed Branches.



(b) Open branch with open faced wrench configuration.



(c) Closed Branch with closed end wrench configuration.



GUID	FeatureGUID	BranchGUID
f46f60aa-46e8-4e2f-86c9-086ad40a224d	48ef4f15-b33d-4fbf-903b-e3ff112b9b2f	b5a6e815-31a4-426b-a8f5-1b6f4fa49f27
f390119b-20c4-4f72-af54-0ad77211bd2b	3f8efa35-11b5-4b8a-a313-bfa45908fa27	5ab4567c-7f42-4a2c-a433-d6cba5cc563d
aa5d5eb5-83b8-4591-89a2-9cfbd818f9c9	00c119a9-f0e7-4be7-9854-1ef9963dbb0d	5a5c895f-ab9c-437c-85a8-91b5db166c45
8d801efd-d485-4e59-b65e-9ff97c7b0c7f	ed5e0702-902a-400c-a596-27aab5778655	5ab4567c-7f42-4a2c-a433-d6cba5cc563d
d46faeca-ebee-42d2-8c71-b2fcc719718d	f7bae867-e535-49d2-bd15-7a15a2ed06ac	5a5c895f-ab9c-437c-85a8-91b5db166c45
7c366d78-f71b-435f-90a8-d975db63a072	aa940540-8e60-467c-b422-b1ab515e07dc	b5a6e815-31a4-426b-a8f5-1b6f4fa49f27

(d) Six feature states with their associated features and branches.

Figure 5.3: Two branches are displayed under a parent branch to create two different wrench configurations. State branch data can be seen in 5.3d which contains all of the states for the different features that make up Figures 5.3a, 5.3b, and 5.3c.

To ensure that the part data was being sent to the server correctly a SQL query was run in Microsoft Visual Studio. The result of the query can be seen in Figure 5.3d. The query simply looked for sketch and extrude states which contain a Branch GUID and exist in the current part. The list in Figure 5.3d shows that there are six states that have a Branch GUID and if the GUIDs are examined we can see that there are three different branches with two features on each. The GUID tells us this information because each GUID is a globaly unique identifier and when the GUID is seen more than one time it represents the same object. This makes sense as each branch contains one sketch and one extrude.

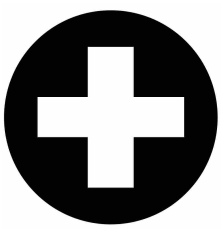
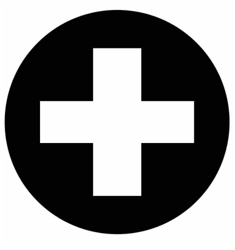
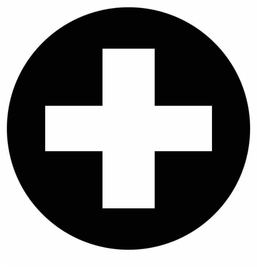
The results from this test help to conclude that users are able to work on the same part in a MESH CAD environment. Users are also able to create new branches and to work on different branches simultaneously without receiving changes to other branches.

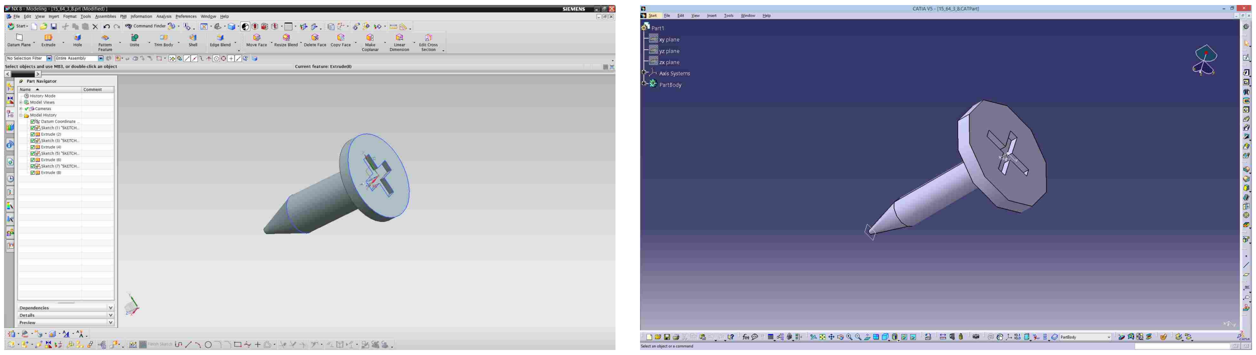
5.2.2 File Size

It was theorized that the database structure of configuration management as well as the ability to perform configuration management on a feature level would reduce storage size. In order to test this theory a test example was designed involving an array of screws. In large assemblies screws are not always present because of the increased loading times presented by all of the fasteners and are typically called out in drawings only [23]. This example helps illustrate that the data stored can be reduced across an array of parts that change only slightly from one part to the next, as well as the storage size needed to create translational STEP and or IGES files.

In order to make more than one screw in a typical CAD system the user would have to create each screw in a separate part and then save those parts. Due to a limit in the NPDB feature set, the screw threads have not been made in the CAD models. Table 5.1 shows all of the configurations of screws that will be made in each CAD system as well as the dimensions that will be changed in each. The only dimensions that will be changed are the head bore size and the length of the screw. The screw diameter was not specified, so in Figure 5.4 there is a slight difference in the screw diameter during the test in single user CAD between 5.4a and 5.4b. This small change may account for some disparity in file size but the difference is negligible. The CAD model for each of the six screw configurations in Table 5.1 were made in NX, CATIA and CREO and a basic example can be seen in Figure 5.4. These CAD files have all been saved and their respective CAD file sizes have been recorded for each system and can be seen in Table 5.2.

Table 5.1: Configurations of wood screws with Phillips-Head Point Size #1 used for testing configuration management.

	Head Bore Size		
			
Lengths	11/64”	13/64”	15/64”
1/4”	Configuration 1	Configuration 3	
3/8”			Configuration 5
1/2”	Configuration 2		
5/8”		Configuration 4	
3/4”			Configuration 6



(a) (b)

Figure 5.4: Screws created in NX and CATIA for the configuration of 15/64” head bore size and length of 3/8”.

These same screws were made in Interop with the new feature-based configuration management. The process for this was slightly different than making the screw for each CAD system. The part was designed using features that are the same in each part first. With the selected parts, they each have a Phillips-Head Point Size of #1 for typical wood screws [24]. The cross in the head where the screw driver is inserted is the same for each screw so that feature can be designed without having to be changed from one screw to the next. There is also a circle that is extruded upwards for each screw by the same amount for each and can be designed in the base part. The same can be said for the point of the screw and the circle that is extruded for the length of the screw. None of these features need to change from one screw to the next. The only features that need to change are the length of screw and the diameter of the head. The dimensions for the lengths and head diameters can be seen in Table 5.1.

To collect the equivalent file size for the screws that were saved in the database, the entire database had been cleared before the creation of these parts. The database then was saved using a script generation process that produces all of the data to reproduce the database in its entirety while the database tables had no data in them. Test users created the parts. After the creation of the parts the entire database was saved again with the database tables full of the data needed to create the CAD files. This was done to compare the file sizes of the parts created in the NPDB with those of the CAD systems individually. To ensure a fair comparison of the part files and the database, blank part files were saved for each CAD system as well as the empty database. These file sizes were then subtracted from the saved part files and the saved blank database was subtracted from the full database resulting in the data needed to just save the part data. This data has been put into a visual format in Table 5.2 with the comparative size being the save size of the completed part file minus the save size of an empty part file.

Table 5.2: Data sizes in KB for different CAD parts, Comparative Size Column removes the size of a blank part if applicable.

System	Blank	Part	IGES	STEP	Comparison
NX	82	213	79	25	131
CATIA	42	123	70	35	81
CREO	51	153	-	-	102
Sum	-	-	-	-	314
Interop	80	88	-	-	8

From Table 5.2 if the Interop comparison size of 8 KB is subtracted from the comparison of each CAD system, the database reduced the total file size by 123 KB from NX, by 73 KB from

CATIA, as well as 94 KB from CREO. These file size reductions are much larger than what was anticipated, but another benefit that was not considered before this study is shown in Table 5.2. Not only did the database structure reduce the file size from the standard CAD part files, it also reduced the translational part size of both IGES and STEP files while maintaining more data in translation through the feature tree. In addition to reducing file sizes for each CAD system, the neutral parametric form is not only the part file for one CAD system, but contains the data for each CAD system. So if the part files for CATIA, NX, and CREO were added together to create a file group that added to 314 KB, the database file would still be 8 KB. This results in a 39.25 times decrease in size.

Table 5.2 shows the comparison of just one part in each CAD system to that in the database. The screw example uses the new feature-level configuration management to create all of the different configurations of screws referenced in Table 5.1 without duplicating data. This leads to a database size of 94 KB, where for each of the other CAD systems the part files would have to be duplicated, leading to multiplying the storage size for the different configurations to be the part file size multiplied by six. The database only has to add one or two states for each new configuration, and as such only increases by 6 KB to 94 KB, which is only a 6.8% increase in size. The decrease in storage presented by feature level configuration management is 98.6% less than what is currently required by copying parts.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

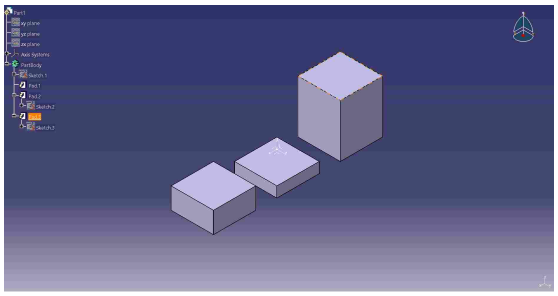
6.1.1 Database

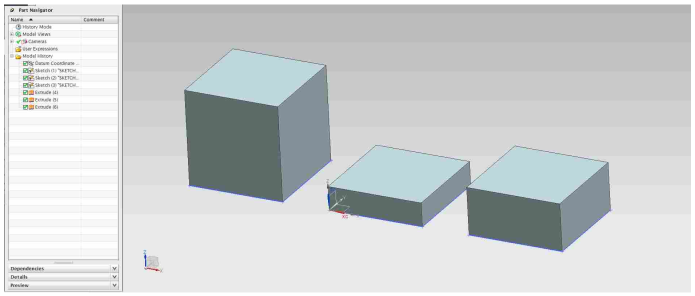
The following conditions validated the new database structure:

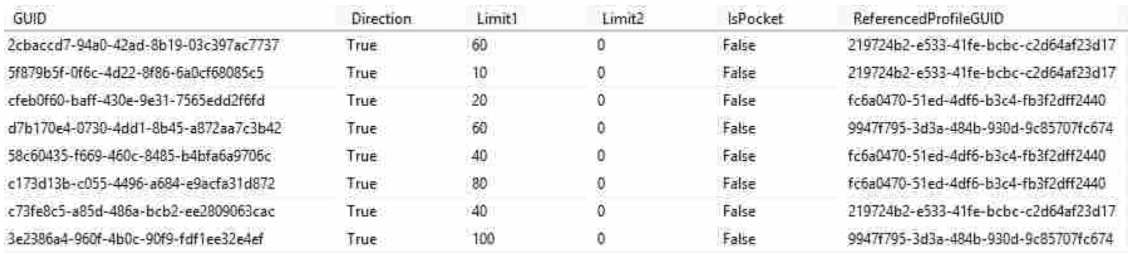
- A created feature adds a new object to the database with a new GUID and a state with a separate GUID
- An edited feature adds a new state to the existing object with a new GUID
- A second user can receive both creation and edits of features without duplication in the CAD file
- Upon part load, only the most recent state of each feature is loaded

The results in section 3.2 show that the database correctly added a new object to the database when a feature is created as well as a state on create or edit. These results are seen in Figure 6.1c. This figure also shows the data needed to validate that an edited feature creates a new state with a new GUID and GUID reference to its parent feature. This demonstrates that data is being saved correctly in the database and it proves that the loading methods provided in Interop are able to view and manipulate the models.

Two users can work in the same part at the same time. The multi-user aspect of Interop was maintained. This was demonstrated in NX for all Interop features and in CATIA and CREO for Extrudes. The results of testing these features can be seen in Figures 6.1a and 6.1b. This proves that users can still access the data from the database and using the algorithms for loading and receiving data from other users in Interop, work within a MESH CAD environment.


(a)


(b)



GUID	Direction	Limit1	Limit2	IsPocket	ReferencedProfileGUID
2cbaccd7-94a0-42ad-8b19-03c397ac7737	True	60	0	False	219724b2-e533-41fe-bcbc-c2d64af23d17
5f879b5f-0f6c-4d22-8f86-6a0cf68085c5	True	10	0	False	219724b2-e533-41fe-bcbc-c2d64af23d17
cfeb0f60-baff-430e-9e31-7565edd2f6fd	True	20	0	False	fc6a0470-51ed-4df6-b3c4-fb3f2dff2440
d7b170e4-0730-4dd1-8b45-a872aa7c3b42	True	60	0	False	9947f795-3d3a-484b-930d-9c85707fc674
58c60435-f669-460c-8485-b4bfa6a9706c	True	40	0	False	fc6a0470-51ed-4df6-b3c4-fb3f2dff2440
c173d13b-c055-4496-a684-e9acfa31d872	True	80	0	False	fc6a0470-51ed-4df6-b3c4-fb3f2dff2440
c73fe8c5-a85d-486a-bcb2-ee2809063cac	True	40	0	False	219724b2-e533-41fe-bcbc-c2d64af23d17
3e2386a4-960f-4b0c-90f9-fdf1ee32e4ef	True	100	0	False	9947f795-3d3a-484b-930d-9c85707fc674

(c)

Figure 6.1: Final state of extrudes shown in both the CATIA (6.1a) and NX (6.1b) CAD clients. Database table data (6.1c) for extrudes shown in 6.1a and 6.1b as well as all edit data. Views are not from the same viewing position but the CAD parts are the same.

D2 stores each state of the part as well as loads the most recent state, and preserves the entire design history. Users are able to work together in multiple CAD environments and editing is done synchronously. The experiment conditions for D2 have been met and have been validated through the example provided in Figure 6.1. These changes also create the foundation that allowed for revision history and configuration management.

6.1.2 Revision History

The following two conditions validated revision history:

- A user can view previous states of the part in a feature list to select which to load.
- A user can select a previous state of the part and load the part at that state.

To prove that a user can view previous states of a part, a part was created that contained a sketch, an extrude, and an extrude edit. A user selected the Part History Window and views the list of all data in the part since creation, as can be seen in Figure 6.2. Figures 6.2a, 6.2b, and 6.2c show a user selecting and loading each state of the part since part creation. Both of the conditions set for revision history validation are met in the series of images presented in Figure 6.2. The algorithm used to perform the loading can be seen in Algorithm 1. These conditions allowed a user to view all previous edits and creates made to a part, which preserves design intent and provides the groundwork for configuration management.

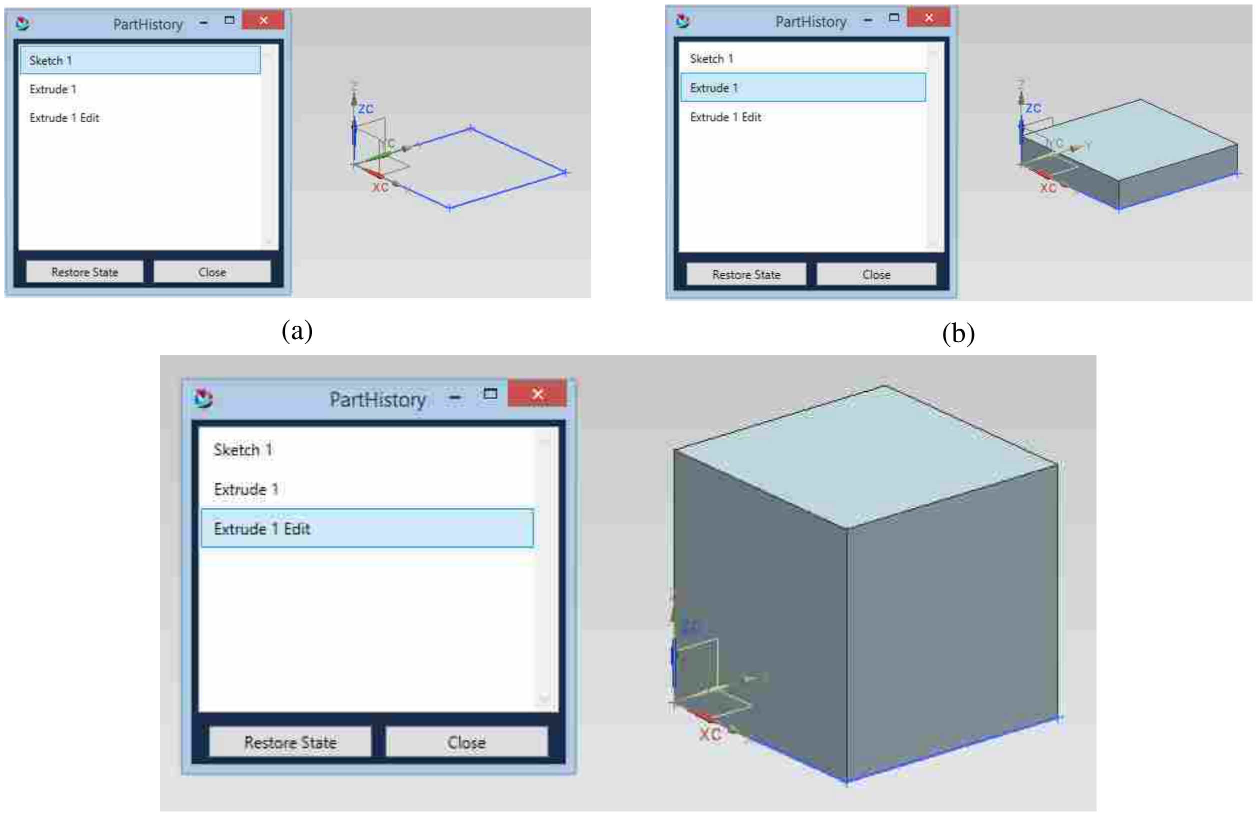


Figure 6.2: Revision History example showing the loaded revisions with the Part History Window shown with the loaded revision highlighted. In 6.2a the original sketch is loaded, in 6.2b the original extrude for the part is loaded and in 6.2c the entire part is loaded up to the most recent extrude edit. The Part History Window is the same window seen in Figure 4.3.

6.1.3 Configuration Management

These conditions confirmed configuration management:

- The user can create a new branch from within a part.

- The user can load a branch without receiving any new data added to the parent branch from the time the new branch was created.

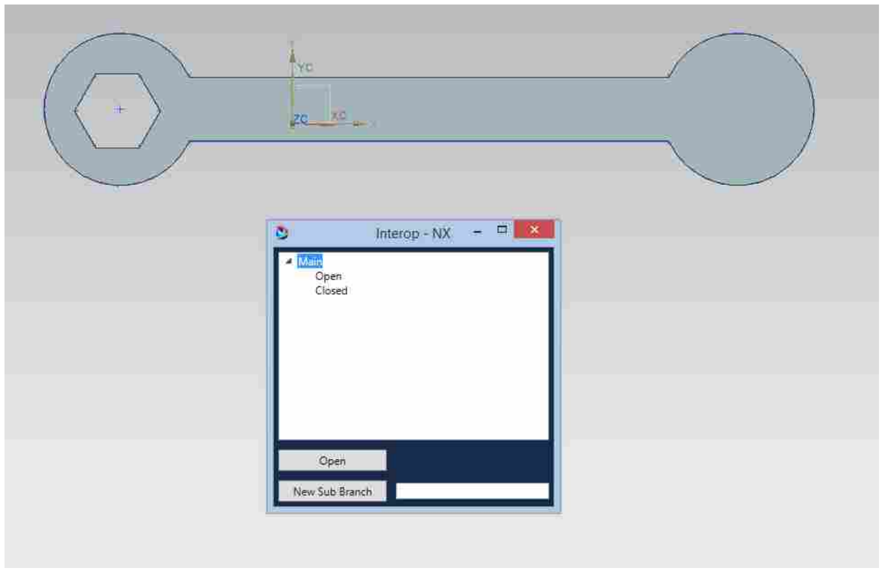
Configuration Management was validated by the experiment where two users created two different versions of a wrench. This experiment can be seen in Figure 6.3, where the main branch in Figure 6.3a contains the base configuration used to create the child branches seen in Figures 6.3b and 6.3c. This test confirmed that users can create a branch from within a part and that edits made in one branch are not applied in another part. It also confirmed that users are able to work in the MESH CAD environment on the same part at the same time if they have the same branch loaded.

Another benefit seen from a different test with screws helped to show how feature level database based configuration management reduces data storage size. The data from this study can be seen in Table 5.2 and shows that for a simple configuration example of screws, the database approach was able to reduce part save sizes by 98.6%.

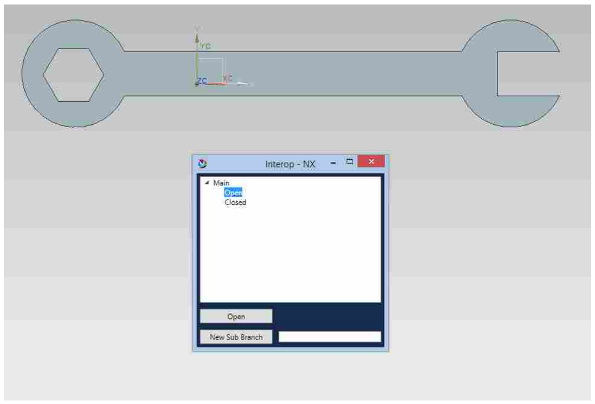
All of the changes made to the Interop environment have been CAD system independent. The majority of the changes took place in the reorganization of the database to support the storage of additional information. The remainder of the changes took place in creating a GUI and then implementing the major changes in the NX plugin for Interop. The NX plugin changes are the only part of this research that can not work with other CAD systems, but the format of how to implement features remains the same. For a full treatment of how to create new features in the system see Appendix E.

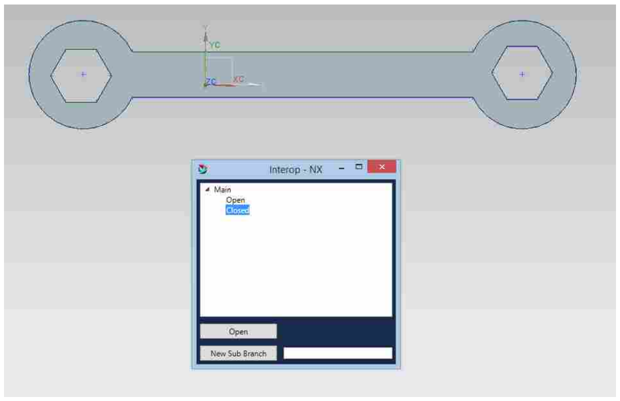
6.2 Further Work

While MESH CAD and the NPCF have been developed to help increase design transparency across a heterogeneous CAD environment, these solutions are still in the early stages of development and lack full connection into PLM systems which are necessary to the design workflow. In order to better test and develop these systems to help address issues, such as the cost of translation of CAD file data, there is a need to allow interaction with features that users are accustomed to having in a PLM system. In a multi-user environment, it becomes increasingly important to know what other designers were intending to do while creating a part and allowing a user to

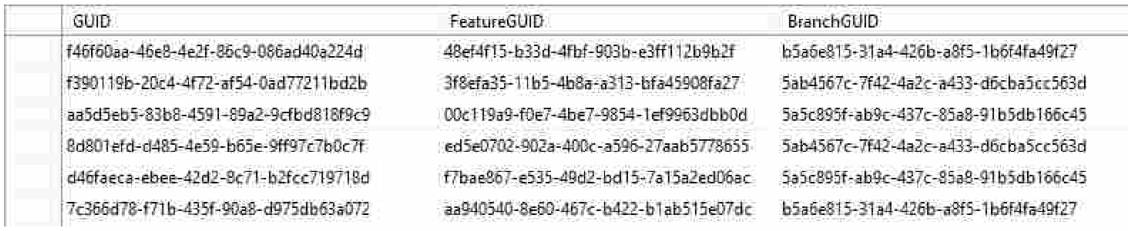


(a)





(b) (c)



	GUID	FeatureGUID	BranchGUID
	f46f60aa-46e8-4e2f-86c9-086ad40a224d	48ef4f15-b33d-4fbf-903b-e3ff112b9b2f	b5a6e815-31a4-426b-a8f5-1b6f4fa49f27
	f390119b-20c4-4f72-af54-0ad77211bd2b	3f8efa35-11b5-4b8a-a313-bfa45908fa27	5ab4567c-7f42-4a2c-a433-d6cba5cc563d
	aa5d5eb5-83b8-4591-89a2-9cfbd818f9c9	00c119a9-f0e7-4be7-9854-1ef9963dbb0d	5a5c895f-ab9c-437c-85a8-91b5db166c45
	8d801efd-d485-4e59-b65e-9ff97c7b0c7f	ed5e0702-902a-400c-a596-27aab5778655	5ab4567c-7f42-4a2c-a433-d6cba5cc563d
	d46faeca-ebee-42d2-8c71-b2fcc719718d	f7bae867-e535-49d2-bd15-7a15a2ed06ac	5a5c895f-ab9c-437c-85a8-91b5db166c45
	7c366d78-f71b-435f-90a8-d975db63a072	aa940540-8e60-467c-b422-b1ab515e07dc	b5a6e815-31a4-426b-a8f5-1b6f4fa49f27

(d)

Figure 6.3: Two branches under a parent branch to create two different wrench configurations. State branch data can be seen in 6.3d which contains all of the states for the different features that make up Figures 6.3a, 6.3b, and 6.3c.

see each change made to the part in correct chronological order thereby moving toward greater visibility between designers. Keeping the database in a state in which referential integrity is maintained helps keep the data passed between designers valid and allows for greater trust in models that are being developed by multiple designers at the same time. Configuration management tools allow users to select features needed for a new design without having to copy data or duplicate work, which allows for quicker modeling practices and a faster turnaround time during the design process. These features add to the validity of the multi-user environment and the NPCF.

PLM systems manage great amounts of data and the changes made to the NPDB only account for a very select feature set managed by PLM. These features however increase the capabilities available to a multi-user environment and the small feature set allows for them to be implemented securely and accurately. Future work will allow for greater control over CAD data in any system and a greater decrease for the time and money required currently for translation.

These changes are important because they help to integrate PLM features into the CAD system giving more power to the designers and allowing for more collaboration. With increased collaboration, it is believed that there will be fewer turn backs in the design process due to errors and more innovation due to the ability to collaborate with other users in real time.

Another benefit to this system that has not been implemented but is a possibility is the fast loading of different configurations. Load times for large assemblies can be extremely long [23]. In some situations a different CAD assembly is used to track the repair and management of different parts in the real world [25]. In these cases the base CAD assembly is the same across all the real world assemblies. When a repair is made changes are made to the CAD assembly which represents the real world part. In this system a different branch of the main part would be created and the edits would take place on that branch. This allows for a unique opportunity for switching between different branches without reloading the entire assembly. This opportunity is not implemented in the current state of the NPDB configuration management loading but is a possibility. The only loading that would need to take place in the changing of the assemblies would be the changes that have been made between the branches. Long load times for complex parts could be reduced dramatically by allowing the CAD system to only load the parts that are needed for the changes between the assemblies.

An additional benefit to the NPCF is that every object in the system is parameterized automatically by storing the data in a field within a database table. That table data is automatically a parameter that can be altered in the database, which allows automation within a PLM system without setting up elaborate parametric constraints or learning multiple CAD system APIs [26]. For this application, an optimization program could write different configurations of a part by changing the variables in the database which creates a new state and all the different configurations could be saved for an optimization run without creating the parts physically.

6.2.1 Thesis Overview

The foregoing has demonstrated and validated four key objectives:

1. The database had been expanded to store all revisions of a part.
2. A user interacted with D2 information to view previous part states.
3. A user made and loaded different configurations of a part through branching.
4. The multi-user and multi-CAD package aspects of Interop were maintained.

Figure 6.4 shows how the database was expanded to D2. From Figures 6.1, 6.2, and 6.3 it is seen that the user was able to interact with D2. Figure 6.3 also demonstrates how a user was able to create new branches and switch to them as well as showing that the multi-user aspect of Interop was maintained.

Table 6.1: Comparison with HSTDB and current methods
to show the conclusions of the research.

Translation Comparison	IGES	STEP	NPCF	HSTDB
Part Geometry	✓	✓	✓	✓
Feature Data	X	X	✓	✓
Revision History	X	X	X	✓
Configuration Management	X	X	X	✓

This work involved restructuring the Interop database for revision history and configuration management. I implemented a full NX plugin to communicate with this new database structure.

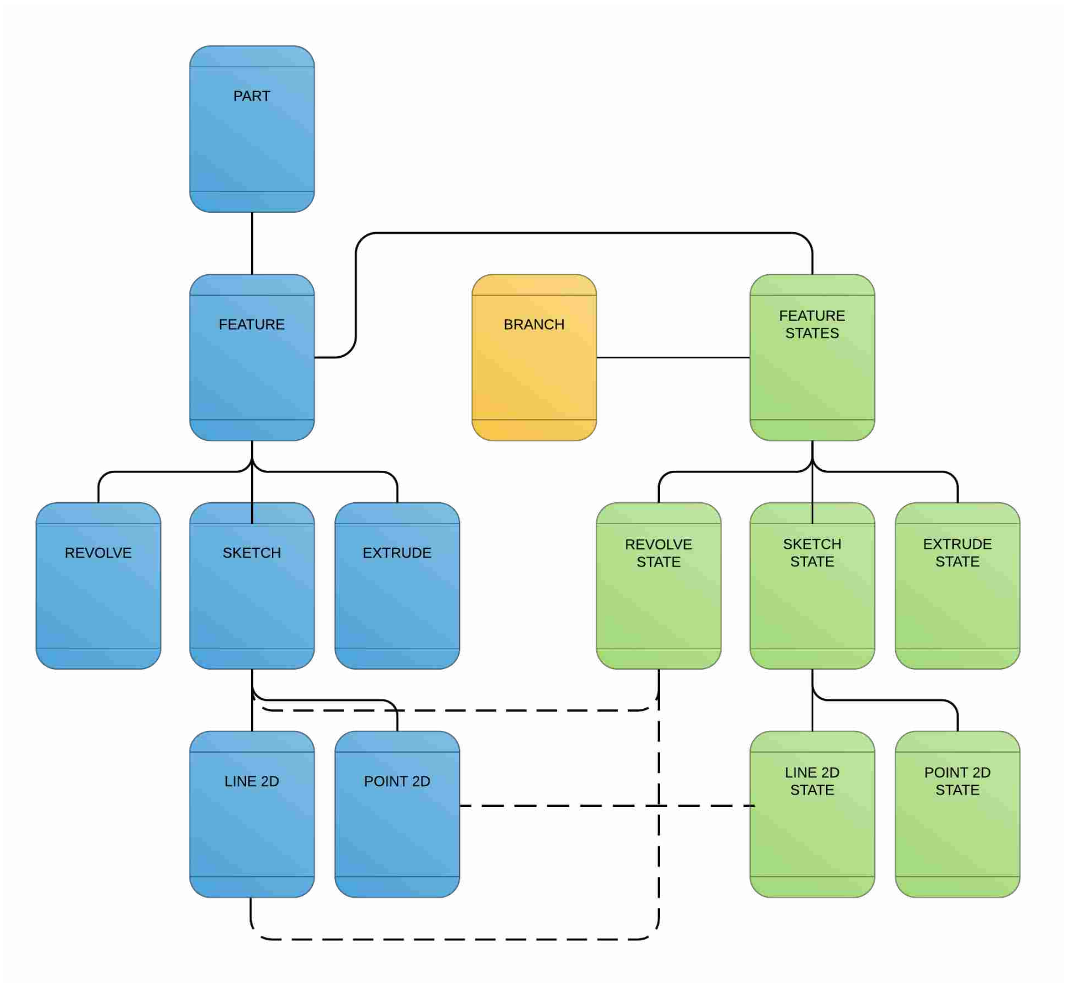


Figure 6.4: D2 showing addition of states tables as well as the branch table with some relationships shown.

To prove that the system was not limited to NX I implemented the extrude feature in NX, CATIA and CREO. Then in NX I added a GUI for the user to interact with the information. I maintained the multi-user aspect of Interop and maintained state based loading.

In conclusion I have shown that I have completed the objectives outlined in the introduction to this research to provide an improvement to a new translation process that maintains revision history and allows for configuration management as seen in Table 6.1. The new Hybrid State Transactional Database (D2) is able to reduce file save sizes and provide more information to users. PLM features have been incorporated into the MESH CAD environment and greater control is available to users.

REFERENCES

[1] Olive−Drab.com, 2017. Fmtv: Family of medium tactical vehicles Webpage, 1. vii, 15, 16

[2] Brunnermeier, S., and Martin, S., 1999. Interoperability Cost Analysis of the U.S. Automotive Supply Chain Tech. Rep. 7007. 1

[3] Leach, L. M., 1983. "Language interface for data exchange between heterogeneous cad/cam databases.." *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.],*, **44**(5). 1

[4] Pratt, M. J., 1998. "Extension of the standard iso10303 (step) for the exchange of parametric and variational cad models." *CDROM Proceedings of the Tenth International IFIP WG,* **5**(5.3). 1

[5] Haenisch, J., 1990. "Cad-exchange-towards a first step implementation." In *Industrial Electronics Society, 1990. IECON'90., 16th Annual Conference of IEEE*, IEEE, pp. 734–739. 1

[6] Freeman, R. S., Bowman, K. E., Red, E., and Staves, D. R., 2015. "Neutral parametric canonical form for 2d and 3d wireframe cad geometry." In *ASME 2015 International Mechanical Engineering Congress and Exposition*, American Society of Mechanical Engineers, pp. V011T14A004–V011T14A004. 2, 12, 16

[7] Li, M., Gao, S., and Wang, C. C. L., 2007. "Real-Time Collaborative Design With Heterogeneous CAD Systems Based on Neutral Modeling Commands." *Journal of Computing and Information Science in Engineering,* **7**(2), p. 113. 2

[8] Sung, C., and Park, S. J., 2007. "A component-based product data management system." *The International Journal of Advanced Manufacturing Technology,* **33**(5-6), pp. 614–626. 8

[9] Phelan, J., 2012. Siemens jt data format accepted as the worlds first iso international standard for viewing and sharing lightweight 3d product information Website, 12 . 8

[10] Red, E., Jensen, C., Ryskamp, J., and Mix, K., 2010. "Nxconnect: Multi-user cax on a commercial engineering software application." In *PACE Glob Annu Forum*, pp. 1–9. 8

[11] Briggs, J. C., Hepworth, A. I., Stone, B. R., Coburn, J. Q., Jensen, C. G., and Red, E., 2015. "Integrated, synchronous multi-user design and analysis." *Journal of Computing and Information Science in Engineering,* **15**(3), p. 031002. 8

[12] Freeman, R. S., 2015. "Neutral Parametric Canonical Form for 2D and 3D Wireframe CAD Geometry." Thesis, BYU. 10

[13] Qiang, L., Zhang, Y., and Nee, A., 2001. “A distributive and collaborative concurrent product design system through the www/internet.” *The International Journal of Advanced Manufacturing Technology,* **17**(5), pp. 315–322. 10

[14] Ramani, K., Agrawal, A., Babu, M., and Hoffmann, C., 2003. “Caddac: Multi-client collaborative shape design system with server-based geometry kernel.” *Journal of Computing and Information Science in Engineering,* **3**(2), pp. 170–173. 10

[15] Li, M., Gao, S., and Wang, C. C., 2007. “Real-time collaborative design with heterogeneous cad systems based on neutral modeling commands.” *Journal of Computing and Information Science in Engineering,* **7**(2), pp. 113–125. 12

[16] Li, M., Gao, S., Li, J., and Yang, Y., 2004. “An approach to supporting synchronized collaborative design within heterogeneous cad systems.” In *ASME 2004 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, American Society of Mechanical Engineers, pp. 511–519. 12

[17] Li, M., Yang, Y., Li, J., and Gao, S., 2004. “A preliminary study on synchronized collaborative design based on heterogeneous cad systems.” In *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*, Vol. 1, IEEE, pp. 255–260. 12

[18] Bowman, K. E., and Shumway, D. “Pseudo-singleton pattern and agnostic business layer for multi-engineer, synchronous, heterogeneous cad.” *Computer-Aided Design and Applications,* **In Press**. 17

[19] Markowitz, V. M., 1991. Safe referential integrity structures in relational databases Tech. rep., Lawrence Berkeley Lab., CA (USA). 17

[20] Loeliger, J., and McCullough, M., 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development.* ” O’Reilly Media, Inc.”. 27

[21] Han, S., 2010. “Macro-parametric: an approach for the history-based parametrics.” *International Journal of Product Lifecycle Management,* **4**(4), pp. 321–325. 32

[22] Hepworth, A. I., Tew, K., Nysetvold, T., Bennett, M., and Greg Jensen, C., 2014. “Automated conflict avoidance in multi-user cad.” *Computer-Aided Design and Applications,* **11**(2), pp. 141–152. 32

[23] Bowland, N., Gao, J., and Sharma, R., 2003. “A pdm-and cad-integrated assembly modelling environment for manufacturing planning.” *Journal of materials processing technology,* **138**(1), pp. 82–88. 48, 57

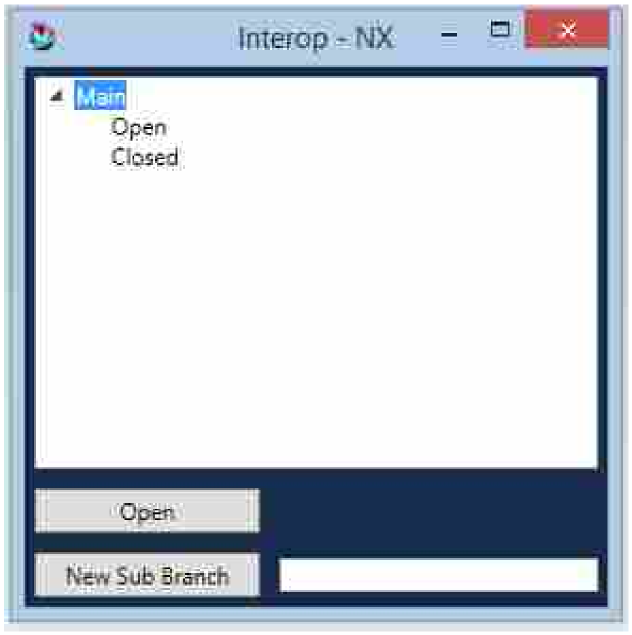
[24] fandfindustrial.com, 2017. Screw-sizes Webpage, 2. 50

[25] Philpotts, M., 1996. “An introduction to the concepts, benefits and terminology of product data management.” *Industrial Management & Data Systems,* **96**(4), pp. 11–17. 57

[26] Gomes, S., Varret, A., Bluntzer, J., and Sagot, J., 2009. “Functional design and optimisation of parametric CAD models in a knowledge-based PLM environment.” *International Journal of Product Development,* **9**(1/2/3), pp. 60 – 77. 58

APPENDIX A. WINDOW CODE IN XAML

A.1 Branching Window



```
<Window x:Class="CADInteropNX.BranchesWindow"
          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          Title="Branches" Height="325" Width="325"
          xmlns:connectData="clr-namespace:ConnectData;assembly=ConnectData">
    <Window.Resources>
        <HierarchicalDataTemplate DataType="{x:Type connectData:DBBranch}" ItemsSource="{Binding
    ChildBranches}">
            <TextBlock Text="{Binding Name}"/>
        </HierarchicalDataTemplate>
    </Window.Resources>
    <Grid DockPanel.Dock="Bottom" x:Name="LayoutRoot" Background="#172a4a">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="auto"/>
```

```
                <RowDefinition Height="auto"/>
                <RowDefinition Height="auto"/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <TreeView DockPanel.Dock="Bottom" x:Name="tvAssemblies" Margin="5" MouseDoubleClick="
        Tree_DoubleClick"/>
            <Button Grid.Row="2" Width="120" Content="Open" Click="open_Click" Padding="3" Margin="5"
         HorizontalAlignment="Left"/>
            <DockPanel Grid.Row="3">
                <Button DockPanel.Dock="Left" Width="120" Content="New Sub Branch" Click="
        NewBranch_Click" Padding="3" Margin="5" />
                <TextBox x:Name="txtSubName" VerticalAlignment="Center" Margin="5"/>
            </DockPanel>
        </Grid>
</Window>
```

Listing A.1: Branches Window Code (XAML)

A.2 Branches Window Code C#

```
using ConnectData;
using ConnectData.Messages;
using NXOpen;
using NXOpen.UF;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Threading;
using System.Windows;
using System.Windows.Documents;
using System.Windows.Input;
using ConnectData.MUObjects;

namespace CADInteropNX
{
    /// <summary>
    /// Interaction logic for Branches.xaml
    /// </summary>
    public partial class BranchesWindow : Window
    {
        public ObservableCollection<ConnectData.DBBranch> InstanceList { get; private set; }
        ClientConnection connection;
        MainWindow mainWindow;

        public BranchesWindow(ClientConnection incomingConnection, MainWindow sendingMainWindow)
        {
            InitializeComponent();

            connection = incomingConnection;
            mainWindow = sendingMainWindow;

            refreshInstanceList();
            tvAssemblies.ItemsSource = InstanceList;

        }

        public void refreshInstanceList()
        {
            NXOpen.Part workPart = Utilities.NXSession.Parts.Work;
            MUPart workPartMU = MUPart.GetInstanceFromNX(workPart as NXOpen.Part);
            DBPart serverPart = workPartMU.ServerPart;

            FeatureListMessage featureListMsg = (FeatureListMessage)connection.sendReceive(new
    WatchPartMessage("NX", serverPart.GUID));

            List<DBFeature> features = featureListMsg.FeatureList.OrderBy(x => x.TreeOrder).
    ToList();

            List<DBBranch> branches = new List<DBBranch>();

            foreach (DBFeature feature in features)
            {
                foreach (DBInteropState state in feature.DBInteropStates)
                {
                    if (!branches.Contains(state.DBBranch) && state.DBBranch != null && state.
    DBBranch.ParentBranch == null)
                    {
                        branches.Add(state.DBBranch);
                        InstanceList = new ObservableCollection<ConnectData.DBBranch>(branches);
                        return;
                    }
                }
            }
        }

        private ObservableCollection<ConnectData.DBBranch> alphabetizeChildren(ConnectData.
    DBBranch parent)
        {
            foreach (ConnectData.DBBranch child in parent.ChildBranches)
            {
                if (child.ChildBranches.Count > 0)
                {
                    child.ChildBranches = alphabetizeChildren(child);
                }
            }

            return new ObservableCollection<ConnectData.DBBranch>(parent.ChildBranches.OrderBy(y
    => y.Name).ToList());
        }

        private void Tree_DoubleClick(object sender, MouseButtonEventArgs e)
        {
            OpenRevision();
        }

        private void open_Click(object sender, RoutedEventArgs e)
        {
            OpenRevision();
        }

        private void close_Click(object sender, RoutedEventArgs e)
        {

        }

        private void NewBranch_Click(object sender, RoutedEventArgs e)
        {
            DBBranch selectedBranch = tvAssemblies.SelectedItem as DBBranch;

            if (selectedBranch == null)
                return;

            if (selectedBranch.ChildBranches!= null && selectedBranch.ChildBranches.Any(x => x.
    Name == txtSubName.Text))
            {
                MessageBox.Show("A branch with that name already exists in this part.");
                return;
            }
            if (selectedBranch.Name == txtSubName.Text)
            {
                MessageBox.Show("Cannot name a branch the same name as its parent branch.");
                return;
            }

            if (!string.IsNullOrEmpty(txtSubName.Text))
            {
                DBBranch newBranch = new DBBranch()
                {
                    Name = txtSubName.Text,
                    ParentGUID = selectedBranch.GUID,
                    GUID = Guid.NewGuid(),
                    TimeStamp = DateTime.Now
                };

                newBranch.ParentBranch = selectedBranch;

                if (selectedBranch.ChildBranches == null)
                    selectedBranch.ChildBranches = new ObservableCollection<DBBranch>();

                selectedBranch.ChildBranches.Add(newBranch);

                connection.send(new AddUpdateMessage(MUObject.UserName, newBranch));
                connection.send(new AddUpdateMessage(MUObject.UserName, selectedBranch));

                mainWindow.currentBranch = newBranch;

                OpenRevision(newBranch);

                this.Close();
            }
        }

        private void OpenRevision(DBBranch incomingSelectedBranch = null)
        {
            DBBranch selectedBranch = tvAssemblies.SelectedItem as DBBranch;

            if (incomingSelectedBranch != null)
            {
                selectedBranch = incomingSelectedBranch;
            }
            else
            {
                if (selectedBranch == null)
                    return;
            }

            //Get the current part from the selected branches feature list
            DBPart currentPart = null;
            if (incomingSelectedBranch == null && selectedBranch.DBInteropStates.Count != 0)
            {
                currentPart = selectedBranch.DBInteropStates.Last().DBFeature.DBPart;
            }
            else
            {
                if (selectedBranch.ParentBranch != null)
                {
                    currentPart = selectedBranch.ParentBranch.DBInteropStates.Last().DBFeature.
    DBPart;
                }
            }

            /* We can't have two parts with the same name open. Scan all open parts and close it
            * if they have the same name. */
            foreach (NXOpen.Part openPart in Utilities.NXSession.Parts.ToArray())
            {
                if (openPart.FullPath.EndsWith(currentPart.PartNumber + ".prt"))
                {
                        openPart.Close(BasePart.CloseWholeTree.True, BasePart.CloseModified.
        CloseModified, null);
                        break;
                    }
                }

                // Open the selected part
                MUPart topLevelPart = MUPart.GetInstanceFromServer(currentPart, null, CADTypes.NX,
        null, selectedBranch);

                NXOpen.PartLoadStatus status;
                Utilities.NXSession.Parts.SetDisplay(topLevelPart.NXPart, false, false, out status);

                mainWindow.currentBranch = selectedBranch;
            }

        }
}
```

Listing A.2: Branches Window Code (C#)

A.3 Part History Window

A.3.1 Part History Window GUI

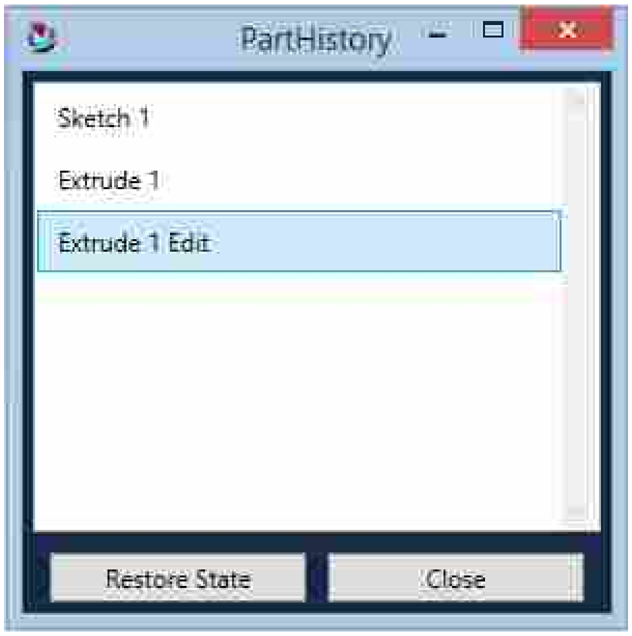


Figure A.1: Part hisotry window defined by the xaml code presented in this chapter.

A.3.2 Part History Window XAML code

```
<Window x:Class="CADInteropNX.PartHistory"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PartHistory" Height="300" Width="300">
    <DockPanel x:Name="LayoutRoot" Background="#172a4a">
        <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal" HorizontalAlignment="Center"
    >
            <Button Width="124" Content="Restore State" Click="open_Click" Padding="3" Margin="5"
    />
            <Button Width="124" Content="Close" Click="close_Click" Padding="3" Margin="5"/>
        </StackPanel>
        <DockPanel>
            <ListBox DockPanel.Dock="Bottom" x:Name="tvAssemblies" MouseDoubleClick="
    Tree_DoubleClick" Margin="5"
                     ScrollViewer.CanContentScroll="True" ScrollViewer.
    VerticalScrollBarVisibility="Visible" ScrollViewer.HorizontalScrollBarVisibility="Disabled">
```

```
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <Label Content="{Binding name}" />
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </DockPanel>
    </DockPanel>
</Window>
```

Listing A.3: Part History Window Code (XAML)

A.3.3 Branches Window Code C#

```
using ConnectData;
using ConnectData.Messages;
using NXOpen;
using NXOpen.UF;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Threading;
using System.Windows;
using System.Windows.Documents;
using System.Windows.Input;
using ConnectData.MUObjects;

namespace CADInteropNX
{
    /// <summary>
    /// Interaction logic for PartHistory.xaml
    /// </summary>
    public partial class PartHistory : Window
    {
        ClientConnection connection;
        MainWindow mainWindow;

        public PartHistory(ClientConnection incomingConnection, MainWindow sendingMainWindow)
        {
            InitializeComponent();

            connection = incomingConnection;
            mainWindow = sendingMainWindow;

            refreshInstanceList();
        }

        public class ReadableItems
        {
            public string name { get; set; }
            public DBInteropState state { get; set; }
        }

        public void refreshInstanceList()
        {
            List<ReadableItems> instances = new List<ReadableItems>();

            NXOpen.Part workPart = Utilities.NXSession.Parts.Work;
            MUPart workPartMU = MUPart.GetInstanceFromNX(workPart as NXOpen.Part);
            DBPart serverPart = workPartMU.ServerPart;

            FeatureListMessage featureListMsg = (FeatureListMessage)connection.sendReceive(new WatchPartMessage("NX", serverPart.GUID));

            List<DBFeature> features = featureListMsg.FeatureList.OrderBy(x => x.TreeOrder).ToList();

            foreach (DBFeature feature in features)
            {
                if (feature.DBInteropStates.Count != 0)
                {
                    feature.DBInteropStates = new ObservableCollection<DBInteropState>(feature.DBInteropStates.OrderBy(x => x.TimeStamp));
                }
            }

            int j = 1;
            int k = 1;
            foreach (DBFeature feature in features)
            {
                int i = 1;
                foreach (DBInteropState state in feature.DBInteropStates)
                {
                    if (!(state is DBPoint2DState) && !(state is DBLine2DState))
                    {
                        ReadableItems item = new ReadableItems();
                        if (state is DBSketchState)
                        {
                            if (i == 1)
                            {
                                item.name = "Sketch " + j.ToString();
                            }
                            else
                            {
                                item.name = "Sketch " + j.ToString() + " Edit";
                            }
                        }
                        else
                        {
                            if (i == 1)
                            {
                                item.name = "Extrude " + k.ToString();
                            }
                            else
                            {
                                item.name = "Extrude " + k.ToString() + " Edit";
                            }
                        }
                        item.state = state;
                        instances.Add(item);
                    }
                    i = i + 1;
                }
                if (feature is DBSketch)
                {
                    j = j + 1;
                }
                else if (feature.DBInteropStates.Count != 0 && feature.DBInteropStates.First() is
    DBExtrudeState)
                {
                    k = k + 1;
                }

            }

            instances.Sort((x, y) => DateTime.Compare(x.state.TimeStamp, y.state.TimeStamp));

            tvAssemblies.Items.Clear();
            foreach (ReadableItems item in instances)
            {
                tvAssemblies.Items.Add(item);
            }
        }

        private void Tree_DoubleClick(object sender, MouseButtonEventArgs e)
        {
            OpenRevision();
        }
        private void open_Click(object sender, RoutedEventArgs e)
        {
            OpenRevision();
        }
        private void close_Click(object sender, RoutedEventArgs e)
        {
            this.Close();
            //mainWindow.inLoadedRevision = false;

            //mainWindow.messageReceiveChannel = new OneWayChannel<Message>(System.Threading.
SynchronizationContext.Current, mainWindow.handleReceivedMessages);
            //DBInstance selectedInstance = tvAssemblies.SelectedItem as DBInstance;
            //MUObject.MUFeatures.Clear();

            ////Ensure that an instance is selected
            //if (selectedInstance == null)
            //    return;

            ////Set the selected part
            //DBPart currentPart = selectedInstance.DBPart;

            ///* We can't have two parts with the same name open. Scan all open parts and close
it
            //    * if they have the same name. */
            //foreach (NXOpen.Part openPart in Utilities.NXSession.Parts.ToArray())
            //{
            //    if (openPart.FullPath.EndsWith(currentPart.PartNumber + ".prt"))
            //    {
            //        openPart.Close(BasePart.CloseWholeTree.True, BasePart.CloseModified.
CloseModified, null);
            //        break;
            //    }
            //}

            //// Open the selected part
            //MUPart topLevelPart = MUPart.GetInstanceFromServer(currentPart, null, CADTypes.NX);

            //NXOpen.PartLoadStatus status;
            // Utilities.NXSession.Parts.SetDisplay(topLevelPart.NXPart, false, false, out status)
;
        }

        private void OpenRevision()
        {
            //Ensure that an instance is selected
            if (tvAssemblies.SelectedItem == null)
                return;

            DBInteropState selectedState = (tvAssemblies.SelectedItem as ReadableItems).state;

            mainWindow.inLoadedRevision = true;

            //Set the selected part
            DBPart currentPart = selectedState.DBFeature.DBPart;

            /* We can't have two parts with the same name open. Scan all open parts and close it
                * if they have the same name. */
            foreach (NXOpen.Part openPart in Utilities.NXSession.Parts.ToArray())
            {
                if (openPart.FullPath.EndsWith(currentPart.PartNumber + ".prt"))
                {
                    openPart.Close(BasePart.CloseWholeTree.True, BasePart.CloseModified.
        CloseModified, null);
                    break;
                }
            }

            //Since the sketch is not the last thing pushed for a sketch but we need to be able
        to load its children who get pushed slightly after them this is a band-aid fix where we set
        the timestamp passed into
            // the system as the last timestamp from the sketch entities of the dbsketch features
         interop states. Not a perfect solution for race cases but this is a proof of concept and I
        am not attempting to
            // support race cases for sketches.
            if (selectedState is DBSketchState)
            {
                selectedState = (from S in (selectedState.DBFeature as DBSketch).DBSketchEntities
                                 from X in S.DBInteropStates
                                 where X.TimeStamp <= selectedState.TimeStamp.AddSeconds(5)
                                 orderby X.TimeStamp
                                 select X as DBInteropState).LastOrDefault();
            }

            // Open the selected part
            MUPart topLevelPart = MUPart.GetInstanceFromServer(currentPart, null, CADTypes.NX,
        selectedState.TimeStamp);

            NXOpen.PartLoadStatus status;
            Utilities.NXSession.Parts.SetDisplay(topLevelPart.NXPart, false, false, out status);
        }

    }
}
```

Listing A.4: Branches Window Code (C#)

APPENDIX B. C# CODE

B.1 CATIA Code for Extrudes

```
/// <summary>
/// This method checks to see if the CATIA extrude is already in the MUFeatures dictionary.  If
    it is it's returned, if not it's created
/// </summary>
public static MUExtrude GetInstanceFromCATIA(Prism clientExtrude, Guid partGUID)
{
    if (MUFeatures.ContainsKey(Utilities.GetGuid(clientExtrude)))
        return (MUExtrude)MUFeatures[Utilities.GetGuid(clientExtrude)];

    MUExtrude result = new MUExtrude(clientExtrude, partGUID);

    return result;
}

/// <summary>
/// Creates a CATIA Extrude
/// </summary>
/// <param name="serverPoint"></param>
void CreateCATIAExtrude(DBFeature serverExtrude)
{
    ServerExtrude = serverExtrude;
    var currentState = CurrentState;
    GUID = serverExtrude.GUID;

    MECMOD.Sketch sketch = MUSketch.GetInstanceFromServer(currentState.ReferencedProfile as
    DBSketch, CADTypes.CATIA).catVersion;

    PARTITF.Pad pad;
    PARTITF.Pocket pocket;

    PartDocument partDoc = CurrentPart.CATIAPart;
    MECMOD.Body partBody = partDoc.Part.Bodies.Item(1);

    ShapeFactory shapeFactory = (ShapeFactory)partDoc.Part.ShapeFactory;
    partDoc.Part.InWorkObject = partBody;

    if (!currentState.IsPocket)
    {
        pad = (PARTITF.Pad)shapeFactory.AddNewPad((MECMOD.Sketch)sketch, currentState.Limit1);
        pad.FirstLimit.Dimension.Value = currentState.Limit1;
        pad.SecondLimit.Dimension.Value = currentState.Limit2;
        pad.DirectionOrientation = currentState.Direction ? CatPrismOrientation.
    catRegularOrientation : CatPrismOrientation.catInverseOrientation;
        pad.set_Name(currentState.DBFeature.Name);
        Utilities.SetGuid(pad, serverExtrude.GUID);
        CATIAExtrude = pad;
    }
    else
    {
        pocket = (PARTITF.Pocket)shapeFactory.AddNewPocket((MECMOD.Sketch)sketch, currentState.
    Limit1);
        pocket.FirstLimit.Dimension.Value = currentState.Limit1;
        pocket.SecondLimit.Dimension.Value = currentState.Limit2;
        pocket.DirectionOrientation = currentState.Direction ? CatPrismOrientation.
    catRegularOrientation : CatPrismOrientation.catInverseOrientation;
        pocket.set_Name(currentState.DBFeature.Name);
        Utilities.SetGuid(pocket, serverExtrude.GUID);
        CATIAExtrude = pocket;
    }

    partDoc.Part.Update();
}

private void UpdateCATIAExtrude(DBFeature serverExtrude)
{
    ServerExtrude = serverExtrude;
    DBExtrudeState currentState = CurrentState;
    try
    {
        /**
         * Right now we don't support Mirror Extent
         * If the user selects Mirror Extent, the SecondLimit, below, will be set to READ ONLY
    and
         * throw an error when we try to assign it.
         */
        CATIAExtrude.FirstLimit.Dimension.Value = currentState.Limit1;
        CATIAExtrude.SecondLimit.Dimension.Value = currentState.Limit2;
    }
    catch (Exception ex)
    {
        //AHA! we've been waiting for you
        string test = "hi";
    }

    if (currentState.Direction)
        CATIAExtrude.DirectionOrientation = CatPrismOrientation.catRegularOrientation;
    else
        CATIAExtrude.DirectionOrientation = CatPrismOrientation.catInverseOrientation;

    Utilities.UpdateCatiaPart();
}

/// <summary>
/// Use this method when the CATIAVersion has changed.
/// </summary>
public void UpdateFromCATIA(Prism ClientExtrude)
{
    CATIAExtrude = ClientExtrude;
    var currentState = CurrentState;

    bool changed = false;

    bool oldDirection = currentState.Direction;
    IProfileReference oldSketch = currentState.ReferencedProfile;
    double oldLimt1 = currentState.Limit1;
    double oldLimt2 = currentState.Limit2;

    if (currentState.Direction != (ClientExtrude.DirectionOrientation == CatPrismOrientation.
    catRegularOrientation))
    {
        changed = true;
        currentState.Direction = ClientExtrude.DirectionOrientation == CatPrismOrientation.
    catRegularOrientation;
    }

    if (currentState.DBFeature.GUID != Utilities.GetGuid(ClientExtrude.Sketch))
    {
        changed = true;
        currentState.ReferencedProfile = Profile.CurrentState.DBFeature as DBSketch;
    }

    if (currentState.Limit1 != ClientExtrude.FirstLimit.Dimension.Value)
    {
        changed = true;
        currentState.Limit1 = ClientExtrude.FirstLimit.Dimension.Value;
    }

    if (currentState.Limit2 != ClientExtrude.SecondLimit.Dimension.Value)
    {
        changed = true;
        currentState.Limit2 = ClientExtrude.SecondLimit.Dimension.Value;
    }

    if (changed)
    {
        if (isExtrudeValid(ClientExtrude))
        {
            InteropLogger.LogEntry("Feature Update Push", CurrentPart, this);
            Connection.send(new AddUpdateMessage(UserName, currentState));
        }
        else
        {
            currentState.Direction = oldDirection;
            currentState.ReferencedProfile = oldSketch;
            currentState.Limit1 = oldLimt1;
            currentState.Limit2 = oldLimt2;
        }
    }
}
```

Listing B.1: CATIA Code for Extrudes

B.2 CREO Code for Extrudes

```
void CreateCreoExtrude(DBFeature serverExtrude)
{
    ServerExtrude = serverExtrude;
    var currentState = CurrentState;
    GUID = serverExtrude.GUID;

    CreoElementTree = CreateElementTree(currentState);

    StringBuilder sb = new StringBuilder();
    sb.AppendLine("CreateFeature");
    sb.AppendLine(ServerExtrude.DBPart.GUID.ToString());
    sb.AppendLine(ServerExtrude.GUID.ToString());
    sb.AppendLine(CreoElementTree);
    Utilities.PerformCommand(sb, sb.Length);
}

string CreateElementTree(DBExtrudeState serverExtrude)
{
    double limit1, limit2, tmp;

    limit1 = serverExtrude.Limit1;
    limit2 = serverExtrude.Limit2;

    if (limit1 < 0 || limit2 < 0)
    {
        tmp = limit1;
        limit1 = -limit2;
        limit2 = -tmp;
    }

    StringWriter sw = new StringWriter();

    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_FEATURE_TREE, 0, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_FEATURE_TYPE, 1, 0, serverExtrude.IsPocket
    ? "916" : "917"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_FEATURE_FORM, 1, 0, "1"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_FEATURE_NAME, 1, 4, serverExtrude.
    DBFeature.Name));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_SURF_CUT_SOLID_TYPE, 1, 0, "917"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_REMOVE_MATERIAL, 1, 0, serverExtrude.
    IsPocket ? "916" : "-1"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_IS_SMT_CUT, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_CUT_NORMAL_DIR, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SKETCHER, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_SECTION, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_SEC_METHOD, 2, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SEC_USE_SKETCH, 2, 10, serverExtrude.
    ReferencedProfileGUID.ToString()));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_FEAT_FORM_IS_THIN, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_MATRLSIDE, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_THICKNESS, 1, 1, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SRF_END_ATTRIBUTES, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_TRIM_QUILT, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_TRIM_QLT_SIDE, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_DIRECTION, 1, 0, serverExtrude.
    Direction ? "1" : "-1"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_EXT_DEPTH, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_FROM, 2, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_FROM_TYPE, 3, 0, "128"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_FROM_VALUE, 3, 1, limit2.ToString
    ()));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_FROM_REF, 3, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_TO, 2, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_TO_TYPE, 3, 0, "262144"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_TO_VALUE, 3, 1, limit1.ToString()
    ));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DEPTH_TO_REF, 3, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_INT_PARTS, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_PATTERN, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_SMT_THICKNESS, 1, 1, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_STD_SMT_SWAP_DRV_SIDE, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_WALL_SHARPS_TO_BENDS, 1, 0, "0"));
    sw.WriteLine(WriteElement(5112, 1, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_FILLETS, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_FILLETS_SIDE, 2, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_FILLETS_VALUE, 2, 1, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_DEV_LEN_CALCULATION, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_DEV_LEN_SOURCE, 2, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_DEV_LEN_Y_FACTOR, 2, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_DEV_LEN_Y_FACTOR_TYPE, 3, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_DEV_LEN_Y_FACTOR_VALUE, 3, 1, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_DEV_LEN_BEND_TABLE, 2, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_PUNCH_TOOL_DATA, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_PUNCH_TOOL_ATTR, 2, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_PUNCH_TOOL_NAME, 2, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_MERGE_DATA, 1, 9));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_MERGE_AUTO, 2, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_SMT_MERGE_KEEP_LINES, 2, 0, "0"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_COMP_DRFT_ANG, 1, 9));
```

```
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DRFT_ANG, 2, 0, "-1"));
    sw.WriteLine(WriteElement((int)ElementID.wfcPRO_E_EXT_DRFT_ANG_VAL, 2, 1, "0"));

    sw.WriteLine("Sketch");
    sw.WriteLine(serverExtrude.ReferencedProfileGUID.ToString());

    return sw.ToString();
}

 void EditCreoExtrude(DBFeature newExtrude)
{
    ServerExtrude = newExtrude;
    DBExtrudeState currentState = CurrentState;


    //bool updated = false;

    //updated = updated || currentState.Direction != ParseElementTree(CreoElementTree, GUID,
    newExtrude.DBPart.GUID).Direction;
    //updated = updated || currentState.Limit1 != currentState.Limit1;
    //updated = updated || currentState.Limit2 != currentState.Limit2;
    //updated = updated || currentState.IsPocket != currentState.IsPocket;
    //updated = updated || currentState.DBSketch.GUID != currentState.DBSketch.GUID;

    //if (updated)
    //{
        //CurrentState = currentState;
        CreoElementTree = CreateElementTree(currentState);

        StringBuilder sb = new StringBuilder();
        sb.AppendLine("EditFeature");
        sb.AppendLine(ServerExtrude.DBPart.GUID.ToString());
        sb.AppendLine(ServerExtrude.GUID.ToString());
        sb.AppendLine(CreoElementTree);
        Utilities.PerformCommand(sb, sb.Length);
    //}
}
```

Listing B.2: CREO Code for Extrudes

B.3 NX Code for Extrudes

```
/// <summary>
/// This method checks to see if the NX extrude is already in the MUFeatures dictionary.
/// If it is it's returned, if not it's created
/// </summary>
public static MUExtrude GetInstanceFromNX(NXOpen.Features.Extrude clientExtrude, Guid partGUID,
    DBBranch currentBranch)
{
    if (MUFeatures.ContainsKey(Utilities.GetGuid(clientExtrude)))
        return (MUExtrude)MUFeatures[Utilities.GetGuid(clientExtrude)];

    return new MUExtrude(clientExtrude, partGUID, currentBranch);
}

/// <summary>
/// Creates an NX Extrude
/// </summary>
/// <param name="serverPoint"></param>
void CreateOrUpdateNXExtrude(DBFeature serverExtrude, DateTime? timestamp = null)
{
    try
    {
        ServerExtrude = serverExtrude;
        var currentState = CurrentState;

        //Check if we are opening a revision that isn't current
        List<DBInteropState> states = new List<DBInteropState>();
        if (timestamp != null)
        {
            currentState = (from S in ServerExtrude.DBInteropStates
                            where S.TimeStamp <= timestamp
                            orderby S.TimeStamp
                            select S as DBExtrudeState).Last();
        }

        GUID = serverExtrude.GUID;
        NXOpen.Part featurePart = MUObject.MUParts[serverExtrude.DBPart.GUID].NXPart;

        //Switch the work part
        NXOpen.Part currentPart = MUObject.CurrentPart.NXPart;
        Utilities.SetWorkWithoutHandler(featurePart);

        //Get the sketch to be extruded, and get the sketch feature
        Profile = MUSketch.GetInstanceFromServer(currentState.ReferencedProfile as DBSketch,
    CADTypes.NX);
```

```
        NXOpen.Sketch nxSketch = Profile.nxVersion;
        NXOpen.Features.SketchFeature nxSketchFeature = (NXOpen.Features.SketchFeature)nxSketch.
    Feature;

        //Create the direction of extrusion from the sketch.  Set sense to forward if the
    connectData.sketch direction is true,
        //and set the sense to reverse if set to false
        NXOpen.Direction sketchDir;
        if (currentState.Direction)
        {
            sketchDir = featurePart.Directions.CreateDirection(nxSketch, Sense.Forward,
    SmartObject.UpdateOption.WithinModeling);
        }
        else
        {
            sketchDir = featurePart.Directions.CreateDirection(nxSketch, Sense.Reverse,
    SmartObject.UpdateOption.WithinModeling);
        }

        //Create extrude builder, and create section in extrude builder
        NXOpen.Features.ExtrudeBuilder extrudeBuilder = featurePart.Features.CreateExtrudeBuilder
    (NXExtrude);
        NXOpen.Section section = featurePart.Sections.CreateSection();
        extrudeBuilder.Section = section;

        //Get extrude length
        extrudeBuilder.Limits.EndExtend.Value.RightHandSide = currentState.Limit1.ToString();
        extrudeBuilder.Limits.StartExtend.Value.RightHandSide = (-currentState.Limit2).ToString()
    ;

        //Pass sketch feature into the section
        //sketch feature -> feature array -> CurveFeatureRule -> SelectionIntentRule array ->
    section
        NXOpen.Features.Feature[] featureContainer = new NXOpen.Features.Feature[1];
        featureContainer[0] = nxSketchFeature;
        NXOpen.CurveFeatureRule curveFeatureRule = featurePart.ScRuleFactory.
    CreateRuleCurveFeature(featureContainer);
        NXOpen.SelectionIntentRule[] rule = new NXOpen.SelectionIntentRule[1];
        rule[0] = curveFeatureRule;
        NXOpen.Point3d helperPoint = new NXOpen.Point3d(0, 0, 0);
        section.AddToSection(rule, null, null, null, helperPoint, NXOpen.Section.Mode.Create,
    false);

        //set extrude direction
        extrudeBuilder.Direction = sketchDir;

        //Get all the bodies from the work part, and set solid bodies as targets for either
    addition/subtraction boolean operation
        NXOpen.Body[] bodies = featurePart.Bodies.ToArray();
        List<NXOpen.Body> targetBodiesList = new List<NXOpen.Body>();
        for (int i = 0; i < bodies.Length; i++)
        {
            if (bodies[i].IsSolidBody == true)
            {
                targetBodiesList.Add(bodies[i]);
            }
        }
        NXOpen.Body[] targetBodies = targetBodiesList.ToArray();

        //Create an undo mark for undoing this operation if there are any bodies found that need
    to be united to this body
        Utilities.NXSession.SetUndoMark(Session.MarkVisibility.Visible, "extrudeUndo");

        //is it an extrusion or subtraction?
        if (currentState.IsPocket == true && targetBodies.Length != 0 && currentState.Limit1 +
    currentState.Limit2 != 0.0)
        {
            extrudeBuilder.BooleanOperation.Type = NXOpen.GeometricUtilities.BooleanOperation.
    BooleanType.Create;
            NXOpen.Body[] nullBodies = { null };
            extrudeBuilder.BooleanOperation.SetTargetBodies(nullBodies);
        }
        else
        {
            extrudeBuilder.BooleanOperation.Type = NXOpen.GeometricUtilities.BooleanOperation.
    BooleanType.Create;
            NXOpen.Body[] nullBodies = { null };
            extrudeBuilder.BooleanOperation.SetTargetBodies(nullBodies);
        }

        //Commit builder
        List<NXOpen.Body> connectedTargetBodies = new List<NXOpen.Body>();
        try
        {
            NXOpen.Features.Extrude result = (NXOpen.Features.Extrude)extrudeBuilder.Commit();
            result.SetAttribute("Name", serverExtrude.Name);
            Utilities.SetGuid(result, GUID);


            NXExtrude = result;

            //Try to unite to bodies
            foreach (NXOpen.Body target in targetBodies)
            {
                NXOpen.Features.BooleanBuilder booleanBuilder = Utilities.NXSession.Parts.Work.Features.CreateBooleanBuilderUsingCollector(null);
                NXOpen.Body resultBody = (NXOpen.Body)Utilities.NXSession.Parts.Work.Bodies.FindObject(result.JournalIdentifier);
                if (resultBody != target)
                {
                    try
                    {
                        //Unite the body just created to the target and see if it works
                        booleanBuilder.Tolerance = .025;
                        booleanBuilder.Operation = NXOpen.Features.Feature.BooleanType.Unite;

                        booleanBuilder.Targets.Add(resultBody);
                        NXOpen.Body[] testBodies = new NXOpen.Body[1];
                        testBodies[0] = target;
                        NXOpen.BodyDumbRule bodyDumbrule = Utilities.NXSession.Parts.Work.ScRuleFactory.CreateRuleBodyDumb(testBodies);
                        NXOpen.SelectionIntentRule[] rules = new SelectionIntentRule[1];
                        rules[0] = bodyDumbrule;
                        ScCollector scCollector = Utilities.NXSession.Parts.Work.ScCollectors.CreateCollector();
                        scCollector.ReplaceRules(rules, false);
                        booleanBuilder.ToolBodyCollector = scCollector;

                        NXOpen.NXObject newObject = booleanBuilder.Commit();

                        //If the last line worked add it to the connectedTargetBodiesList
                        connectedTargetBodies.Add(target);

                        booleanBuilder.Destroy();
                    }
                    catch
                    {/* We don't actually want this to do anything. Just catching errors so that I can get a list of the objects that work. */
                        booleanBuilder.Destroy();
                    }
                }
            }
        }
        catch (NXException ex)
        {

        }

        //Undo to the last undo and then do the extrude with the connectedTargetBodies if the
    list length > 0
        if (connectedTargetBodies.Count == 1)
        {
            Utilities.NXSession.UndoToLastVisibleMark();
            if (currentState.IsPocket)
                extrudeBuilder.BooleanOperation.Type = NXOpen.GeometricUtilities.BooleanOperation
    .BooleanType.Subtract;
            else
                extrudeBuilder.BooleanOperation.Type = NXOpen.GeometricUtilities.BooleanOperation
    .BooleanType.Unite;
            extrudeBuilder.BooleanOperation.SetTargetBodies(connectedTargetBodies.ToArray());

            try
            {
                NXOpen.Features.Extrude result = (NXOpen.Features.Extrude)extrudeBuilder.Commit()
    ;
                result.SetAttribute("Name", serverExtrude.Name);
                Utilities.SetGuid(result, GUID);

                NXExtrude = result;
            } catch (NXException ex)
            {
            }
        }
        else if (connectedTargetBodies.Count == 0 && currentState.IsPocket)
        {
            Utilities.NXSession.UndoToLastVisibleMark();
            extrudeBuilder.BooleanOperation.Type = NXOpen.GeometricUtilities.BooleanOperation.
    BooleanType.Subtract;
            extrudeBuilder.BooleanOperation.SetTargetBodies(targetBodies);
        }

        CurrentState = currentState;

        //Change work part back to part user was working on
        try
        {
            Utilities.SetWorkWithoutHandler(currentPart);
        }
        catch (NXOpen.NXException ex) { }

    }
    catch (NXException ex)
    {
        MUSketch sketch = MUSketch.GetInstanceFromServer(CurrentState.ReferencedProfile as
    DBSketch, CADTypes.NX);
        sketch.nxVersion.Color = 146;
        sketch.nxVersion.RedisplayObject();
        throw new Exception("An error occured creating an extrude" + ex.Message, ex);
    }
}
 /// <summary>
/// This method is called by the Instance method above. A NX extrude is converted to a server
    extrude
/// </summary>
private MUExtrude(NXOpen.Features.Extrude clientExtrude, Guid partGUID, DBBranch currentBranch)
{
    //HACK: There must be two extrude builders running?
    NXExtrude = clientExtrude;
    NXOpen.Part featurePart = MUObject.MUParts[partGUID].NXPart;

    Guid FeatureGUID = Guid.NewGuid();

    NXOpen.Features.ExtrudeBuilder clientExtrudeBuilder = featurePart.Features.
    CreateExtrudeBuilder(clientExtrude);

    bool direction = true;
    bool isPocket = false;

    //set the direction of the extrusion
    if (clientExtrudeBuilder.Direction.Sense == Sense.Reverse)
        direction = false;

    //make it a subtraction
    if (clientExtrudeBuilder.BooleanOperation.Type == NXOpen.GeometricUtilities.BooleanOperation.
    BooleanType.Subtract)
        isPocket = true;

    //find the sketch that the extrude is based on
    NXOpen.Features.Feature[] parents = clientExtrude.GetParents();
    NXOpen.Sketch parentSketch = null;
    foreach (NXOpen.Features.Feature feat in parents)
    {
        if (feat is NXOpen.Features.SketchFeature)
        {
            parentSketch = ((NXOpen.Features.SketchFeature)feat).Sketch;
            break;
        }
    }

    NXOpen.Features.SketchFeature parentSketchFeature = (NXOpen.Features.SketchFeature)
    parentSketch.Feature;
    Profile = MUSketch.GetInstanceFromNX(parentSketchFeature, CurrentPart.GUID, currentBranch);

    //create the new extrude
    var currentExtrudeState = new ConnectData.DBExtrudeState()
    {
        Direction = direction,
        IsPocket = isPocket,
        Limit1 = Convert.ToDouble(clientExtrudeBuilder.Limits.EndExtend.Value.RightHandSide),
        Limit2 = -Convert.ToDouble(clientExtrudeBuilder.Limits.StartExtend.Value.RightHandSide),
        //Name = GUID.ToString(),
        GUID = Guid.NewGuid(),
        //DBPart = MUObject.CurrentPart.ServerPart,
        ReferencedProfile = Profile.DBSketch as DBSketch,
        ReferencedProfileGUID = Profile.GUID,
        //FeatureTypeID = 5
        DBBranch = currentBranch,
        BranchGUID = currentBranch.GUID
    };

    currentExtrudeState.DBFeature = new DBFeature()
    {
        GUID = FeatureGUID,
        Name = FeatureGUID.ToString(),
        DBPart = MUObject.CurrentPart.ServerPart,
        FeatureTypeID = 5
    };

    GUID = FeatureGUID;

    Utilities.SetGuid(clientExtrude, GUID);
    Utilities.SetName(clientExtrude, currentExtrudeState.DBFeature.Name);

    Message message = new AddUpdateMessage(UserName, currentExtrudeState.DBFeature);
    Connection.send(message);

    Message message2 = new AddUpdateMessage(UserName, currentExtrudeState);
    Connection.send(message2);
    currentExtrudeState.DBFeature.DBInteropStates.Add(currentExtrudeState);
    ServerExtrude = currentExtrudeState.DBFeature;

    MUFeatures.Add(GUID, this);
}
```

Listing B.3: NX Code for Extrudes

APPENDIX C. LOADING CODE

```

void CreateNXPart(DBPart serverPart, DateTime? timestamp = null, DBBranch currentBranch = null)
{
    GUID = serverPart.GUID;
    PartNumber = serverPart.PartNumber;
    ServerPart = serverPart;
    MUParts.Add(GUID, this);

    /* Remove part changed handler before creating a part. This ensures that
     * the WorkpartChanged handler doesn't fire before the GUID of the new
     * part is set. We'll add the handler back after the part is created. */
    Utilities.RemovePartHandlers();

    //Create a blank part file to pull the features into
    NXOpen.FileNew newFile = Utilities.NXSession.Parts.FileNew();
    newFile.TemplateFileName = "model-plain-1-mm-template.prt";
    newFile.Application = NXOpen.FileNewApplication.Modeling;
    newFile.Units = NXOpen.Part.Units.Millimeters;
    newFile.NewFileName = "C:\\Temp\\" + ServerPart.PartNumber + ".prt";
    newFile.MakeDisplayedPart = true;
    NXOpen.Part nxPart = newFile.Commit() as NXOpen.Part;
    NXPart = nxPart;

    Utilities.SetGuid(NXPart, GUID);
    Utilities.SetName(NXPart, PartNumber);

    MUObject.CurrentPart = this;

    Utilities.AddPartHandlers();

    // Look up the features related to the part
    FeatureListMessage msg = (FeatureListMessage)Connection.sendReceive(new WatchPartMessage("NX"
    , ServerPart.GUID));
    List<DBFeature> NewFeatures = msg.FeatureList.OrderBy(x => x.TreeOrder).ToList();

    // Upon load if the branch is null then set it to the first features branch
    var count = NewFeatures.Where(x => x is DBSketch).ToList().Count;
    var sketchList = NewFeatures.Where(x => x is DBSketch).ToList().OrderBy(x => x.TreeOrder);

    if (currentBranch == null && NewFeatures.Where(x => x is DBSketch).ToList().Count > 0 &&
    NewFeatures.Where(x => x is DBSketch).ToList().OrderBy(x => x.TreeOrder).First().
    DBInteropStates.OrderBy(x => x.TimeStamp).ToList().First().DBBranch != null)
    {
        currentBranch = NewFeatures.Where(x => x is DBSketch).ToList().OrderBy(x => x.TreeOrder).
    First().DBInteropStates.OrderBy(x => x.TimeStamp).ToList().First().DBBranch;
    }
    else if (NewFeatures.Where(x => x is DBSketch).ToList().Count == 0)
    {
        currentBranch = new DBBranch()
        {
            Name = "Main",
            GUID = Guid.NewGuid()
        };
    }


    // Check to make sure features were created before the timestamp
    if (timestamp != null)
    {
        foreach (DBFeature feature in NewFeatures.ToList())
        {
            if (feature.TimeStamp > timestamp)
            {
                NewFeatures.Remove(feature);
            }
        }
    }

    //Get the parent branches
    List<Guid> parentBranchGuids = new List<Guid>(); //Also will include current branch
    var parentBranch = currentBranch.ParentBranch;
    parentBranchGuids.Add(currentBranch.GUID);
    while (parentBranch != null)
    {
        parentBranchGuids.Add(parentBranch.GUID);
        var tempBranch = parentBranch;
        parentBranch = tempBranch.ParentBranch;
    }

    Dictionary<Guid, DateTime> timestampDictionary = new Dictionary<Guid,DateTime>();

```

```
// Check to make sure the features have states were created on the current branch or its parent branches.
foreach (DBFeature feature in NewFeatures.ToList())
{
    var stateCount = 0;
    List<DBSketch> sketchesToRemoveEntities = new List<DBSketch>();
    DateTime? latestTimestamp = null;
    foreach (DBInteropState state in feature.DBInteropStates.ToList())
    {

        //See if the state is an extrude or sketch and make sure it is on the current branch or a parent branch.
        if ((state is DBExtrudeState || state is DBSketchState) && (parentBranchGuids.Contains(state.DBBranch.GUID)))
        {
            var tempCount = 0;
            stateCount++;
            tempCount++;
            //Check to make sure that the state if on a parent branch was made before the child branch which is related to the current branch was created.
            var parent = currentBranch.ParentBranch;
            var temp = currentBranch;
            while (parent != null)
            {
                //If the state is on a parent branch.
                if (state.DBBranch.GUID == parent.GUID)
                {
                    // And if the current branch was created before the state on the parent branch was created remove the count for that state.
                    if (temp.TimeStamp < state.TimeStamp)
                    {
                        stateCount--;
                        tempCount--;

                        //Remove the state from the local stash so it isn't loaded.
                        NewFeatures.Find(x => x.GUID == feature.GUID).DBInteropStates.Remove(state);
                        if (state is DBSketchState) {
                            sketchesToRemoveEntities.Add((state as DBSketchState).DBFeature as DBSketch);
                        }
                    }
                }
                parent = parent.ParentBranch;
            }
```

```
                if (tempCount > 0)
                {
                    if (latestTimestamp != null && state.TimeStamp > latestTimestamp)
                    {
                        latestTimestamp = state.TimeStamp;
                    }
                }
            }
            else if (state is DBExtrudeState || state is DBSketchState)
            {
                NewFeatures.Find(x => x.GUID == feature.GUID).DBInteropStates.Remove(state);
                if (state is DBSketchState)
                {
                    sketchesToRemoveEntities.Add((state as DBSketchState).DBFeature as DBSketch);
                }
            }
        }
        if (stateCount == 0 && (feature is DBSketch || feature.FeatureTypeID == 5))
        {
            NewFeatures.Remove(feature);
        }
        //If the feature is a sketch we must also remove all of its sketch entities or the system will build the sketch anyway.
        //We don't want to remove all unless their sketch is gone from the newFeatures list or if they have states that do not exist in the current branch we also want to remove those.
        if (feature is DBSketch)
        {
            foreach (var sketchEntity in NewFeatures.ToList())
            {
                if (sketchEntity is DBSketchEntity)
                {
                    if ((sketchEntity as DBSketchEntity).DBSketch.GUID == feature.GUID)
                    {
                        if (stateCount == 0) {
                            var itemToRemove = NewFeatures.Find(x => x.GUID == sketchEntity.GUID);
                            NewFeatures.Remove(itemToRemove);
                        } else if (sketchesToRemoveEntities.Count != 0) {
                            foreach (var sketchEntityState in (sketchEntity as DBSketchEntity).DBInteropStates.ToList())
                            {
                                if (sketchEntityState.TimeStamp > latestTimestamp)
                                {
                                    var itemToRemove = NewFeatures.Find(x => x.GUID == sketchEntity.GUID);
                                    NewFeatures.Remove(itemToRemove);
                                }
                            }
                        }
                    }
                }
            }
            if (latestTimestamp != null)
            {
                timestampDictionary.Add(feature.GUID, latestTimestamp.Value);
            }
        }
    }

    // Add new features to the part
    while (NewFeatures.Count > 0)
    {
        if (timestampDictionary.ContainsKey(NewFeatures[0].GUID))
        {
            CreateFeature(NewFeatures[0], CADTypes.NX, timestampDictionary[NewFeatures[0].GUID].AddSeconds(5));
        }
        else
        {
            CreateFeature(NewFeatures[0], CADTypes.NX, timestamp);
        }
        NewFeatures.Remove(NewFeatures[0]);
    }

    // Add child components to the part
    List<DBInstance> Children = ServerPart.Children.ToList();
    foreach (DBInstance child in Children)
    {
        MUInstance.GetInstanceFromServer(child, CADTypes.NX);
    }

    Connection.send(new AddUpdateMessage(UserName, currentBranch));

    CurrentBranch = currentBranch;
}
```

Listing C.1: Loading code within MUPart.cs showing how the algorithms provided in the thesis were applied.

APPENDIX D. DATABASE .EDMX FILE

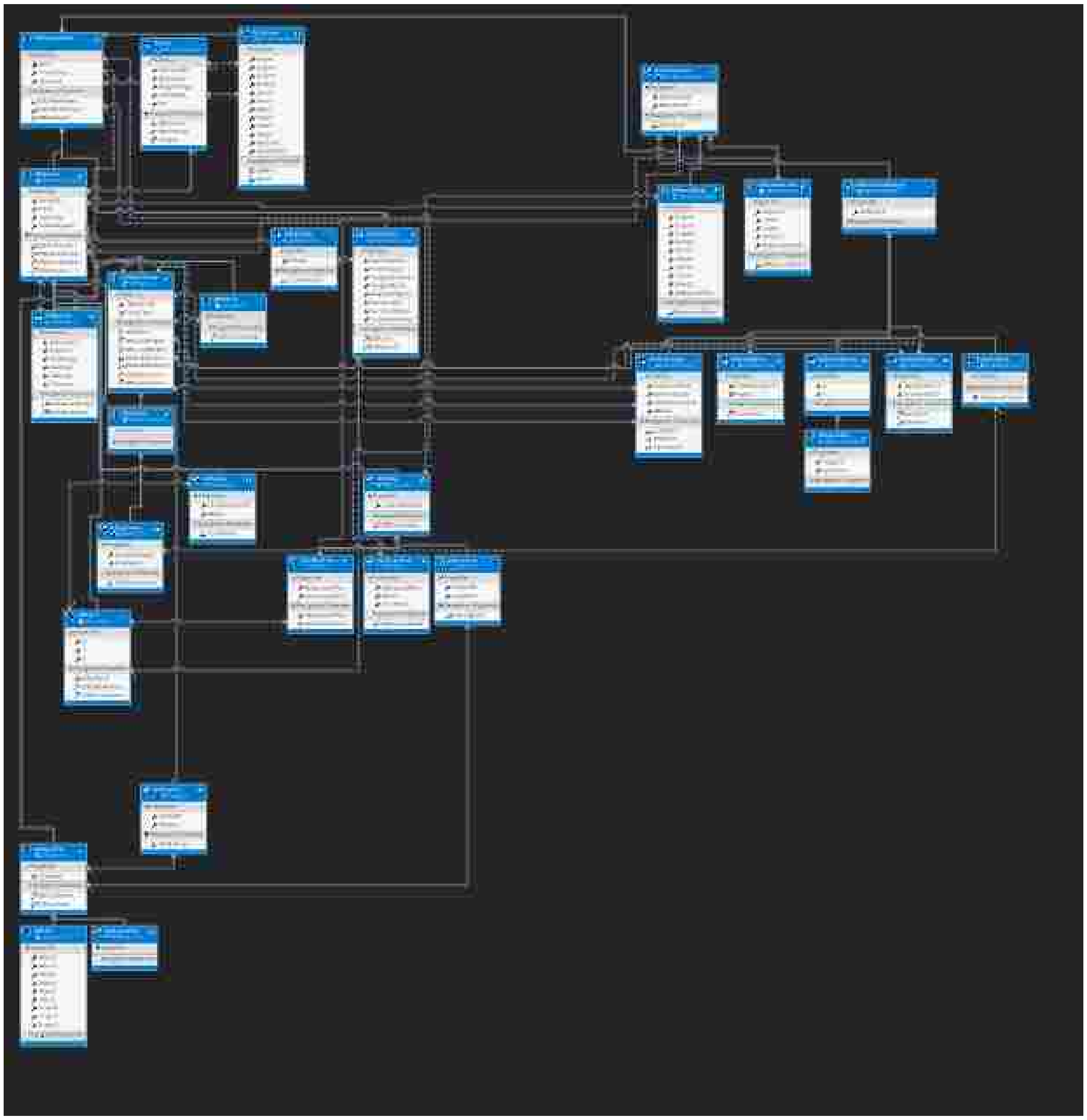
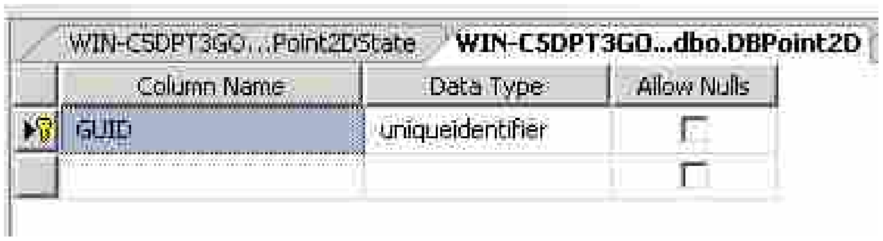


Figure D.1: Full database .edmx file showing all connections and tables.

APPENDIX E. ADDING A NEW FEATURE

E.1 Database Setup

1. Remote into server
2. Open SQL Management Studio
3. Under the database ”State-Transaction” add 2 new tables
   - Feature
     This table represents the feature in the database, this Feature Table only has a primary key which can be seen in Figure E.1. Every table that is created in the database should have this primary key which is a GUID.



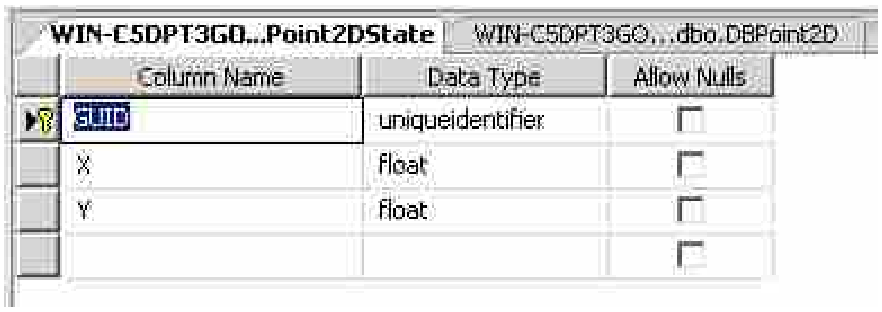
WIN-C5DPT3GO...Point2DState | WIN-C5DPT3GO...dbo.DBPoint2D

Column Name	Data Type	Allow Nulls
GUID	uniqueidentifier	☐
		☐

Figure E.1: Feature Table for DBPoint2D which shows a GUID as the primary key.

   - State
     This table contains the information to define an instance of this feature using the Neutral Parametric Canonical Form. In Figure E.2 a 2D point is represented. As shown in the example, to create a point in any CAD system the CAD system needs an X coordinate and a Y coordinate.

4. Add relationships to the tables
   Every table that you add to the database needs to inherit from another table that exists in the


WIN-C5DPT3GO...Point2DState | WIN-C5DPT3GO...dbo.DBPoint2D

Column Name	Data Type	Allow Nulls
GUID	uniqueidentifier	☐
X	float	☐
Y	float	☐
		☐

Figure E.2: Table containing the state information to define an instance of the Point2DState.

database if you are adding a feature. In order to add a new relationship right click on the table and select relationships. Figure E.3 shows how to add a new relationship between the feature and other features that it needs in order to be created. In Figure E.3 an example from Line2D is shown. In the window on the right, the primary key table is the feature table that you want to relate the current feature to, in this instance the table is DBSketchEntity which represents a Point2D. The reference should always be GUID under this primary key table. Under Foreign Key Table the feature for which you are creating relationships should automatically be selected. In this relationship window select the table entry for the relationship. In Figure E.3 the connection is StartPointGUID. In summary by doing this you are creating a link between StartPointGUID and the Point2D table's primary key allowing the code side implementation to grab the data contained in the Point2D table.

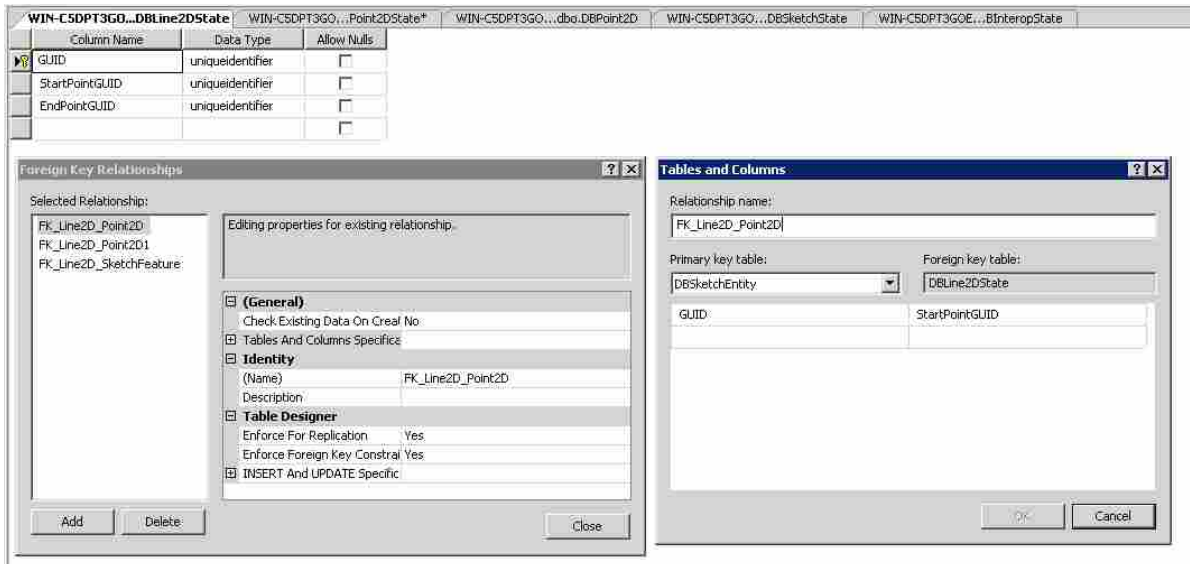
5. Add inheritance relationship

   In addition to possibly creating table relationships you also need to create an inheritance relationship. In Figure E.4 the typical relationship for a new sketch feature is shown. The SketchEntityState is what the table should point to and both fields should be GUID. For a typical feature that is not contained in a sketch the Primary key table should be DBInterop-State.

6. Return to visual studio (Images from Visual Studio 2012)

7. Add table to edmx file.

   In order to add these new tables to the code we return to Visual Studio 2012 and open the Interop solution. Under the solution explorer find and open the file under the ConnectData



WIN-C5DPT3GO...DBLine2DState | WIN-C5DPT3GO...Point2DState* | WIN-C5DPT3GO...dbo.DBPoint2D | WIN-C5DPT3GO...DBSketchState | WIN-C5DPT3GOE...BInteropState

Column Name	Data Type	Allow Nulls
GUID	uniqueidentifier	☐
StartPointGUID	uniqueidentifier	☐
EndPointGUID	uniqueidentifier	☐
		☐

Foreign Key Relationships

Selected Relationship:
FK_Line2D_Point2D
FK_Line2D_Point2D1
FK_Line2D_SketchFeature

Editing properties for existing relationship.

⊟ (General)	
Check Existing Data On Creat	No
⊞ Tables And Columns Specifica	
⊟ Identity	
(Name)	FK_Line2D_Point2D
Description	
⊟ Table Designer	
Enforce For Replication	Yes
Enforce Foreign Key Constrai	Yes
⊞ INSERT And UPDATE Specific	

Add | Delete | Close

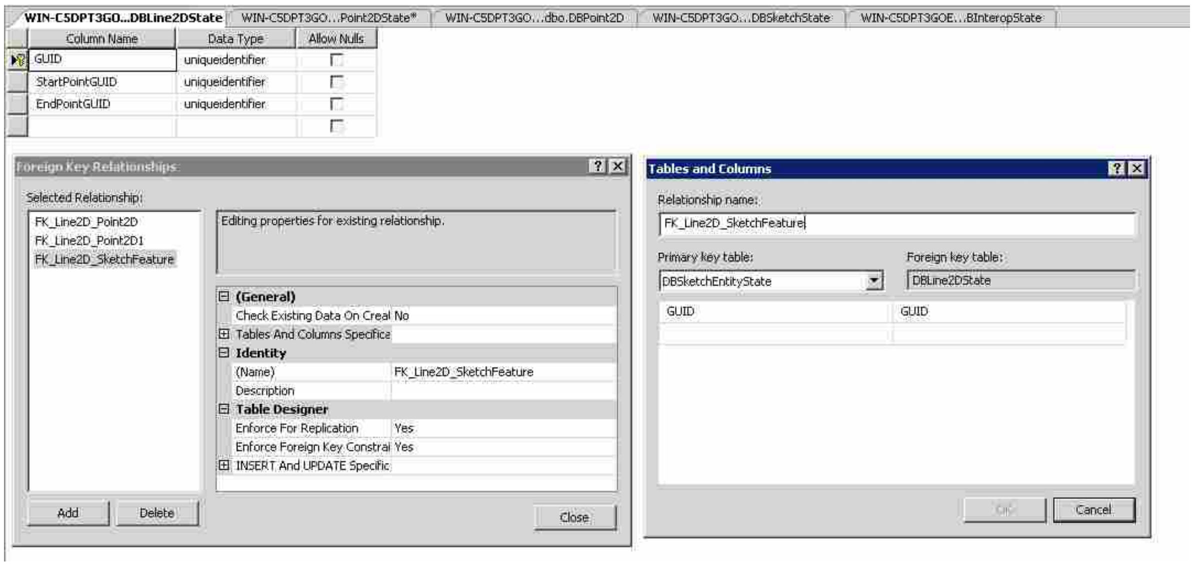
Tables and Columns

Relationship name: FK_Line2D_Point2D

Primary key table:	Foreign key table:
DBSketchEntity	DBLine2DState
GUID	StartPointGUID

OK | Cancel

Figure E.3: Foreign Key Relationship between Line2D and Point2D for a start point.



WIN-C5DPT3GO...DBLine2DState | WIN-C5DPT3GO...Point2DState* | WIN-C5DPT3GO...dbo.DBPoint2D | WIN-C5DPT3GO...DBSketchState | WIN-C5DPT3GOE...BInteropState

Column Name	Data Type	Allow Nulls
GUID	uniqueidentifier	☐
StartPointGUID	uniqueidentifier	☐
EndPointGUID	uniqueidentifier	☐
		☐

Foreign Key Relationships

Selected Relationship:
FK_Line2D_Point2D
FK_Line2D_Point2D1
FK_Line2D_SketchFeature

Editing properties for existing relationship.

⊟ (General)	
Check Existing Data On Creat	No
⊞ Tables And Columns Specifica	
⊟ Identity	
(Name)	FK_Line2D_SketchFeature
Description	
⊟ Table Designer	
Enforce For Replication	Yes
Enforce Foreign Key Constrai	Yes
⊞ INSERT And UPDATE Specific	

Add | Delete | Close

Tables and Columns

Relationship name: FK_Line2D_SketchFeature

Primary key table:	Foreign key table:
DBSketchEntityState	DBLine2DState
GUID	GUID

OK | Cancel

Figure E.4: Foreign Key Relationship to define inheritance between Line2D and SketchFeature.

project titled ConnectData.edmx. With this file open right click to bring up the menu seen in Figure E.5. Click the line ”Update Model from Database...” to open the window seen in Figure E.6. Under the add tab, select the table that you would like to add. You will have to add both the feature and the state tables. In Figure E.6 the DBLine2DState table is selected to be added. Click ”Finish” in the bottom to add the table.

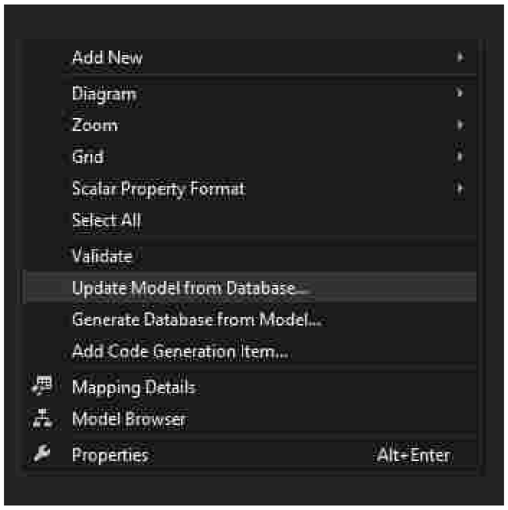
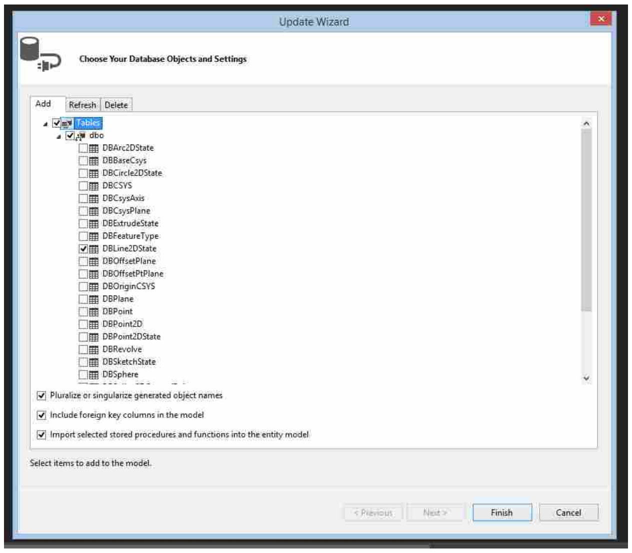


Figure E.5: Right click menu in the .edmx file window of visual studio with “Update Model from Database” highlighted.



- [✓] Tables
  - [✓] dbo
    - [ ] DBArc2DState
    - [ ] DBBaseCsys
    - [ ] DBCircle2DState
    - [ ] DBCSYS
    - [ ] DBCsysAxis
    - [ ] DBCsysPlane
    - [ ] DBExtrudeState
    - [ ] DBFeatureType
    - [✓] DBLine2DState
    - [ ] DBOffsetPlane
    - [ ] DBOffsetPtPlane
    - [ ] DBOriginCSYS
    - [ ] DBPlane
    - [ ] DBPoint
    - [ ] DBPoint2D
    - [ ] DBPoint2DState
    - [ ] DBRevolve
    - [ ] DBSketchState
    - [ ] DBSphere

[✓] Pluralize or singularize generated object names

[✓] Include foreign key columns in the model

[✓] Import selected stored procedures and functions into the entity model

Select items to add to the model.

Figure E.6: Window showing how to add a new database table by updating the model from the database.

8. Remove the inheritance line

   At this point the inheritance is treated as a normal relationship so select the line that displays to show the table’s relationship with its parent table as seen in Figure E.7. Delete this line.

9. Add inheritance relationship

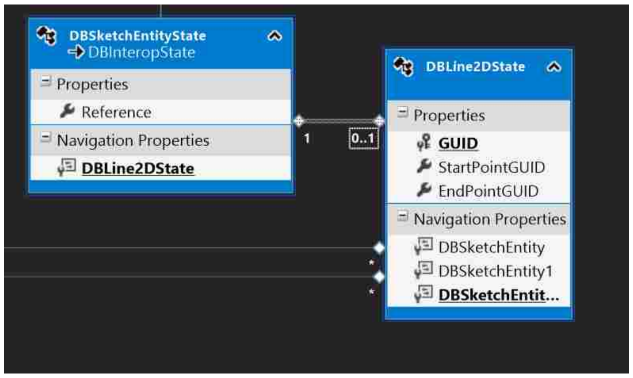
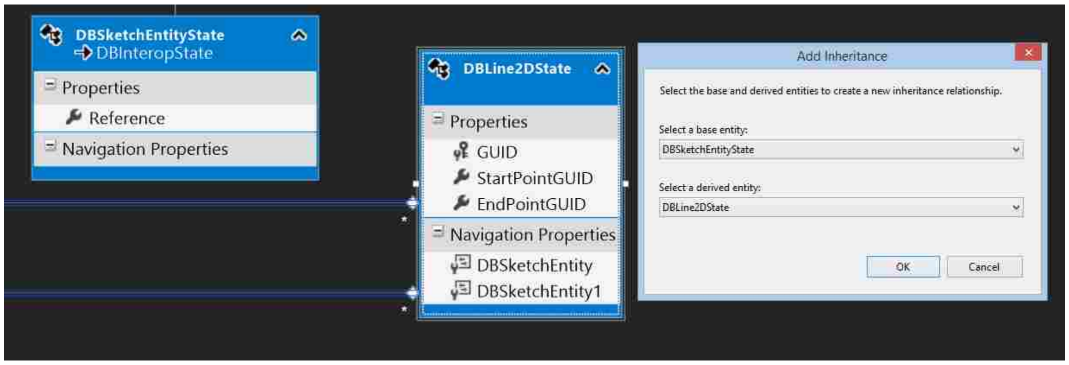


Figure E.7: Edmx file window showing the DBLine2DState and its parent table DBSketchEntityState right after being added in visual studio.

- Right click in the background of the .edmx file and select add and then inheritance.
- When the ”Add Inheritance” window appears select the base entity as the parent table and the derived entity as the table that you just added as seen in Figure E.8 for the Line2D table.
- Select ”ok” to add the inheritance relationship.
- Select the GUID in the new table and delete it since the GUID is now being inherited from a parent table. The missing GUID can be seen in Figure E.9.



Select the base and derived entities to create a new inheritance relationship.

Select a base entity: DBSketchEntityState

Select a derived entity: DBLine2DState

Figure E.8: Edmx file window showing the DBLine2DState and its parent table DBSketchEntityState with the add inheritance window showing how to create a new inheritance relationship.

10. Rename navigation properties

    As seen in Figure E.9 the navigational properties are generic names to the table the feature is linked to. We need to rename these properties to give context later in the code. In this instance we select the DBSketchEntity and we can see that StartPointGUID is highlighted meaning that this navigational property points to the SketchEntity for the start point of the line. We rename this property ”StartPoint” and the second entry ”EndPoint”. We also need to rename the navigation properties under DBSketchEntity to ”DBLine2DStartPoint” and ”DBLine2DEndPoint”. These changes can be seen in Figure E.10.

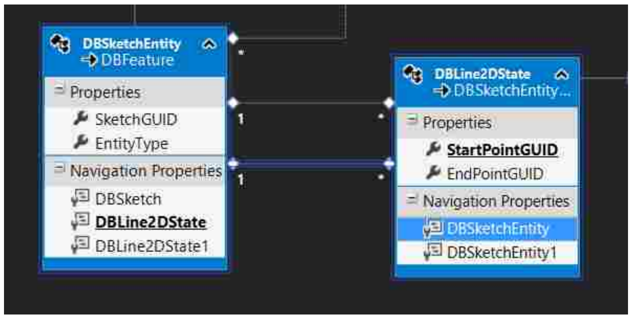


Figure E.9: Initial view of the DBSketchEntity and DBLine2DState before renaming navigation properties.

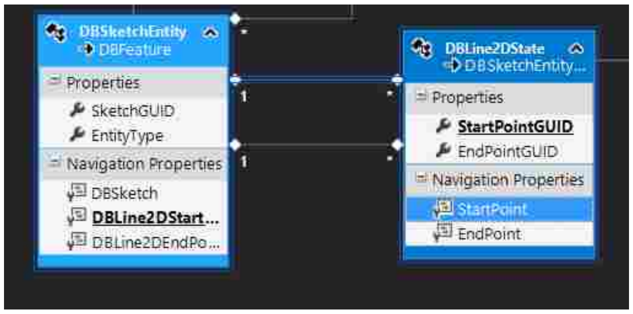


Figure E.10: Initial view of the DBSketchEntity and DBLine2DState after renaming navigation properties.

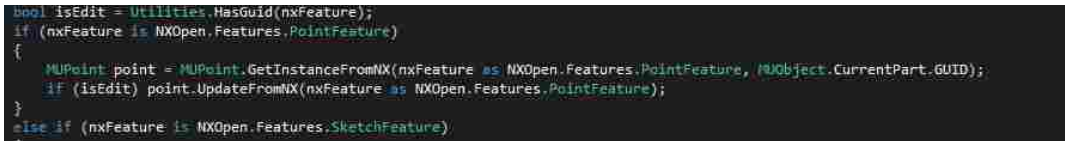
The current state of the database tables for Interop can be seen in Appendix D.

E.2 Code

1. Register UI Callbacks

   Collect the name of the button in NX and add to the .men file. This changes for each system but to do so in NX open NX and right click on the ribbon and select ”customize” at the bottom of the right click menu. From here select ”Keyboard”. After the keyboard window is open select the button you would like to know the name of on the left and copy the id from the right side of the window.

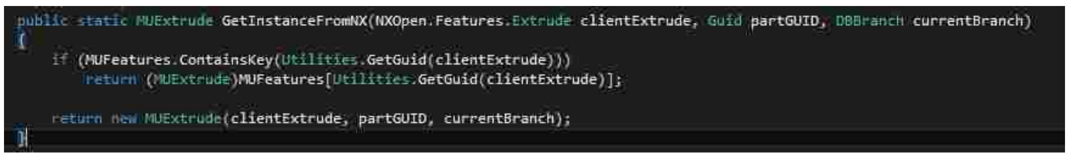
2. Add feature to list in MainWindow.xaml.cs function PushFeature.
   In the push feature function add a new if statement for the feature you are creating, similar to Figure E.11 but customized for your feature.



```
bool isEdit = Utilities.HasGuid(nxFeature);
if (nxFeature is NXOpen.Features.PointFeature)
{
    MUPoint point = MUPoint.GetInstanceFromNX(nxFeature as NXOpen.Features.PointFeature, MUObject.CurrentPart.GUID);
    if (isEdit) point.UpdateFromNX(nxFeature as NXOpen.Features.PointFeature);
}
else if (nxFeature is NXOpen.Features.SketchFeature)
```

Figure E.11: Code segment for finding and updating a feature in NX.

3. Add function GetInstanceFromNX for your feature
   In the ConnectData project under MUObjects you will find a file for your newly created tables. (If you do not see it go to the .edmx file we manipulated previously and select build from the file menu and then select transform all t4 templates.) Inside this file add a function for GetInstanceFromNX that looks similar to Figure E.12. This will check to see if the object exists and if it doesn’t then system will create a new feature using the constructor for your new object passing in the object, partGUID, and current branch.

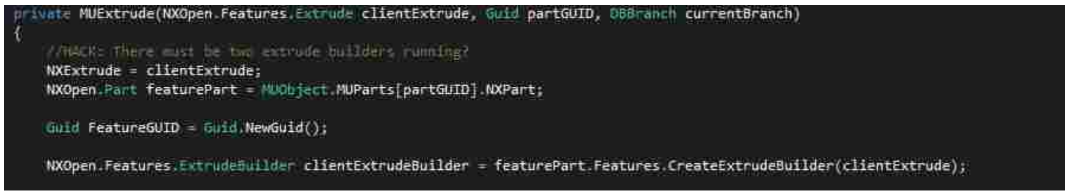


```
public static MUExtrude GetInstanceFromNX(NXOpen.Features.Extrude clientExtrude, Guid partGUID, DBBranch currentBranch)
{
    if (MUFeatures.ContainsKey(Utilities.GetGuid(clientExtrude)))
        return (MUExtrude)MUFeatures[Utilities.GetGuid(clientExtrude)];

    return new MUExtrude(clientExtrude, partGUID, currentBranch);
}
```

Figure E.12: Code segment for getting instance from NX.

4. Update the constructor for your current object
   At this point this code will change depending on the CAD system that you are adding the feature for. For a full example of extrude see Appendix B.

For NX you need to grab the feature, which is the first object passed in to the constructor, and create a builder. This process can be seen in Figure E.13.



```
private MUExtrude(NXOpen.Features.Extrude clientExtrude, Guid partGUID, DBBranch currentBranch)
{
    //HACK: There must be two extrude builders running?
    NXExtrude = clientExtrude;
    NXOpen.Part featurePart = MUObject.MUParts[partGUID].NXPart;

    Guid FeatureGUID = Guid.NewGuid();

    NXOpen.Features.ExtrudeBuilder clientExtrudeBuilder = featurePart.Features.CreateExtrudeBuilder(clientExtrude);
```

Figure E.13: Code segment to create a builder for an extrude in NX.

At this point you need to create the message to send to the server. This message must contain all of the data needed to fill the database table that you created. For Extrude the information that is needed can be seen in Figure E.14. This creates the new state and feature tables, sets the GUID to a new GUID, sets the name of the feature, and then sends the messages to the server. These messages must be sent in order, the parent feature must be pushed before the child state table data. After this has been done we must add the feature GUID to the MUFeatures dictionary. This section of the code should be very similar between CAD systems with the only part that changes being the data on the right side of the equals sign.

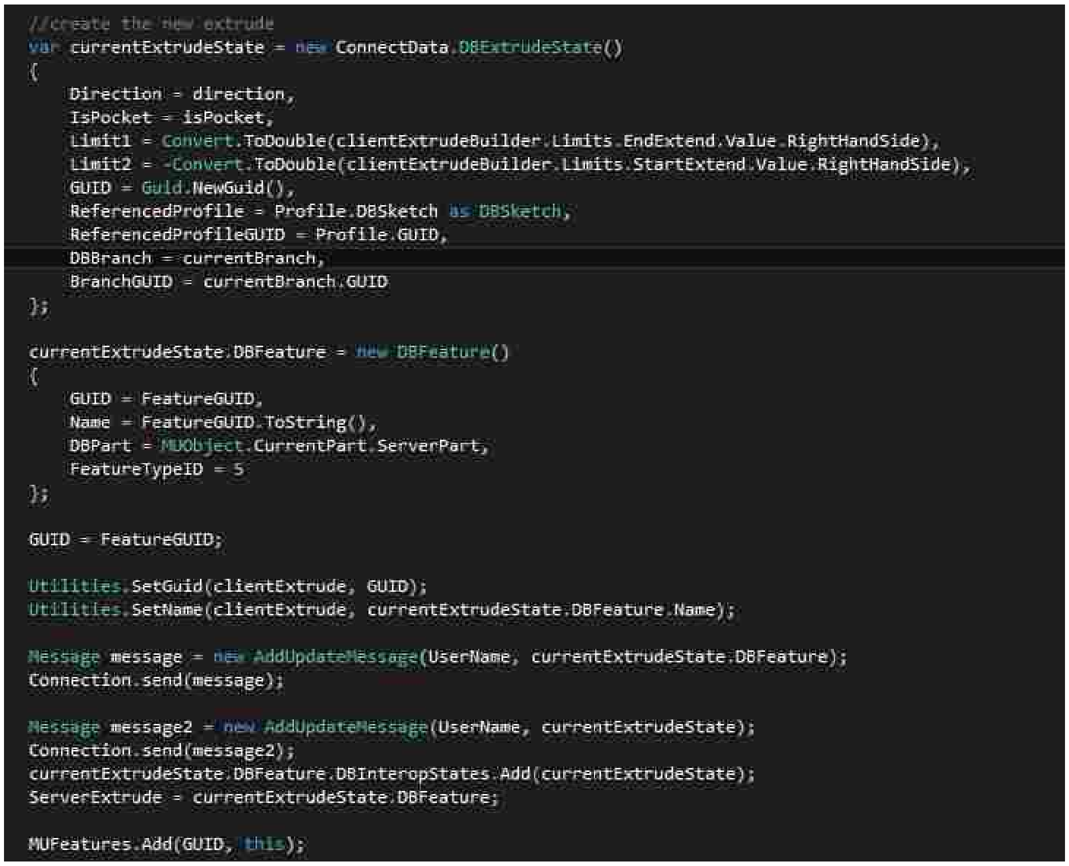
This is all that is needed to update the code to push the feature data. This step will require research along with creating the state table which requires the user to define the NPCF for the feature which takes time and research.

5. Check edits

   To check edits add the function Update from NX to the MUNewFeature file you are currently editing. In this updateFromNX function you need to check every data point that is being set in the table to see if it has changed. If no change has been made, do not send a message because that will create a new edit state which is not needed as no edit took place. Figures E.15 and E.16 this process can be seen but without any of the data.

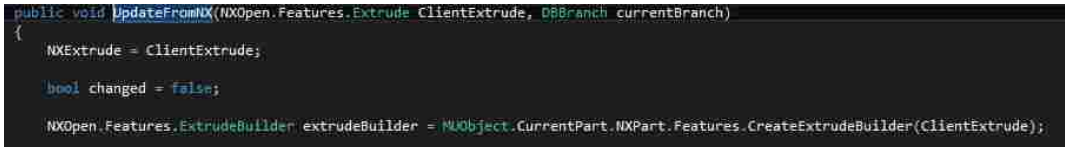
6. Handle incoming features from other users

   To handle incoming features the msg will be executed in the function executeMessage which is in the MainWindow file of the CAD program plug-in that you are working on. For the most



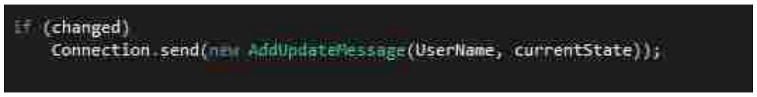
```
//create the new extrude
var currentExtrudeState = new ConnectData.DBExtrudeState()
{
    Direction = direction,
    IsPocket = isPocket,
    Limit1 = Convert.ToDouble(clientExtrudeBuilder.Limits.EndExtend.Value.RightHandSide),
    Limit2 = -Convert.ToDouble(clientExtrudeBuilder.Limits.StartExtend.Value.RightHandSide),
    GUID = Guid.NewGuid(),
    ReferencedProfile = Profile.DBSketch as DBSketch,
    ReferencedProfileGUID = Profile.GUID,
    DBBranch = currentBranch,
    BranchGUID = currentBranch.GUID
};

currentExtrudeState.DBFeature = new DBFeature()
{
    GUID = FeatureGUID,
    Name = FeatureGUID.ToString(),
    DBPart = MUObject.CurrentPart.ServerPart,
    FeatureTypeID = 5
};

GUID = FeatureGUID;

Utilities.SetGuid(clientExtrude, GUID);
Utilities.SetName(clientExtrude, currentExtrudeState.DBFeature.Name);

Message message = new AddUpdateMessage(UserName, currentExtrudeState.DBFeature);
Connection.send(message);

Message message2 = new AddUpdateMessage(UserName, currentExtrudeState);
Connection.send(message2);
currentExtrudeState.DBFeature.DBInteropStates.Add(currentExtrudeState);
ServerExtrude = currentExtrudeState.DBFeature;

MUFeatures.Add(GUID, this);
```

Figure E.14: Code segment showing how to create a new extrude state and feature then send those features to the database.



```
public void UpdateFromNX(NXOpen.Features.Extrude ClientExtrude, DBBranch currentBranch)
{
    NXExtrude = ClientExtrude;

    bool changed = false;

    NXOpen.Features.ExtrudeBuilder extrudeBuilder = MUObject.CurrentPart.NXPart.Features.CreateExtrudeBuilder(ClientExtrude);
```

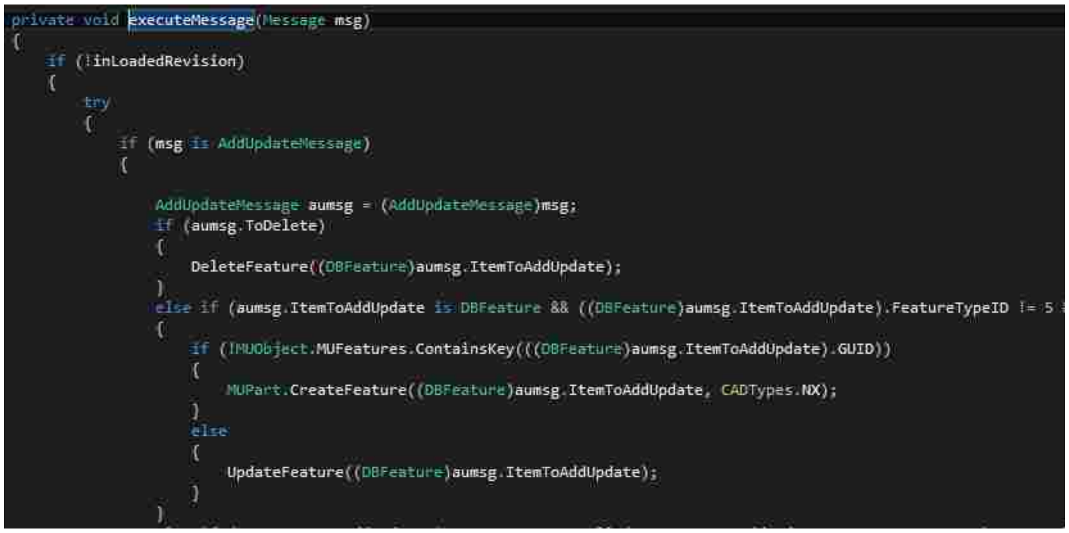
Figure E.15: Code segment for creating a builder in an update from NX.



```
if (changed)
    Connection.send(new AddUpdateMessage(UserName, currentState));
```

Figure E.16: Code segment for only sending the updated state if an edit has been performed.

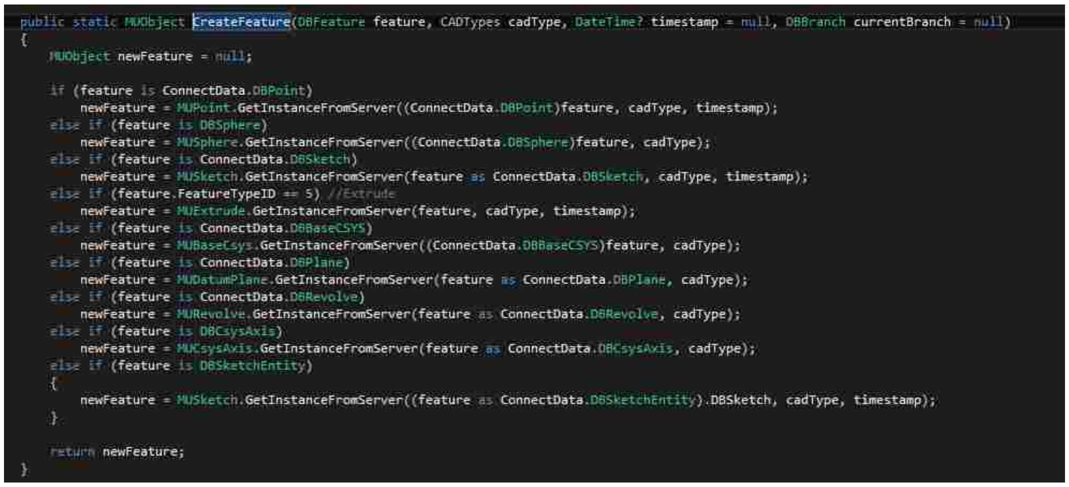
part features will be handled by the UpdateFeature function call inside of executeMessage (seen in Figure E.17) but sometimes this file will need to be edited such as for incoming plane objects.



```
private void executeMessage(Message msg)
{
    if (!inLoadedRevision)
    {
        try
        {
            if (msg is AddUpdateMessage)
            {

                AddUpdateMessage aumsg = (AddUpdateMessage)msg;
                if (aumsg.ToDelete)
                {
                    DeleteFeature((DBFeature)aumsg.ItemToAddUpdate);
                }
                else if (aumsg.ItemToAddUpdate is DBFeature && ((DBFeature)aumsg.ItemToAddUpdate).FeatureTypeID != 5
                {
                    if (!MUObject.MUFeatures.ContainsKey(((DBFeature)aumsg.ItemToAddUpdate).GUID))
                    {
                        MUPart.CreateFeature((DBFeature)aumsg.ItemToAddUpdate, CADTypes.NX);
                    }
                    else
                    {
                        UpdateFeature((DBFeature)aumsg.ItemToAddUpdate);
                    }
                }
```

Figure E.17: Code segment for executeMessage showing how new messages from the server are handled.

Create feature will always need to be edited for new features and the feature you are adding will need to be added to the if statement contained within this function.



```
public static MUObject CreateFeature(DBFeature feature, CADTypes cadType, DateTime? timestamp = null, DBBranch currentBranch = null)
{
    MUObject newFeature = null;

    if (feature is ConnectData.DBPoint)
        newFeature = MUPoint.GetInstanceFromServer((ConnectData.DBPoint)feature, cadType, timestamp);
    else if (feature is DBSphere)
        newFeature = MUSphere.GetInstanceFromServer((ConnectData.DBSphere)feature, cadType);
    else if (feature is ConnectData.DBSketch)
        newFeature = MUSketch.GetInstanceFromServer(feature as ConnectData.DBSketch, cadType, timestamp);
    else if (feature.FeatureTypeID == 5) //Extrude
        newFeature = MUExtrude.GetInstanceFromServer(feature, cadType, timestamp);
    else if (feature is ConnectData.DBBaseCSYS)
        newFeature = MUBaseCsys.GetInstanceFromServer((ConnectData.DBBaseCSYS)feature, cadType);
    else if (feature is ConnectData.DBPlane)
        newFeature = MUDatumPlane.GetInstanceFromServer(feature as ConnectData.DBPlane, cadType);
    else if (feature is ConnectData.DBRevolve)
        newFeature = MURevolve.GetInstanceFromServer(feature as ConnectData.DBRevolve, cadType);
    else if (feature is DBCsysAxis)
        newFeature = MUCsysAxis.GetInstanceFromServer(feature as ConnectData.DBCsysAxis, cadType);
    else if (feature is DBSketchEntity)
    {
        newFeature = MUSketch.GetInstanceFromServer((feature as ConnectData.DBSketchEntity).DBSketch, cadType, timestamp);
    }

    return newFeature;
}
```

Figure E.18: Create feature if statement showing where and how to add a new feature in the create feature list.

7. Handle feature specific bugs

   This method is the general method for adding new features to the state-transactional database but as new features are added they will need to be debugged. I have only implemented a new feature that worked exactly right first try two times out of 70 or so features so changes will most likely be needed to fix feature specific bugs.