A PLATFORM INDEPENDENT PROCESS DATA EXCHANGE MECHANISM
BETWEEN JAVA APPLICATIONS AND COMPUTER AIDED
MANUFACTURING SYSTEMS

A thesis presented to

the faculty of the

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

of

Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Vikesh Jain

November 2003
This thesis entitled

A PLATFORM INDEPENDENT PROCESS DATA EXCHANGE MECHANISM
BETWEEN JAVA APPLICATIONS AND COMPUTER AIDED
MANUFACTURING SYSTEMS

by

Vikesh Jain

has been approved for
the Department of Industrial and Manufacturing Systems Engineering
and the Russ College of Engineering and Technology by

Dušan Šormaz
Assistant Professor of Industrial and Manufacturing Systems Engineering

R. Dennis Irwin
Dean, Russ College of Engineering and Technology
A PLATFORM INDEPENDENT PROCESS DATA-EXCHANGE MECHANISM BETWEEN JAVA APPLICATIONS AND CAM SYSTEMS

(115 pp.)

Director of Thesis: Dušan Šormaz

The need to share data between applications is more then ever before as integration of different applications is becoming difficult. This requires an integrated manufacturing system to achieve effective information exchange between disparate applications. The focus of this research was on developing a process data exchange mechanism between a Java application and a CAM application allowing process data to be exchanged between two different applications. This provides interoperability between applications and also allows the CAM application to be controlled from outside its environment. The approach that was taken in this thesis can in general be applied to different manufacturing applications for sharing process data.

Approved: Dušan Šormaz

Assistant Professor Industrial and Manufacturing Systems Engineering
ACKNOWLEDGMENTS

It is a pleasure to thank the many people who made this thesis possible. My sincere gratitude to my thesis advisor, Dr. Dušan Šormaz, who helped and guided me towards implementing the ideas presented in this thesis. His dedication to research and his effort in the development of my thesis was an inspiration throughout this work.

The thesis would not be successful without other members of my committee, Dr. Dale Masel and Dr. Bhavin Mehta. Special thanks to them for their substantial help and suggestions during the development of this thesis. I would like also to thank Dr. Dale Masel for his class on guidelines for how to write thesis.

Thanks to my fellow colleagues and members of the IMPlanner Group, Srinivas Rajaraman, Sachin Jain, Harihara Sharma, Jaikumar Arumugam for their excellent cooperation and suggestions. I would also like to thank my roomies Fanil, Kapil, Reddy and Suresh for their constant support and encouragement.

Finally, it has been pleasure to pursue graduate studies at IMSE department at Ohio University, a unique place that has provided me with great exposures to intricacies underlying development, programming and integration of different industrial systems; thus making this thesis possible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>14</td>
</tr>
<tr>
<td>1.1 Application Data exchange</td>
<td>14</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>15</td>
</tr>
<tr>
<td>1.3 Need for JNI and XML for Process Data exchange</td>
<td>16</td>
</tr>
<tr>
<td>1.4 Problems in Process Data exchange</td>
<td>17</td>
</tr>
<tr>
<td>1.5 Research Goal</td>
<td>18</td>
</tr>
<tr>
<td>1.6 Summary of Process Data exchange</td>
<td>18</td>
</tr>
<tr>
<td>1.7 Outline of Thesis</td>
<td>20</td>
</tr>
<tr>
<td>2 BACKGROUND WORK</td>
<td>21</td>
</tr>
<tr>
<td>2.1 Process Specification Language</td>
<td>21</td>
</tr>
<tr>
<td>2.2 XML Overview</td>
<td>26</td>
</tr>
<tr>
<td>2.2.1 Structure of XML document</td>
<td>27</td>
</tr>
<tr>
<td>2.2.2 Components of an XML document</td>
<td>28</td>
</tr>
<tr>
<td>2.3 UG / Open Application Programming Interface</td>
<td>29</td>
</tr>
<tr>
<td>2.3.1 Unigraphics manufacturing environment</td>
<td>30</td>
</tr>
</tbody>
</table>
2.4 Previous Work in Process-data exchange 33

2.5 Java Native Interface 35
  2.5.1 JNI components 36
  2.5.2 JNI structure 37
  2.5.3 Distributed computing environment with CORBA 39

3 IMPLANNER SYSTEM ARCHITECTURE 42
  3.1 Process-data exchange Architecture 44

4 METHODOLOGY 47
  4.1 Process Data exchange Approach 47
  4.2 Querying CAM Data 48
    4.2.1 Sample CAM Query Output 48
  4.3 Process Mapping 51
  4.4 Creating CAM Data 54
  4.5 Saving and Retrieving Process in XML 56

5 IMPLEMENTATION 61
  5.1 Process-Plan Model 61
    5.1.1 MfgProcess Classes 62
    5.1.2 CAMPartModel Class 63
    5.1.3 CAMOperation Class 65
5.1.4 CAMSystem Class

5.1.5 Implementing Native Methods Using JNI

5.2 Process Data exchange Graphical User Interface

5.3 Retrieving CAM System Data
  5.3.1 Retrieving Tags from Native Application
  5.3.2 Extracting the Details of a UG Operation
  5.3.3 Getting Groups for UG Operation
  5.3.4 Starting the CAM Session

5.4 Creating Operations in CAM System
  5.4.1 Process Mapping
  5.4.2 Creating Operations in Unigraphics

5.5 Saving Process Model in XML File
  5.5.1 Writing an XML File
  5.5.2 Writing the Element Tags
  5.5.3 Reading an XML File

6 TESTING

6.1 Testing Examples
  6.1.1 NetExample
  6.1.2 Loading Part File
  6.1.3 Querying CAM Operation Data

6.2 Creating Operations in Unigraphics
6.3 Saving Process Data 106
6.4 Reading the XML 107

7 CONTRIBUTIONS, LIMITATIONS AND FUTURE WORK 109
7.1 Contributions 109
7.2 Limitations 110
7.3 Future work 110

REFERENCES 112
LIST OF TABLES

Table 4-1 Process mapping 52
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Example using PSL for defining an activity object</td>
<td>23</td>
</tr>
<tr>
<td>2-2</td>
<td>Example of translation to ILOG using PSL</td>
<td>23</td>
</tr>
<tr>
<td>2-3</td>
<td>Application A’s terminology and ontology</td>
<td>24</td>
</tr>
<tr>
<td>2-4</td>
<td>Application B’s terminology and ontology</td>
<td>25</td>
</tr>
<tr>
<td>2-5</td>
<td>PSL ontology</td>
<td>25</td>
</tr>
<tr>
<td>2-6</td>
<td>Structure of an XML document</td>
<td>27</td>
</tr>
<tr>
<td>2-7</td>
<td>Elements of an XML file</td>
<td>28</td>
</tr>
<tr>
<td>2-8</td>
<td>Sample Unigraphics file with a block</td>
<td>30</td>
</tr>
<tr>
<td>2-9</td>
<td>Loading of Unigraphics templates for initializing a CAM session</td>
<td>30</td>
</tr>
<tr>
<td>2-10</td>
<td>Dialog for CreateOperation in Unigraphics</td>
<td>31</td>
</tr>
<tr>
<td>2-11</td>
<td>Groups related to a Unigraphics operation</td>
<td>32</td>
</tr>
<tr>
<td>2-12</td>
<td>JNI mapping</td>
<td>36</td>
</tr>
<tr>
<td>2-13</td>
<td>Sample Java class calling a native method</td>
<td>37</td>
</tr>
<tr>
<td>2-14</td>
<td>Header file for the HelloWorld class</td>
<td>38</td>
</tr>
<tr>
<td>2-15</td>
<td>native implementation of displayHelloWorld method</td>
<td>39</td>
</tr>
<tr>
<td>3-1</td>
<td>IMPlanner system architecture</td>
<td>42</td>
</tr>
<tr>
<td>3-2</td>
<td>Process-data exchange architecture</td>
<td>44</td>
</tr>
<tr>
<td>4-1</td>
<td>Results of sample CAM query</td>
<td>49</td>
</tr>
<tr>
<td>4-2</td>
<td>Querying CAM data from Java application</td>
<td>49</td>
</tr>
<tr>
<td>4-3</td>
<td>CAM data generation</td>
<td>54</td>
</tr>
<tr>
<td>4-4</td>
<td>Operation Navigator Tree (ONT)</td>
<td>56</td>
</tr>
</tbody>
</table>
Figure 5-20 Method \textit{createOperation} defined in \textit{CAMPartModel} \hfill 83

Figure 5-21 Implementation of native method \textit{createOperation} \hfill 84

Figure 5-22 Method \textit{writeXML} implemented in the \textit{MfgProcess} class part 1 \hfill 87

Figure 5-23 Method \textit{writeXML} implemented in the \textit{MfgProcess} class part 2 \hfill 88

Figure 5-24 Method \textit{writeXML} for \textit{ToolConstraint} class \hfill 89

Figure 5-25 Process data generated by 3I-PP system part 1 \hfill 89

Figure 5-26 Process data generated by 3I-PP system part 2 \hfill 90

Figure 5-27 Method startElement for \textit{constraintSaxHandler} Class \hfill 91

Figure 5-28 Method endElement for \textit{constraintSaxHandler} class \hfill 92

Figure 6-1 \textit{NetExample’s} Process Data exchange Interface \hfill 94

Figure 6-2 Anc101\_Manuf.prt file \hfill 95

Figure 6-3 List of operations from the Anc101\_Manuf.prt file \hfill 97

Figure 6-4 Switcharm.prt file \hfill 98

Figure 6-5 List of Operations from the switcharm\_manuf.prt file \hfill 98

Figure 6-6 Block\_.prt file with list of features \hfill 99

Figure 6-7 \textit{NetExample} interface before Block.prt file is loaded \hfill 100

Figure 6-8 \textit{NetExample} interface after Block.prt file is loaded \hfill 101

Figure 6-9 Feature panel for the slot feature \hfill 101

Figure 6-10 Features and Process are shown in a \textit{JTree} \hfill 102

Figure 6-11 Create Operation Pop Up Menu for Block.prt file \hfill 103

Figure 6-12 Operation successfully created inside Block.prt file \hfill 104

Figure 6-13 List of Operations retrieved from Block.prt file \hfill 105

Figure 6-14 Process and Feature Data saved to BlockXml File \hfill 106
Figure 6-15 List of Features read from BlockXML.xml file.
1 INTRODUCTION

1.1 Application Data exchange

Technological innovations and advances are rapidly changing today’s manufacturing infrastructure. Manufacturers are constantly being challenged to become more productive through shortened product development cycles. Information is a key issue in addressing these challenges. An integrated manufacturing system is required so as to allow effective information exchange. The need to bind together disparate systems has made application data exchange the main focus of research. This research aims at providing effective information exchange between applications that shares information with another application on its successful implementation.

This research puts emphasis on the development of a platform independent data exchange mechanism by linking a Java application and a native application. The system that is used for this research is an extension of the IMPlanner system. The IMPlanner system has several independent modules each performing a specific task in the product development life cycle. The IMPlanner system includes the following modules: feature modeling, feature recognition, process selection, machine selection, process selection, process data exchange and process sequencing. In this research the focus is on process data exchange. The systems used for this research aim at building a link between engineering software and enterprise software to provide platform interoperability.
1.2 Motivation

Data exchange between the 3I-PP (Integrated, Incremental and Intelligent Planner) and the knowledge-based system is carried out by a Java Application as explained in [6]. The IMPlanner system uses a knowledge-based system developed in LISP/CLOS to generate process plans based on the feature data extracted from the CAD System. The visual representation of the processes is handled by the process visualization module, as described in [1]. However, the disadvantage of the current implementation is that it does not provide means to link the process data generated by the knowledge-based system with manufacturing environment of the existing CAD/CAM software packages. In order to utilize the existing capabilities of the modern CAD/CAM packages this research aims at integrating the process data generated for features by knowledge-based system with the manufacturing environment provided by the CAD/CAM modeling software.

The proposed approach of this research provides flexibility during the product development cycle in the early development phases and fully supports the concept of design-for-manufacturability. Java language has many powerful features that make it suitable for application data exchange. The application program interface of the CAD/CAM package Unigraphics is built in "C" language. In order to map the process data from the Java applications with other native application Java supports Java Native Interface (JNI) language.
1.3 Need for JNI and XML for Process Data exchange

The 3I-PP knowledge-based system extracts the feature-based information from Unigraphics, a modeling environment. The feature-based data for a part model is read by the 3I-PP knowledge-based system, which then generates processes corresponding to these features. This process information is then sent to a Java application using XML as data exchange mechanism. The data in the XML file is represented by a series of tags that corresponds to the process objects and their attributes. The knowledge-based system provides several process attributes can be mapped along with the process objects to the Unigraphics manufacturing environment.

The process data is stored as an XML file by the Java application. Data once stored using XML can be used by almost every application that needs the process information. This process data can be read by a Java application and mapped to the Unigraphics manufacturing environment. The process data cannot be transferred physically owing to the partial implementation of the process data generation mechanism of the IMPlanner system for only one process. However, conceptually the process data transfer can be achieved once the information for all the processes corresponding to different features is generated by the IMPlanner system. The Java application then reads the newly stored XML data and maps the equivalent process objects to operations inside the Unigraphics manufacturing environment.
Java provides native application interaction through JNI language. This is a powerful feature that allows heterogeneous applications to be interoperable across a distributed environment for exchange of process data at the same time using the capabilities of the knowledge-based system to generate processes based on the feature selection. Thus JNI and XML provide interoperability between distributed applications to provide data integration.

1.4 Problems in Process Data exchange

Currently it is difficult to find a tool that allows data exchange between CAD/CAM applications and the knowledge-based system and at the same time provides capability for visualizing the actual process inside the manufacturing environment provided by the CAD/CAM software. The IMPlanner system consists of different modules that allow data exchange between a part file and a Java application by passing the feature information to the 3I-PP and the knowledge-based system, which generates processes corresponding to each feature. The process data is then passed to the Java side using sockets. The IMPlanner system currently generates information for only one process.

Another limitation with the current data exchange mechanism is that it is not integrated with the manufacturing environment provided by the CAD/CAM package. This would otherwise provide more flexibility in controlling the process inside the CAD/CAM environment and there is not control on the process attributes.
1.5 Research Goal

The goal of this thesis is to develop a data exchange mechanism that is capable of transferring process data from a Java application to any CAD/CAM system and vice versa. Currently the process generation module generates processes by the data exchange between the 3I-PP system and the Java application. The data would be stored on the Java application side and this would allow the process data to be mapped to native CAD/CAM applications. The data exchange mechanism would represent the process mapping module under the IMPlanner system.

The process mapping module would allow remote applications to interact and exchange process-specific information which would allow applications to be interoperable and interact with any native application to use the capabilities of the manufacturing environment provided by modern CAD/CAM software packages. Also this would facilitate two-way interactions in a distributed environment adding to increased flexibility.

1.6 Summary of Process Data exchange

When a part file is selected by the Java process-plan application, feature objects corresponding to the features in the part file are created on the Java application side. Upon selection, a feature object generates a process on the Java client application side. On selecting the process that gets generated it will map to an equivalent operation inside the Unigraphics environment based on matching process attributes.
The Process mapping module allows the data to be exchanged between, the Java application and the UG/Open Application Programming Interface (API). This is made possible by using JNI. The Java Native Interface allows the selected process object to make native method calls from the Java client application by passing the process object as an argument. As a result of this data exchange, the attributes necessary to create an operation corresponding to a particular feature become available inside the Unigraphics manufacturing environment. JNI allows the Java client application to request all the data corresponding to a specific operation from the Unigraphics environment. This allows controlling the parameters for a specific operation by querying the information back from the Unigraphics environment. This data exchange capability leads to distributed application development where the Java client application can interact with other CAD/CAM applications owing to its platform independence and leads to rapid application development at the same time providing flexibility for use.

The processes generated corresponding to the feature object that is selected are stored in an XML file. An XML writer is implemented on the Java client side that will store all the process information for the feature file that is selected by the user. Once the process data is stored in XML format it can be again retrieved by selecting the same file from the Java client application. This XML file is then read using a XML reader. The same file can be reloaded to make the process information available to the user, thereby allowing the user to control the Unigraphics environment by mapping processes based on requirements.
1.7 Outline of Thesis

The thesis is divided into 7 chapters. Chapter 1 presents an introduction to topics included in this research work. It addresses motivation for the thesis, need for process-data exchange, difficulties in process-data exchange, research goals and a brief summary of the process-data exchange mechanism.

Chapter 2 provides the background related work to this research and provides an overview of the recent research work done in areas covered by this thesis. It also explains the Unigraphics Application Programming Interface (UG /Open API) in particular.

Chapter 3 presents the architecture of the current IMPlanner system. It explains in detail about the architecture used to develop the process-data exchange.

Chapter 4 and 5 discuss more about the methodology and how its implementation was carried out on the IMPlanner system.

Chapter 6 describes about the test cases on which the implementation was verified.

Chapter 7 provides the contributions, limitations and directions for the future work in this area.
2 BACKGROUND WORK

This section reviews the current research work done in the field of process data-exchange. The first part of this chapter focuses on Process Specification Language (PSL) for data-exchange environment. The next part of this chapter emphasizes the use of Extensible Markup Language (XML) and Unigraphics application programming interface (UG/Open API) for data-exchange. This is followed by the research work done in distributed environment using Common Object Request Broker Architecture (CORBA). This chapter also focuses on the use of Java Native Interface (JNI) for process data-exchange and then discusses in detail, the terminologies and terms used in this thesis research.

2.1 Process Specification Language

Manufacturing applications today are required to communicate with each other to share information. One of the major obstacles for all enterprises is the lack of means to share information between applications. The problem is more acute in a heterogeneous environment where each application is from a different domain. For example, the knowledge-based system for the IMPlanner system built in LISP interacts with the Java process-plan application, which, in turn, exchanges data with the API of Unigraphics that is built in “C” language. The major challenge to overcoming this problem is having a translator that is capable of interpreting information from one application to another. This requires the use of a separate
translator between every application, which adds to the complexity of the data exchange environment.

Ciocoiu, Gruninger, and Nau [2] stress the importance of semantic and syntactic translation between applications that can be achieved by developing sharable and re-useable models known as ontologies. Ontologies consist of terminology, along with its associated semantics, that facilitate the translation between applications. For example, a term “resource” can be interpreted in a different way by two different applications [22] as machine or manpower. This requires proper semantics to be associated along with the syntactic representation. The significance of re-usable models is that models associated with similar application requirements are similar.

Ciocoiu, et al. [2] describe the use of ontology to facilitate interoperability by applying it to enterprise modeling. The paper describes a pilot implementation that uses a common ontology as a translator for exchanging process information between two different manufacturing applications: ProCAP, a process-modeling tool based on the IDEF3 method of systems modeling; and ILOG, A C++ library for constraint-based scheduling by using Process Specification Language (PSL) as a means of data exchange. This paper describes a scenario where the planning and scheduling departments in an organization share information using PSL for integrating applications. PSL aims at providing a complete and correct exchange of process information among manufacturing applications.
The PSL ontology is based on Knowledge Interchange Format (KIF). To solve the problem of data exchange, a translator was created between IDEF3, ProCAP’s modeling language, and PSL and another between PSL and the ILOG Scheduler. The process-plan from the ProCAP tool is converted into PSL; the scheduler then imports the process-plan from PSL format into the ILOG scheduler and optimizes the plan inside the scheduler. An example of this is Figure 2-1 the duration of an activity object.

```
(defrelation duration ( ?a ?d)
  (forall(?t1 ?t2)
   (=>(and(=?t1(Beginof ?a))
     (=?t2(Endof ?a)))
     (=?d(time_minus>t2?t1)))))
```

Figure 2-1 Example using PSL for defining an activity object

activity is the difference between its start and end times for all occurrences of the activity.

The ilcActivity concept in ILOG maps to the activity concept in PSL only if activity is both primitive and nondet_res_activity, as shown in Figure 2-2.

```
(forall (?a)
  (=> (and (nondet_res_activity ?a)
           (primitive ?a))
       (<= ilcActivity a (activity ?a))))
```

Figure 2-2 Example of translation to ILOG using PSL

This approach uses PSL as a medium of communication rather than a direct translator. As a result, once the information is translated into PSL it can be
translated to any other application. Schlenoff, et al. [13] focus on the concepts in
PSL based on the KIF. They describe the PSL ontology to be two-tiered, where
the process related concepts common to all the manufacturing applications are
classified into the first tier and the concepts that form the core of PSL ontology are
the resources, process and time intervals. This ontology allows for exchanging
only basic process information. The extension of this ontology represents the
second tier that provides robustness to the ontology. For example a translation
mechanism through PSL between two applications A and B, is described in [13],
Where Application A’s terminology and ontology are shown in Figure 2-3.

Terminology:
{resource: inject_mold(x)}
Ontology
( < = > (resource@a ?r)
  (exists (?a)
   (reusable ?r ?a))

Figure 2-3 Application A’s terminology and ontology

It defines individual r as a machine if there exists some activity for which it is
reusable. Similarly, Application B’s terminology and Ontology are shown in Figure
2-4. A resource ?r is a machine tool if there exists an activity ?a. If any other
activity which also requires the resource ?r it is not possible to perform the activity
until activity ?a completes its occurrence.
Terminology:
(define-class inject_mold
  (subclass-of machine-tool))

Ontology
( <= > (machine-tool ?r)
  (exists (?a)
    (forall(?ap ?s ?sp)
      (=> (and (common ?a ?ap ?r)
          (poss ?a ?s)
          (poss ?ap ?s)
          (Do ?a ?s ?sp)))
      (poss ?ap ?sp)))))

Figure 2-4 Application B’s terminology and ontology

The neutral representation between the application A and B is provided using PSL, as shown in Figure 2-5. It defines a resource ?r as a machine if there exists some activity for which it is reusable.

( <= > (machine ?r ) a ?r)
  exists (?a)
  (reusable ?r ?a)))

Figure 2-5 PSL ontology

The translators written using PSL may not be accurate for semantic descriptions, which can lead to ambiguity. Also, PSL works with only those languages that are PSL compliant.

Lubell [10] describes an approach for process-data exchange using PSL and Extensible Markup Language (XML), which represents data in a structured
manner. When coupled with PSL, it can provide the semantic and structural representation of process data.

This provides a natural advantage as XML is growing as a popular means of data exchange. However, the drawback of XML is that it does not provide a means for including the process constraints like starting and ending times of an activity along with PSL for complete process representation.

2.2 XML Overview

XML was originally developed by a working group under the World Wide Web Consortium (W3C) in 1996. This specification [3], together with associated standards (Unicode and ISO/IEC 10646 for characters, Internet RFC 1766 for language identification tags), provides all the information necessary to understand XML representation language.

XML is a standard, simple, and self-describing way of encoding both text and data so that content can be processed with little human intervention and can be exchanged across any operating system, hardware, or applications. XML is not really an independent markup language but rather a meta language [23] [12] that enables users to build their own markup languages. XML merely provides a linguistic framework that permits the extremely diverse applications integrated with it to understand one another.

XML supports multilingual documents and the Unicode standard. Tags, attributes, and element structure provide context information that can be used to interpret the meaning of content, thereby facilitating the process of data exchange.
One of the benefits of using XML is it can embed multiple data types and also separate the content from the presentation. This gives user the flexibility to present data in any format.

The strength of XML lies in its ability to exchange data between any kind of applications [1]. As different applications rarely use the same set of tools to exchange data with XML in place, the data can be transferred between applications without losing any information. XML is capable of moving the data across networks by encapsulating the data into a series of custom tags by forming a structured document and includes the semantic information inside the tags. The applications can easily interpret this XML data and present it to the user in a specific format. Thus, it can be said the XML is widely used for data exchange.

2.2.1 Structure of XML document

An XML document consists of elements and attributes. The first line in XML document represents the XML version in use, as shown in Figure 2-6.

```
<?xml version = “1.0”?>
<BankAccount>
   <Account type = “checking”>
      <Number>1234</Number>
      <Name>John Mayor</Name>
      <Balance>2345</Balance>
   </Account>
   <Account type = “saving”>
      <Number>4567</Number>
      <Name>John Mayor</Name>
      <Balance>3435</Balance>
   </Account>
</BankAccount>
```

Figure 2-6 Structure of an XML file
The next line represents a comment inside the XML document. The next line is the root element of the XML document; i.e., `<BankAccount>`. The next line contains the child element `<Account>` with an attribute `type`. The `type` attribute gives information about the type of `Account`. The next three lines contain the child elements of the Account element, as shown in Figure 2-7. The next line contains the closing `</Account>` element. The next line starts a new child element under the parent `<BankAccount>` element.

```
<Number>4567</Number>
<Name>John Mayor</Name>
<Balance>3435</Balance>
```

Figure 2-7 Elements of an XML file

The next line contains again the `<Account>` element followed by its child elements. Finally the document ends with closing tags for the `</Account>` and `</BankAccount>` element.

### 2.2.2 Components of an XML document

An XML document contains markup and content. The markup provides meaningful representation of the content. The markup is like metadata that describes the content. Elements and attributes constitute the markup part of the XML document. An element or attributes is the key indicator of the structure of the document. For example, the element `Account` has the attribute as `type`. The element can have more than one element inside its tag. These elements represent the child elements of the given element, as shown in Figure 2-6. In order for the
document to be structured, every starting tag has a corresponding ending tag that begins with the character ‘/’.

2.3 UG / Open Application Programming Interface

Unigraphics is one of the most powerful CAD/CAM packages that provide the user with the flexibility to import any product format and visualize the actual manufacturing process. One of Unigraphic’s greatest strengths has always been its integration of CAD and CAM. Unigraphics provides an open application programming interface (UG/Open) [21] which allows almost any application to control Unigraphics from outside its environment.

UG/Open is a product name that encompasses the flexible integration of many different software applications with Unigraphics through an open architecture. UG/Open, by design, focuses on enabling open architectures to be utilized by third parties, customers, and in-house users when integrating or customizing software applications [21]. UG/Open provides the applications and tools that enable customers to interface with the Unigraphics Object Model through UG/Open API; to create and manipulate user-defined objects (also referred to as custom objects), including managing their associability with other UG objects and providing methods for updating and displaying the user defined objects; customize the UG graphical interface to reflect third party applications; and to utilize and integrate new UG/Open technologies as they become available.
2.3.1 Unigraphics manufacturing environment

The manufacturing mode in Unigraphics requires a part model file to be initialized to the machining environment. A Unigraphics part file that contains a block with a slot is shown in Figure 2-8. The slot on the given component can be generated through a milling operation. This requires the CAM Session Configuration to be set to mill_planar.

Figure 2-8 Sample Unigraphics file with a block

The configuration files define the CAM working environment. They also make available the entire tool libraries and high-level parameters required for the session. When the session is configured to mill_planar; i.e., mill_planar.dat file, it calls the mill_planar.opt file which loads the operation templates for mill_planar and drill corresponding to mill_planar.prt and drill.prt files, as shown in Figure 2-9.

Figure 2-9 Loading of Unigraphics templates for initializing a CAM session
As a result of this, the CAM setups that become available are mill_planar and drill. The CAM General configuration includes the mill_planar, drill, mill_multi-axis, and mill_contour .dat files. Unigraphics can be initialized to any of these setups. Setups are Template Part Files that contain templates that define the available type options and groups that initially appear in the Operation Navigator, the Create Operation dialog, and the Create Group dialog.

The Create Operation dialog for the part model file in Figure 2-10 prompts the user to input data corresponding to the program, method, geometry, and tool groups.

![Create Operation](image)

Figure 2-10 Dialog for CreateOperation in Unigraphics

*Operation Navigation Tool (ONT)* is a graphical user interface that allows managing operations and operation parameters for a given part model. The ONT uses a tree structure to show the relationship between groups and operations. All
the operations required for machining can be grouped into a Program Group. The Program Order View of the ONT shows to which program group each operation belongs.

The operations are defined in the Program group based on the machining sequence. The Method Group organizes the operations under common machining applications, which share the same parameter values; e.g., rough, semi-finish, finish. The Geometry Group shows the machining geometry and machine coordinate system (MCS) each operation will use. The Tool Group organizes the operation based on the machine tools used. The four groups that relate to an operation are shown in Figure 2-11.

![Figure 2-11 Groups related to a Unigraphics operation](image)

The user has to input all the operation parameters manually to create an operation inside the Unigraphics environment. This puts significant overhead on the user to specify all the required parameters like processing time, feed, speed,
etc. This is one of the major limitations in generating operations inside Unigraphics. In order to provide the ability to the user to control Unigraphics from outside its environment, the concept of native application integration was sought. The approach that has been taken in this thesis is to integrate the native CAD System environment with the Java application, as Java strongly supports native application integration owing to its platform-independent feature. This will allow more control from the user perspective as the user can go back and forth between the modeling and manufacturing environment and simulate the entire machining operations and create operations. This approach aims at using the capability of the existing CAD/CAM package to generate and view operations within the CAD/CAM package.

2.4 Previous Work in Process-data exchange

Data exchange between applications is carried out by means of files interface (one module writes and the other reads it), database interface (the modules access the database), object interface (the modules exchange objects using object exchange protocol like CORBA or COM), or XML interface (the modules exchange tagged text streams).

Šormaz, Jain, and Borse [16] [17] describe an automated data exchange mechanism that provides data mobility between disparate and heterogeneous systems leading to an integrated manufacturing system. The process of data exchange in [16] [19] is between a 3I-PP knowledge-based system and a Java application using XML as a medium for data exchange. The IMPlanner system
used in this process of data exchange consists of various modules like process selection and feature extraction. The client makes a request to load the knowledge base into 3I-PP and the IMPlanner system sends a request to the server to load the feature data into its environment. The IMPlanner system requests the feature data from the 3I-PP system, which generates an XML Stream and sends the feature data to the client. When the client selects a feature for which to generate a process, a request is sent back to the knowledge-based system [20], which generates the alternative processes and sends them back to the client.

The knowledge-based system that is used for the process of data exchange is implemented in LISP/CLOS. All the process information from the LISP server is sent to the Java client over the network using XML streams and sockets [6]. The Java client reads the XML stream and interprets the process information. The limitation of the current approach is that it does not use the capabilities of a modern CAD system to control the design for manufacturing, as the user can simulate the actual manufacturing operation within the CAD/CAM environment. One of the goals of this thesis is extending the approach discussed in papers [14] [16] to provide a means to store process-specific information locally on the client side.

The visualization of manufacturing processes in the IMPlanner system is described in papers [14] [18]. The framework for visualizing the process makes use of Java and Java3d APIs. The data structures from the Java classes (for
feature, process, and tool) are converted into geometric objects using Java3d utilities. The Java3d environment allows visualizing the actual machining process on a Java applet. The process visualization is made possible by the process data (feed, speed), which provide the necessary information for time-dependent geometry and time interpolation. This architecture provides the flexibility to visualize every individual component in 3d animation and in 3d wire frame model. Though this architecture allows visualizing of the actual manufacturing process in a simulated environment, it places significant overhead on the process visualization interface. An alternative to the approach is to use the capabilities provided by the modern CAM/CAM applications, like different file formats supported, to design and manufacture a product. This is one of the major areas of focus for this thesis. The idea is to use the capabilities provided by a CAD/CAM package like Unigraphics and provide the user flexibility and control between the design and manufacturing process. In this approach, a user can have greater control in selecting the operations that are to be performed on a specific feature.

2.5 Java Native Interface

Applications today require data to be exchanged between them freely without being bound to the application that creates the data. Java Native Interface (JNI) provides an approach towards this end. JNI is a standard programming interface [4] [5] for writing Java native methods and embedding the native applications into Java virtual machine. The JNI allows Java code that runs within a Java virtual
machine (VM) to operate with applications and libraries written in other languages, such as C or C++. This ensures that the Java code is portable across all platforms.

The JNI framework allows the use of native methods to do many operations. Native method may represent legacy applications or they may be written for solving a particular problem, which can best be handled outside the Java environment. The advantage of using a native method is that it can easily update Java objects and make them available to the Java application. The native method uses the JNI framework to call a Java method to pass required parameters and returns the results back when the method completes, as shown in Figure 2-12.

![Figure 2-12 JNI mapping](image)

**2.5.1 JNI components**

The components of a Java Native Interface Application are Java native methods represented by the `.java` file, the Java header file (`.h` file) and the Java native method implementation (`.c` file). The Java application defines the native
methods which are invoked at run time by loading the native applications dynamic
linking library (.dll file). The .dll file contains the actual method implementation of
the native methods.

2.5.2 JNI structure

A simple Java application that accesses a native method [7] is shown in Figure
2-13. The first line defines the name of the Java class HelloWorld. The next line
defines the native method displayHelloWorld().

class HelloWorld {
    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }

    Figure 2-13 Sample Java class calling a native method

The keyword “native” is used as a part of the method definition that tells the
Java compiler that the method displayHelloWorld is a native language method.
The implementation of this method is done in a separate source file.

The next line, loadLibrary ("hello"), loads the shared library into the Java class.
The shared library is created by compiling the native language code that
implements the displayHelloWorld method. The next line contains the main
method. Inside the main method an instance of the HelloWorld class is created.
This instance is used to call the native method displayHelloWorld.
The Java application is compiled to generate a class file. A Javah utility is run on the compiled Java class to generate a header file for native applications. The native method displayHelloWorld within the HelloWorld class becomes Java_HelloWorld_displayHelloWorld, as shown in Figure 2-14. The Java_HelloWorld_displayHelloWorld method provides the implementation for the HelloWorld class's native method displayHelloWorld.

```
#include <jni.h>
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    displayHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Figure 2-14 Header file for the HelloWorld class

The implementation of the native method accepts two parameters. The first parameter is JNIIEnv env interface pointer. This parameter is used to access the objects and parameter passed to it from the Java application. The second parameter jobject obj references the current instance of the Java object. Figure
2-15 shows the implementation of the native method in “C” language. The “C” source file contains the actual implementation of the native method display *HelloWorld*.

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

Figure 2-15 native implementation of displayHelloWorld method

The displayHelloWorld method prints a “Hello World” statement when the *HelloWorld* Java class is run. The first line in the “C” source file contains the header file `<jni.h>`. This file provides the information, which a native language code requires to interact with Java environment. The next line contains the *HelloWorld.h* file, which was generated earlier by running the Javah utility on the *HelloWorld* class file. The include file `<stdio.h>` is used to provide access to `printf` function from the `<stdio>` library.

### 2.5.3 Distributed computing environment with CORBA

An important feature of large computer networks such as the World Wide Web or Corporate intranets is that they are heterogeneous. The Common Object Request Broker Architecture (CORBA) is an older middleware technology that competes with Enterprise Java Beans (EJB), which has its own architecture.
CORBA is a standard proposed by the object management group [8] in order to promote interoperability between distributed object systems. CORBA is a useful platform for World Wide Web-based applications because it abstracts many of the inherent complexities of distributed applications. Also, CORBA provides a programming-language-neutral interface that describes the syntactic aspects of the services supported by remote objects. CORBA is not merely a manufacturing-specific standard. It is a complete distributed object computing framework that extends applications across networks, languages, component boundaries and operating systems.

Vinoski [22] describes the Object Management Group (OMG) architecture that is used for integrating diverse applications within a distributed environment. This architecture focuses on extracting common functionality from CORBA applications into a set of standard objects that perform clearly defined functions accessed through standardized OMG interface definition language (IDL). The Object Management Architecture (OMA) uses the Object Model to define objects that are distributed across a heterogeneous environment and it also uses a Reference model to characterize interactions between objects. The OMA model in [22] describes an object as an immutable entity whose services can be accessed through an interface. Clients issue requests to objects to perform services on their behalf. The actual implementation objects requested by the client are hidden from the requesting client. Clients make object requests by using an object reference. These object references have standardized formats as those of OMG standard
Internet Inter-ORB Protocol and Distributed Computing Environment Common Inter-ORB Protocol. CORBA uses Object Request Broker (ORB) to communicate from one ORB to another.

An important feature of OMG IDL, as described in [22], is language independence. OMG IDL is a declarative language that allows interfaces to be defined separately from object implementations. This allows objects from different languages to communicate with each other in a heterogeneous environment. OMG IDL mappings allow CORBA to meet the requirements of the real-world implementation. CORBA requires an out-of-process remote call which slows down performance. Also it has interoperability problems among vendor ORB’s and defines more services than vendors usually support.
3 IMPLANNER SYSTEM ARCHITECTURE

The IMPlanner system [20] has modules for process selection, feature modeling, process plan, and process visualization in the manufacturing domain. The significance of the IMPlanner system Figure 3-1 is that it allows data to be exchanged in a distributed environment among heterogeneous applications.

Figure 3-1 IMPlanner system architecture
The main features provided by the IMPlanner system are object-oriented environment, distributed applications, platform independence, universal standard for data exchange, and remote processing. The IMPlanner system uses standard applications like CAD/CAM package and 3I-PP process planner, and the data exchange mechanism is carried out by means of software tools like Java, Lisp, XML, and JNI.

The 3I-PP process planner as described in [9] exchanges data with the visualization module. The 3I-PP system is a knowledge-based application that integrates the CAPP and provides information about process selection, machine and tool constraints, and cutting parameter selection. Feature extraction of a geometric component is performed by the user and entered into the system. Information about a feature is retrieved by the feature-based module, which, in turn, exchanges data with the 3I-PP process planner through a 3I-PP schema.

The 3I-PP system generates processes for the features, which are represented in the 3I-PP schema. Data is exchanged between the 3I-PP system and the process-plan module in a client server environment where the 3I-PP knowledge-based system represents the server and the process-plan module acts as a client that initiates a request to the server. The process-plan model interacts with the process-plan network module to generate a process network.

The visualization module provides the capability to animate the actual machining process in a Java3d environment. The visualization system interacts with the user by animating the feature generation process on the component. This
virtual machining environment provides visualization of components like work-piece geometry before and after machining, tool path, tool, and slot feature.

### 3.1 Process-data exchange Architecture

The architecture, as shown in Figure 3-2, has been implemented in this thesis is an extension of the IMPlanner system described in [16] [20].

![Figure 3-2 Process-data exchange architecture](image)

Data exchange takes place when the application interface corresponding to the process-modeling module interacts with the Java process-plan model. Upon receiving a request from the Java application interface to generate an operation
inside the Unigraphics environment, the Java process-plan model interacts with the *MfgProcess* class to map the selected process to an equivalent operation inside the Unigraphics environment. The *MfgProcess* application, as a result of mapping, creates a Unigraphics-referenced Java object. Communication with the Unigraphics environment is achieved by means of native application calls made by the Java process-plan application with the UG/Open interface. As a result of this, the mapped objects are passed as arguments to the native application calls. The JNI interface links the Java process-plan model with the UG manufacturing application using UG/Open API.

The data corresponding to specific operations gets transferred finally into the Unigraphics environment. This causes operations to be created in the Unigraphics environment. The data exchange between the process-plan modeling interface and the Unigraphics environment is possible in both directions; i.e., the Java application interface can also query the information corresponding to the CAM operations inside the Unigraphics environment. This feature gives more control to the user as this allows a user to visualize the actual machining operation inside the Unigraphics environment and, at the same time, gives the user the flexibility to vary different parameters, which would affect the final machining operation. This approach allows the user to utilize the capabilities of both the modeling and manufacturing environment provided by the Unigraphics. The Java application interface also provides the capability to store the process-plan information along with their associated features and processes. The process attributes are also
stored. Process data storage is achieved when the Java application interface writes the process data using an XML writer object provided by the MfgProcess. This results in the process data being stored in an XML format. Reading of an XML file is done by means of a XML reader object provided by the MfgProcess instance.
4 METHODOLOGY

4.1 Process Data exchange Approach

The approach that was taken for this thesis was to provide data exchange between a Java application and Unigraphics manufacturing application. There were two possible approaches that were identified that would allow this data exchange process. The first approach was to interact with both the modeling and machining environment of Unigraphics from the process-plan model. However, the drawback of this approach was that the application programming interface, which links the geometry to the operations, was not available for Unigraphics environment. The second approach was to interact with the manufacturing environment of Unigraphics. As the application programming interface was provided for the manufacturing environment, the second approach was used for this thesis. In addition, XML was adopted for saving process data for the process-modeling application. XML was chosen mainly because of its universal vendor support and platform independent features. Data exchange between the Java application interface for the process-plan modeling module and CAD/CAM application was achieved in four different stages:

1. Querying CAM data
2. Process mapping
3. Creating CAM data
4. Saving process into XML
The different stages involved in the process data exchange mechanism are explained in this chapter.

4.2 Querying CAM Data

The first step that was identified to exchange data between the Java application interface and the Unigraphics manufacturing application was to test the ability to interact with the native environment of Unigraphics from the Java application interface. In order to provide interaction between Java application and the native Unigraphics environment, Java’s native integration technology, JNI, was chosen as it allows Java to interact with any application running on a different platform. The feasibility of this approach was tested by querying the UG/Open API from the Java application. UG/Open API for Unigraphics is built in “C” language; this requires Java objects to be referenced from the native “C” application. This two-way interaction between Java and UG/Open API provides better interoperability between native applications.

4.2.1 Sample CAM Query Output

The result of querying the Unigraphics environment for a part model from the Java Application interface is shown in Figure 4-1. It represents the geometric data that was queried from Unigraphics part file. This data corresponds to the Program, Method, Machine Tool and Cutter group, their members, operation types, and subtypes. Based on the results of this approach, JNI technology was selected for data exchange between Java and Unigraphics environment, as shown in Figure 4-2.
Each operation in Unigraphics is associated with a program, machine tool, method, and cutter group. An operation contains all the information required to generate a tool path. The operations in Unigraphics are based on a fixed set of templates corresponding to a given setup (machining environment). For example, a milling setup loads all the predefined templates of the milling operation type. The output in Figure 4-1 shows the different parameters associated with an operation inside Unigraphics. The operation name parameter describes the name of the operation inside the Unigraphics environment; e.g., PLANAR_MILL. Each
operation is associated with an operation tag to uniquely identify an object inside Unigraphics environment. The parameter operation type refers to the category of operation like mill_planar, drill, mill_contour and mill_multiaxis. The next parameter, GEOMETRY GROUP, defines the machining geometry and the orientation of the part on the machine tool based on the CAM setup. The different types of geometry groups in Unigraphics are Machine Coordinate System (MCS), mill geometry or workpiece, mill boundary and mill area. The GEOMETRY GROUP is identified by a unique tag number corresponding to different geometric groups. The next parameter, member name, refers to the operations within a specific geometric group; e.g., PLANAR_MILL and FINISH_WALLS.

The parameter PROGRAM GROUP allows grouping and ordering operations into programs. This allows new PROGRAM GROUP’s to be created. All operations required to machine the top face of a part can compose a single program group. The order of operations within the PROGRAM GROUP indicates the actual machining order of the operations on a given part like ROUGH_WALLS and FINISH_WALLS. Each PROGRAM GROUP is also identified by a unique tag number in the Unigraphics environment. The next parameter, METHOD GROUP, defines the cutting method for a specific operation like ROUGH, SEMI_FINISH, and FINISH. The number of members parameter lists the total number of members for a specific group. The CUTTER GROUP organizes operations based on the cutting tools. The CUTTER GROUP is identified by a unique tag number.
4.3 Process Mapping

The MfgProcess application in Java is associated with different machining processes. The IMPlanner system currently generates different processes as a result of data exchange with the 3I-PP system. In order to make the information about the processes available inside Unigraphics, it was necessary to map the existing MfgProcess from the IMPlanner system to equivalent Unigraphics operations.

The processes in the IMPlanner system had to be represented as operations and, moreover, the processes in the IMPlanner system are Java objects, whereas the operations are represented by tags in Unigraphics environment. Also, the processes in IMPlanner system were represented by different names compared to the operations in Unigraphics. The approach that was taken to map these processes involved identifying classifications of processes having common machining parameters as the operations in Unigraphics. An example of this is shown in Table 4-1 where the FaceMilling process on the Java side gets mapped to FACE_MILLING operation inside Unigraphics. The difficulty involved in mapping processes to operations is it is not very clear which process maps to which operation for processes like EndMilling, EndMillingSlotting and EndMillingPeripheral. These processes can be a combination of any of the Unigraphics milling operations like PLANAR_MILLING, PROFILE_MILLING and ROUGH or FINISH_WALLS.
The mapping procedure involves identifying all the operations under different templates in the Unigraphics environment. Templates are predefined operations or groups within a CAMPartModel. All the operations from different Unigraphics templates are classified based on the operation types like mill_planar and drill. The MfgProcess can also be associated with more than one operation in Unigraphics. For example, the EndMillingPeripheral machining process can be defined as a combination of FINISH_WALLS and FINISH_FLOOR in terms of Unigraphics operations.

Table 4-1  Process mapping

<table>
<thead>
<tr>
<th>Java Processes</th>
<th>Mapped UG Processes</th>
<th>mill_planar</th>
<th>drill</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreMaking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HoleImproving</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HoleMaking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HoleStarting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Milling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drilling</td>
<td>DRILLING</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Boring</td>
<td>BORING</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CounterBoring</td>
<td>COUNTERBORING</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>EndMillingPeripheral</td>
<td>FINISH_WALLS</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EndMilling</td>
<td>FINISH_FLOOR</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EndMillingSlotting</td>
<td>CLEANUP_CORNERS</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PLANAR_MILL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PLANAR_PROFILE</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROUGH_FOLLOW</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROUGH_ZIGZAG</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROUGH_ZIG</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FaceMilling</td>
<td>FACE_MILLING</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FinishMilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RoughMilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SemiFinishMilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reaming</td>
<td>REAMING</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>EndDrilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CenterDrilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Lathe)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrecisionBoring</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>Name</td>
<td>Status</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>SideMilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SlabMilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpadeDrilling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpotDrilling</td>
<td>SPOT_DRILLING</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPOT_FACING</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling</td>
<td>PECK_DRILLING</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BREAKCHIP_DRILLING</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COUNTERSINKING</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TAPPING</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>THREAD_MILLING</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CAVITY_MILL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ZLEVEL_FOLLOW_CAVITY</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ZLEVEL_FOLLOW_CORE</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ZLEVEL_ZIGZAG</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ZLEVEL_PROFILE</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ZLEVEL_PROFILE_STEEP</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FIXED_CONTOUR</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTOUR_ZIGZAG</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTOUR_AREA</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTOUR_NON_STEEP</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTOUR_FOLLOW</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTOUR_SURFACE_AREA</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTOUR_DIR_STEEP</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOWCUT_SINGLE</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOWCUT_MULTIPLE</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOWCUT_REF_TOOL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARIABLE_CONTOUR</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VC_MULTI_DEPTH</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VC_BOUNDARY ZZ_LEAD_LAG</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VC_SURF_AREA ZZ_LEAD_LAG</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FIXED_CONTOUR</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL_MILL</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ZIG_ZAG_SURFACE</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

In this way the manufacturing processes (*MfgProcess*) get represented as operations. Also, the operations that are created in Unigraphics will have the same nomenclature as the actual operations within the Unigraphics environment. These
Unigraphics-referenced objects or operations are then called based on the \textit{MfgProcess} that is selected at the Process Plan modeling interface. These Unigraphics-referenced objects on the Java application side then call a native function for creating an operation from the “C” source file, which interacts with UG/Open.

4.4 Creating CAM Data

Once the mapping of processes to operations was done, the next stage was to create operations inside Unigraphics. The feasibility of this approach was tested by interacting with the “C” application for the UG/Open API directly. This involved creating an operation in Unigraphics by specifying the operation name at run time for the “C” application. The Unigraphics environment was then checked for this operation based on the results of this pilot implementation. JNI was used to interact with the Java application with the “C” application, as shown in Figure 4-3.

![Figure 4-3 CAM data generation](image-url)
The *MfgProcess* class interacts with the UG/Open API whenever the user selects a feature from the process-modeling interface to generate an operation. Each feature for a part may be associated with one or more processes. The process selected by the user at the process-modeling interface gets mapped as a Unigraphics referenced object. This operation is then passed as an argument to the appropriate native functions. As a result of this, the “C” application has a reference to the actual operation that has to be created in the Unigraphics environment. These Unigraphics-referencing objects on the JNI implementation side finally generate an operation inside Unigraphics by invoking the UG/Open API.

The Unigraphics environment requires a part model to be initialized to the manufacturing environment for loading the libraries and supporting templates for the manufacturing environment. Only after the Unigraphics environment is changed from the modeling to the manufacturing mode can the operations be generated. The status of the operation generated can be known on the Java side by querying the Unigraphics environment at this point. The Unigraphics environment is then queried from the “Java-application side” to get the details of all the operations inside the Unigraphics environment. As a result, the processes selected at the user interface finally generate an equivalent Unigraphics operation. The operation that is created can be seen in the Operation Navigator Tree (ONT) shown in Figure 4-4.
4.5 Saving and Retrieving Process in XML

The advantage of using XML for storing process information is its wide acceptance as a standard for data exchange. XML describes metadata unlike data stored in a database. This is a useful feature that describes the data in detail with its attributes. These features make XML preferable over any other storage mechanism. The Java process-plan application should be able to retrieve process information from the XML file. This required an XML parser to be used for reading the XML document, as shown in Figure 4-5.
The XML parser was implemented inside the *MfgProcess* application. The *MfgProcess* provides content handlers for reading an XML document. The content handler inside the *MfgProcess* system reads each element of the XML document and delegates the task of parsing elements to the appropriate content handlers based on the type of element. The SAX parser allows individual parts of each document to be processed in isolation from other parts. The parser reads the XML elements Figure 4-6 and generates their equivalent Java objects.

Figure 4-6 XML document for a single manufacturing process
The XML parser reads the header of an XML document first to identify it. The XML parser will then read the first element `<edu.ohiou.implanner.process.EndMillingSlotting>` and call the content handler corresponding to the `EndMillingSlotting` process. The attributes `procName` and `processingTime` are parsed as attributes of the Java object. The XML parser reads the next `<ToolConstraint>` element and calls the SAXHandler of the `ProcessConstraint` class that handles the `ToolConstraints`. The methods `getConstraintOperator`, `getOperatorValue` are called on the constraints and assigned to Java objects. Similarly, the SaxHandlers for the Tool and CuttingParameter classes are called for parsing the Tool and Cutting parameter elements. In this way, each element in the XML document is successfully parsed.

The information pertaining to the processes in the Java application for the process-plan model has to be saved so that it can be retrieved later. This would offer an advantage in that the process information does not have to be regenerated every time.

The approach that was followed to save the process information in XML format is shown in Figure 4-7. The `MfgProcess` object represents the machining processes as Java objects. The `MfgProcess` system implements an XML writer, which generates XML documents. The XML Writer first creates an empty new XML document and attaches the XML header.
The header is the first line that gets written in an XML document. This is used to identify a XML document. The next step is generating XML elements. An XML document is composed of starting and ending tags or elements that make up the entire XML document. The Java objects are used to generate these elements. Each starting element should have an ending element for the document to be valid.

The elements that make up the XML document describe the relationships between processes, their constraints, tools, etc. As a result of this approach, it was possible to save the process information into an XML file. A sample of an XML document to store the process information is shown in Figure 4-8. An XML Writer Object is used to generate an XML document. The Writer Object writes the XML header to an empty XML file. The XML Writer identifies different object types corresponding to features, processes, etc. The elements get created based on the object types.
The XML Writer checks if the given object is an instance of *MfgFeature*. If the object type is of *MfgFeature*, then the XML writer creates an element associated with it. This element is then written to the XML document. The next line will check if the feature above has processes associated with it. The next element `<ProcessList>` gets generated in a similar manner. The process parameters are passed as attributes to the XML elements.
5 IMPLEMENTATION

The first part of this chapter describes the process-plan model and the graphical user interface that is used for process data exchange. The next part of this chapter focuses on retrieving data from the CAM system and mapping the process information to create operations in the Unigraphics environment. This chapter concludes by describing the implementation procedure for saving the process data into an XML file and retrieving process data from the stored XML file.

5.1 Process-Plan Model

The process-plan model consists of several different classes of objects. The hierarchy of all the classes responsible for the process-data-exchange mechanism is shown in Figure 5-1.

![Figure 5-1 Hierarchy of classes in the process-plan model](image-url)
The line in Figure 5-1 represents the classes implemented in this thesis for achieving the process-data-exchange mechanism. The ImpObject class is the root class for the process planning and related objects. It extends the ViewObject class which provides viewable interfaces for all its subclasses.

Class MfgPartModel contains the information for all the manufacturing features. The NetExample displays feature data by extending the MfgPartModel class. The CADSystem class provides means for linking to a Unigraphics file. The CADObject class represents the upper most class in the hierarchy of geometric features. The PartModel class contains information about geometric features and extends the CADObject class which, in turn, extends the IMPObject class. Class CAMPartModel queries and creates operations with the part file. The CAMSystem class interacts with the native dll application by calling native functions. The CAMOperation class defines attributes for operations that references CAM Operation data.

5.1.1 MfgProcess Classes

The MfgProcess class is a common class for all the manufacturing processes. It corresponds to any machining process that is used to generate a manufacturing feature. The methods that are defined in the MfgProcess class are getFeature, getFeatureName, getProcessList, getProcessTime, getSaxHandler, getTool Constraint, addInstance, writeXML. It also contains the abstract methods getCAMType, getCAMSumbType. The different classes that correspond to the manufacturing processes are explained in this section.
The **Drilling** Class is used to create a drilling process. It extends the **CoreMaking** class. Classes **EndDrilling**, **GunDrilling**, and **TwistDrilling** extend the **CoreMaking** class and are used to create **EndDrilling**, **GunDrilling**, and **TwistDrilling** process. The **TwistDrilling** class defines the method **getCAMType** and **getCAMSubType**. The **Boring** class creates a boring process and contains the method **getCAMType** and **getCAMSubtype**. It extends the class **HoleImproving**. The classes **CounterBoring**, **Honing**, **Precision Boring**, **Reaming** also extend the **HoleImproving** class and are similar to the **Boring** class. The **Reaming** and **CounterBoring** class implement the method **getCAMType** and **getCAMSubType**.

The classes **SideMilling**, **EndMillingPeripheral**, **EndMillingSlotting**, **RougMilling**, **FaceMilling**, and **FinishMilling** each extend the **Milling** class and create their respective processes. The methods **getCAMType** and **getCAMSubType** are defined for the classes **EndMillingPeripheral**, **FaceMilling**, and **FinishMilling**. The **EndMillingPeripheral** class also defines the methods **rotateTool3d**, and **setProcessConstraint**.

The **Milling** class extends the **MfgProcess** class creates a **Milling** process and also defines the method **getCAMType** and **getCAMSubType**. The **CoreMaking** class extends the **HoleMaking** class. The **Holemaking** class, in turn, extends the **MfgProcess** class.

### 5.1.2 CAMPartModel Class

The process-plan model application interacts directly with the UG/Open API. The **CAMPartModel** class is responsible for generating operations in a part file and
provides information specific to each operation. It defines the method `populateOperations` to create a list of operations from Unigraphics part file. An instance of the `CAMSystem` class is passed as an argument to the `populateOperations` method. The implementation of the `populateOperations` method is shown in Figure 5-2.

```java
public void populateOperations(CAMSystem camSystem)
{
    int oper[] = camSystem.getOperationTags(this.getObjectID());
    System.out.flush();
    for(int i=0;i<oper.length;i++)
    {
        System.out.println("tag in java is: " + oper[i]);
        CAMOperation camOperation = new CAMOperation(oper[i]);
        camSystem.getOperationDetails(camOperation);
        operations.add(camOperation);
    }
}
```

Figure 5-2 Implementation of `populateOperations` method

An integer array is used to store the list of all operation tags obtained by calling the `getOperationTags` method on an instance of `CAMSystem`. For each operation tag that is retrieved, a `CAMOperation` instance is created, and in each case the method `getOperationDetails` is called to retrieve the type and subtype of an operation. Finally, all the operations are added to a list of operations in `CAMPartModel`.

The `addOperation` method adds the operation in the `CAMPartModel` to a linked list and for each operation calls the methods `getOperationDetails` and
getOperationGroups on an instance of CAMSystem to retrieve the groups, type and subtype. The code of this method is shown in Figure 5-3.

```java
public void addOperation(int operationTag)
{
    CAMOperation cm= new CAMOperation(operationTag);
    CAMSystem cs = new CAMSystem();
    cs.getOperationDetails(cm);
    cs.getOperationGroups(cm);
    operations.add(cm);
}
```

Figure 5-3 Method addOperation in CAMPartModel

5.1.3 CAMOperation Class

The CAMOperation class creates an object operation tag that corresponds to a Unigraphics operation, which is identified, by a unique tag. This class stores the tag and, therefore, serves as a process model reference to Unigraphics operations. This class defines the attributes operationType, geomGrp, progGrp, methdGrp, and cuttrGrp corresponding to the geometry, program, method, and cutter groups. These attributes are populated by the methods in the native CAMSystem class.

5.1.4 CAMSystem Class

The CAMSystem class is responsible for interaction between other Java objects and the native application’s dynamic link library (.dll). The CAMSystem class defines all the native methods, as shown in Figure 5-4. The actual implementation of the native methods is done on the native application side in “C” language. The native methods are declared in the CAMSystem class by using the
keyword *native*. The public and private keywords in the method declarations control the user’s accessibility of these methods. The native method definitions are declared as private so that they cannot be modified by any method calls.

```java
public native int[] getOperationTags(int partTag);
private native void getOperationDetails(CAMOperation camoper, int operTag);
public void getOperationDetails(CAMOperation camoper)
{
    getOperationDetails(camoper, camoper.getObjectID());
}
private native void getOperationGroups(CAMOperation camoper, int operTag);
public void getOperationGroups(CAMOperation camoper)
{
    getOperationGroups(camoper, camoper.getObjectID());
}
private native int createOperation(String camType, String camSubType);
public int createOperation(MfgProcess process)
{
    int operationID;
    operationID = createOperation(process.getCAMType(), process.getCAMSubType());
    return operationID;
}
private native int CAMInitialize();
```

Figure 5-4 Native methods defined in *CAMSystem* class

### 5.1.5 Implementing Native Methods Using JNI

First, the *CAMSystem* class is compiled and then a *Javah* utility is run on the compiled class to generate a *CAMSystem.h* header file. This header file contains all the method declarations to be implemented in the native *dll* application.
`#include <jni.h>
/*
 * Class:     edu_ohiou_implanner_cam_CAMSystem
 * Method:    getOperationTags
 * Signature: (I)[I
 */
JNIEXPORT jIntArray JNICALL JNICALL
Java_edu_ohiou_implanner_cam_CAMSystem_getOperationTags
(JNIEnv *, jobject, jint);

/*
 * Class:     edu_ohiou_implanner_cam_CAMSystem
 * Method:    getOperationDetails
 * Signature: (Ledu/ohiou/implanner/cam/CAMOperation;I)V
 */
JNIEXPORT void JNICALL JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_getOperationDetails
(JNIEnv *, jobject, jobject, jint);

/*
 * Class:     edu_ohiou_implanner_cam_CAMSystem
 * Method:    getOperationGroups
 * Signature: (Ledu/ohiou/implanner/cam/CAMOperation;I)V
 */
JNIEXPORT void JNICALL JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_getOperationGroups
(JNIEnv *, jobject, jobject, jint);

/*
 * Class:     edu_ohiou_implanner_cam_CAMSystem
 * Method:    createOperation
 * Signature: (Ljava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_createOperation
(JNIEnv *, jobject, jstring, jstring);

/*
 * Class:     edu_ohiou_implanner_cam_CAMSystem
 * Method:    CAMInitialize
 * Signature: ()I
 */
JNIEXPORT jint JNICALL JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_CAMInitialize
(JNIEnv *, jobject);

#ifndef __cplusplus
#endif
#endif

Figure 5-5 Header file CAMSystem.h
The native functions are defined in a *dll* application for each of the function declarations in the `CAMSystem.h` file. Finally, the *dll* application is loaded inside the `CAMSystem` class, as shown in Figure 5-6.

```java
static {
    System.load("UGCamProject.dll");
}
```

Figure 5-6 Loading the native CAMSystem dll

### 5.2 Process-Data-Exchange Graphical User Interface

A process-data exchange interface, as shown in Figure 5-7 is used for exchanging process related information between the process-plan model and the
Unigraphics manufacturing application.

*NetExample* class provides the process-data-exchange applet. Components of the process data exchange applet are a *buttonPanel* that has *JButtons* for providing mouse click events to facilitate interaction with Unigraphics, a *JTree* panel, and a detail panel. Operations retrieved from Unigraphics are displayed in a *JTree* in a separate Java applet. The *buttonPanel* has the following buttons: *Open UGFile*, *OpenXmlFile*, *StartCamSession*, *SaveXmlFile*, and *CamOperations*. The buttons *OpenUGFile* and *OpenXmlFile* are used for opening Unigraphics and XML files. *SaveXmlFile* button allows the process data to be saved to an XML file and the *StartCamSession* button and *CamOperations* button are used for initializing a session and displaying the operations obtained from the Unigraphics part file.

When the process-data-exchange application is run for the first time, the buttons *StartCamSession*, *SaveXmlFile* and *CamOperations* are disabled as long as a part file or an XML file is not selected. After a part file is selected, the *JTree* panel on the left side of the split-pane shows the root node *part* with a list of features that correspond to the features in part file. When a feature node in the *JTree* is selected, a feature applet containing feature information is displayed on the right side of the split-pane. Operation of the application interface is explained in the following sections.

### 5.3 Retrieving CAM System Data

This section describes the implementation of *CAMSystem* and its supporting classes for querying the operation information from Unigraphics environment.
5.3.1 Retrieving Tags from Native Application

A part containing a list of operations is opened by clicking on the button OpenPartFile. This results in an instance of CAMPartModel being created on the Java application side. This instance of CAMPartModel calls the populate Operations method with an instance of CAMSystem class as an argument to retrieve all the operations from the selected part file. The CAMSystem class calls the native methods from the native dll application. The getOperationTags method takes a part tag as an argument and returns all the operations associated with the selected part. The getOperationTags method from the process-plan Java application calls the OperationTags method of the native dll application.

The getOperationTags method is defined by the keyword native, which tells the compiler that the getOperationTags method is a native method definition in the CAMSystem class on the Java side. The return type for this method is int []. An array of integers corresponding to each operation inside Unigraphics is returned. Each integer in the array uniquely identifies a Unigraphics operation. For example, the getOperationTags method for a part with the operations REAMING, DRILLING, and FACE_MILLING returns the result, as shown in Figure 5-8.

| Total number of Operations from UG are: 3 |
| Tag in Java is: 258 |
| Tag in Java is: 257 |
| Tag in Java is: 301 |

Figure 5-8 Results of calling getOperationTags method
The tag 256 refers to the REAMING operation inside the Unigraphics. Each operation inside Unigraphics has a unique tag number. Similarly, the tags 257 and 301 are used to identify the operation DRILLING and FACE_MILLING. The total number of operations refers to the total number of operations inside the part file. Next, the CamOperations button is clicked to display all the operations from the selected part file. A snippet of code for the native function getOperationTags is shown in Figure 5-9.

```c
JNIEXPORT jintArray JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_getOperationTags
   (JNIEnv *env, jobject obj, jint part_tag) {
   int err,type,subtype;
   tag_t obj_tag = NULL_TAG;
   jintArray jtagArray;
   int size=0;
   int allArray[ALL_SIZE];
   int *tagArray;
   while ((obj_tag=UF_OBJ_cycle_all(part_tag,obj_tag)) !=NULL_TAG) {
      err=UF_OBJ_ask_type_and subtype(obj_tag,&type,&subtype);
      if(err) {
         printf("Error : %d, Could'nt get object type. ",err);
      } else {
         if(type==100) {
            allArray[size]=obj_tag;
            size++;
         }
      }
   }
   tagArray = truncateArray (allArray, ALL_SIZE, size);
   if (size != 0) {
      jtagArray = (*env)->NewIntArray (env, size);
      (*env)->SetIntArrayRegion (env,jtagArray,0,size,tagArray);
      return jtagArray;
   }
```

Figure 5-9 Implementation of getOperationTags native method
The implementation of the native function `getOperationTags`, as it appears in
the header file, accepts two parameters even though its method definition on the
Java side accepts no parameters. JNI requires every native method to have these
two parameters. The first parameter for every native method is a `JNIEnv` interface
pointer. It is through this pointer that the native code accesses parameters and
objects passed to it from the Java application.

The second parameter is `jobject`, which references the current object itself. It is
equivalent to "this" variable in Java. For the native instance method
`getOperationTags` method, the `jobject` argument is a reference to the current
instance of the object. The third argument `part_tag` refers to the `tag` associated
with the part that was opened. The `env` pointer variable accesses the class of the
Java object `obj` and the `UF_OBJ_cycle_all` routine returns the `tag` of the object
`obj_tag` for a given `part_tag`. `UF_OBJ_ask_type_and_subtype` routine is then
called on the `obj_tag` object, which returns the object `type` and `subtype` of a
Unigraphics object. All the machining operations have a predefined `tag` value of
100. The machining operations are stored in an array `allArray`.

A `truncateArray` routine is called to trim the size of `allArray` to reduce its size to
the number of machining operations in Unigraphics. The `env` variable accesses the
`NewIntArray` and sets the size of the Java `tag` array `jtagArray`. The
`SetIntArrayRegion` routine is used to populate the `jtagArray` with all the machining
operation `tags`. The `jtagArray` of operations is finally returned by the native method
getOperationTags. A CAMOperation is created on the Java side for each operation returned by the jtagArray.

5.3.2 Extracting the Details of a UG Operation

The types and names of operations for the part tag are retrieved by calling the getOperationDetails method of CAMSystem class. This getOperationDetails method takes an instance of CAMOperation class as an argument and calls the native method getOperationDetails which takes two arguments: an instance of CAMOperation class and the operation tag operTag corresponding to an operation for the selected part file, as shown in Figure 5-10. The native method getOperationDetails then calls the native dll implementation of the getOperationDetails method to retrieve information specific to each operation like the name and type of operation.

```java
private native void getOperationDetails(CAMOperation camoper,int operTag);

public void getOperationDetails(CAMOperation camoper)
{
    getOperationDetails(camoper,camoper.getObjectID());
}
```

Figure 5-10 Method getOperationDetails defined in CAMSystem class

The implementation of the native dll method getOperationDetails is shown in Figure 5-11. The class of the java-objects variable camoper is accessed through the env variable by calling the get ObjectClass routine with the camoper object as an argument. Method UF_OPER_ask_oper_type when called on the operTag gives the type of operation for the given tag. Next, the method GetFieldID retrieves
JNIEXPORT void JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_getOperationDetails
  (JNIEnv *env, jobject obj, jobject camoper, jint operTag) {
  jfieldID fid;
  jstring jstr;
  const char *str;
  int op_type;
  char opName[UF_OPER_MAX_NAME_LEN + 1];
  jclass cls = (*env)->GetObjectClass(env, camoper);

  // Operation Type
  UF_OPER_ask_oper_type(operTag,&op_type);
  fid = (*env)->GetFieldID(env, cls, "operationType", "I");
  if (fid == 0) {
    return;
  }
  (*env)->SetIntField(env, camoper, fid, op_type);

  // Operation Name
  UF_OPER_ask_name_from_tag(operTag,opName);
  fid = (*env)->GetFieldID(env, cls, "objectName", "Ljava/lang/String;" );
  if(fid == 0) {
    return;
  }
  jstr = (*env)->NewStringUTF(env, opName);
  str = (*env)->GetStringUTFChars(env, jstr,0);
  (*env)->ReleaseStringUTFChars(env, jstr, opName);
  (*env)->SetObjectField(env, camoper, fid, jstr);
}

Figure 5-11 Implementation of native method getOperationDetails

the identifier *fid* for the instance integer member variable *operationType*. The
*operationType* Java variable is defined in the *CAMOperation* class. *SetIntField*
method sets the field *fid* to a new value *op_type*. For getting the name of the tag
method, *UF_OPER_ask_name_from_tag* is called on the *operTag* and a character
array *opName* is returned The *GetFieldID* method is then called on the variable
objectName and returns the identifier of type string for the given class. A new Java string jstr is returned when the method NewStringUTF is called on opName.

Next, the method GetStringUTFChars returns a pointer to the UTF encoding of the string jstr. Finally, the method ReleaseStringUTFChars is called to free up any space occupied by the native code. A new value is set for field id fid by calling the method SetObjectField with a value jstr.

### 5.3.3 Getting Groups for UG Operation

Once the operation name and its type are retrieved, the different groups associated with an operation can be obtained by calling the getOperationGroups native method. The groups that get returned are the Program, Method, Geometry and Cutter Group. The information for each of these groups is assigned to the attributes geomGrp, progGrp, methdGrp, cuttrGrp defined by the CAMOperation class. The getOperationGroups inside the CAMSystem class has two different declarations. The method getOperationGroups (CAMOperation camoper) Figure 5-12 takes an instance of CAMOperation class.

```java
private native void getOperationGroups(CAMOperation camoper, int operTag);

public void getOperationGroups (CAMOperation camoper)
{
    getOperationGroups (camoper, camoper.getObjectID());
}
```

Figure 5-12 Method getOperationGroups defined in CAMSystem class
The method `getObjectID` returns an operation from Unigraphics. This method, in turn, calls the native method `getOperationGroups(CAMOperation camoper, int operTag)` defined in the `CAMSystem` class. The arguments refer to an instance of the `CAMOperation` class and the object id for each operation, which is a numerical value that identifies an operation uniquely. The `getOperationGroups` Java method calls the native method `getOperationGroups` Figure 5-13. This native method then `dll` applications function `getOperationGroups` and references each operation tag with its respective groups.

```c
JNICALL void JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_getOperationGroups(JNIEnv *env, jobject obj, jobject camoper, jint operTag) {
    jclass cls = (*env)->GetObjectClass(env, camoper);
    jint fid;
    fid = (*env)->GetFieldID(env, cls, "geomGrp", "I");
    if (fid == 0) { return;}
    (*env)->SetIntField(env, camoper, fid, operTag);
    fid = (*env)->GetFieldID(env, cls, "progGrp", "I");
    if (fid == 0) { return;}
    (*env)->SetIntField(env, camoper, fid, operTag);
    fid = (*env)->GetFieldID(env, cls, "methdGrp", "I");
    if (fid == 0) { return;}
    (*env)->SetIntField(env, camoper, fid, operTag);
    fid = (*env)->GetFieldID(env, cls, "cuttrGrp", "I");
    if (fid == 0) { return;}
    (*env)->SetIntField(env, camoper, fid, operTag);
}
```

Figure 5-13 Implementation of the native method `getOperationGroups`
This causes the information pertaining to a group for each operation to be available inside the \textit{CAMSystem} class on the Java side.

The native \textit{dll} application’s \textit{getOperationGroups} method has arguments \textit{camoper} and \textit{operTag}. Method \textit{getObjectClass} is called on the Java object \textit{camoper} and the class returned is stored in a Java class variable \textit{cls}. The geometry group for an operation can be retrieved by calling the Unigraphics routine \textit{UF\_OPER\_ask\_geom\_group} on the \textit{operTag} parameter. Next, the \textit{GetFieldID} method is called to retrieve the identifier \textit{fid} for the instance integer member variable \textit{geomGrp}. Finally, the \textit{env} pointer variable is used to set the \textit{fid} with a value of \textit{group\_tag} by calling the method \textit{SetIntField}. Similarly, the information for different groups like the Program, Method, and Cutter groups is retrieved by calling the appropriate UG routines in each case.

\textbf{5.3.4 Starting the CAM Session}

Before creating the operations in the Unigraphics environment, all the necessary templates and libraries corresponding to the manufacturing environment have to be loaded. This is done by calling the \textit{startCAMSession} button. When the \textit{StartCamSession} button in the process-data-exchange applet is clicked, it calls the \textit{startCAMSession} method for initializing the Unigraphics environment. This method in turn, calls the \textit{CAMInitialize} method of the \textit{CAMSystem} class. The \textit{CAMInitialize} method is a native method inside the \textit{CAMSystem} class. The \textit{CAMInitialize} method calls the
The `Java_edu_ohiou_implanner_cam_CAMSystem_CAMInitialize` method from the native `dll` application. The part file is set to the manufacturing mode when this method returns a value of zero and fails to initialize for a return value of -1. The implementation of the `startCAMSession` method on the Java side is shown in Figure 5-14.

```java
private native int CAMInitialize();

public int startCAMSession()
{
    int result = -1;
    result = CAMInitialize();
    if (result == 0)
    {
        System.out.println(">> CAM session successfully initialized.");
    }
    return result;
}
```

Figure 5-14 Methods for initializing the CAM session

This native method declaration in the `CAMSystem` class provides only the method signature for `CAMInitialize`. It provides no implementation for the method. The implementation for `CAMInitialize` is provided in a separate native `dll` application. This method returns the output, as shown in Figure 5-15, when executed successfully.

```
result : 0
CAM Session Successfully initialized
```

Figure 5-15 Result of CAM session initialization
Native *dll* implementation of the method *CAMInitialize* involves calling the *UF_CAM_init_session* method from the UG/Open API. This method *UF_CAM_init* initializes the current CAM session based upon the contents of the configuration file. If a CAM session currently exists it is first unloaded. For a native *CAMInitialize* method the *jobject* argument refers to the current instance of the object. The native *CAMInitialize* method takes the arguments *JNIEnv* and *jobject*, as shown in Figure 5-16. The native method returns an integer value, which is type cast by using *jint* to the Java *type* variable so that it can be returned to the Java application directly.

```c

JNIEXPORT jint JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_CAMInitialize
    (JNIEnv *env, jobject jobj) {
    return((jint)UF_CAM_init_session());
}

```

Figure 5-16 Implementation of the native *CAMInitialize* method

### 5.4 Creating Operations in CAM System

After the session is initialized, operations can be created on a part by selecting a process corresponding to a feature node. The mapping of processes into operations and their subsequent creation inside Unigraphics environment is described in this section.

#### 5.4.1 Process Mapping

Mapping of machining processes into Unigraphics operations requires that the processes are able to create a Unigraphics operation. Methods *getCAMType* Figure 5-17 and *getCAMSubType* are defined for each process that extends the...
MfgProcess class. The getCAMType method for a TwistDrilling process is shown in Figure 5-17.

```java
public String getCAMType() {
    String classType = properties.getProperty(this.getClass().getName() + ".camtype");
    if (classType != null)
        return classType;
    else
        return properties.getProperty("edu.ohiou.implanner.processes.holemaking.camtype");
}
```

Figure 5-17 Implementation of getCAMType method

Method getProperty is called on the properties object in which it searches for the property with key specified by class name edu.ohiou.implanner.processes.holemaking.camtype in the properties file. If it is not found in this properties file, the default properties list and its defaults, recursively, are then checked. This method returns null if the property is not found. The properties file, as shown in Figure 5-18 contains a list of all the keys for different process objects so that they

```
edu.ohiou.implanner.processes.holemaking.camtype=drill
edu.ohiou.implanner.processes.milling.camtype=mill_planar

edu.ohiou.implanner.processes.holemaking.camsubtype=DRILLING
edu.ohiou.implanner.processes.milling.camsubtype=FINISH_WALLS
edu.ohiou.implanner.processes.couterboring.camsubtype=COUNTERBORING
edu.ohiou.implanner.processes.reaming.camsubtype=REAMING
edu.ohiou.implanner.processes.spotdrilling.camsubtype=SPOT_DRILLING
edu.ohiou.implanner.processes.face milling.camsubtype=FACE_MILLING
edu.ohiou.implanner.processes.boring.camsubtype=BORING
edu.ohiou.implanner.processes.twistdrilling.camsubtype=DRILLING
edu.ohiou.implanner.processes.endmillingslotting.camsubtype=PLANAR_MILLING
edu.ohiou.implanner.processes.endmillingperipheral.camsubtype=PLANAR_PROFILE
```

Figure 5-18 Properties file for mapping processes to Unigraphics operations
can be mapped to operations.

The matching key in the properties file for the TwistDrilling process returns the camtype for this process as drill. Similarly, the getCAMSubType method is called with the key edu.ohiou.implanner.processes.holemaking.camsubtype. A value for camsubtype is then returned based on the process to be mapped. For the TwistDrilling process, a camsubtype with a value of DRILLING is returned. This refers to the name of Unigraphics equivalent operation for the TwistDrilling process. This is repeated for all other processes, and the corresponding operations to be mapped are specified in the properties file and are accessed every time a process is selected by the user at the process data exchange applet to create an operation in Unigraphics.

The processes that extend the MfgProcess Figure 5-19 define the camType and camsubtype. The camsubtype corresponds to processes on the Java side. Each of these processes is defined in a separate class on the process-plan application side. The different classes that extend the MfgProcess are Boring, Reaming, Milling, SpotDrilling, CounterBoring, FaceMilling, TwistDrilling, etc. shows the classes that extend the MfgProcess.
5.4.2 Creating Operations in Unigraphics

A part for which the operations are to be created is loaded first at the process data exchange applet. The createInstance method of NetExample Class creates a list of features with processes, which are shown in the JTree panel of the data exchange applet. The part model may accept features for feature-based design and can generate processes using 3I-PP, but due to the limitation in data exchange, a sample of features was directly implemented in Java. Thus, the features in the process data exchange applet simulate the features of the part. When a feature is clicked, a feature panel is shown in the right half of the split-pane. The feature panel displays all the feature-related parameters. Once the
feature node is clicked and expanded, a list of processes that can be created in Unigraphics for a feature are shown. Since the CAM session has to be initialized at this point, the \textit{StartCamSession} button should be clicked. Method \textit{startCAMSession} is called which initializes the CAM session. At this point, all the setup and configuration files are loaded for creating a UG operation.

A \textit{right click} mouse event is implemented on each process node in the \textit{JTree} panel. A popup trigger is displayed with a \textit{createOperation} item for the right click mouse event on the process node. On clicking the \textit{createOperation} item in the popup trigger, the \textit{NetExample} class calls the \textit{createUGProcess_action Performed} method. This method calls the \textit{createOperation} method on an instance of \textit{MfgProcess} class, as shown in Figure 5-20.

```java
private native int createOperation(String camType, String camSubType);

public int createOperation(MfgProcess process)
{
    int operationID;
    operationID = createOperation(process.getCAMType(), process.getCAMSubType());
    return operationID;
}
```

Figure 5-20 Method \textit{createOperation} defined in \textit{CAMPartModel}

Method \textit{addOperation} is also called with processed as an argument on an instance of \textit{CAMPartModel} class. The \textit{createOperation} method of the \textit{CAMSystem} class calls the native declaration of the \textit{createOperation} method.
The `createOperation` method accepts an instance of `MfgProcess` as parameters and calls the methods `getCamType` and `getCamSubType` on it. These methods return the names of setup and the process name for the process selected by the user from the `JTree` panel of the process data exchange applet. Once the `camType` and `camSubType` are retrieved, the native declaration of the `createOperation` method is called with the `camType` and `camSubType` as arguments. The `createOperation` method with the native declaration, in turn, calls the actual implementation of this method inside the native `dll` application.

The native method implementation of the `createOperation` method Figure 5-21

```c
JNIEXPORT jint JNICALL Java_edu_ohiou_implanner_cam_CAMSystem_createOperation
  (JNIEnv *env, jobject obj, jstring type, jstring subtype) {
  int err;
  tag_t new_oper;
  char name[31];
  char *strTyp;
  char *strSubTyp;

  strTyp= (*env)->GetStringUTFChars(env, type, 0);
  strSubTyp= (*env)->GetStringUTFChars(env, subtype,0);

  err = UF_OPER_create (strTyp, strSubTyp, &new_oper);
  err = UF_OPER_ask_name_from_tag(new_oper,name);

  UF_PART_save();
  (*env)->ReleaseStringUTFChars(env, type, strTyp);
  (*env)->ReleaseStringUTFChars(env, subtype, strSubTyp);
  return ((jint)new_oper);
}
```

Figure 5-21 Implementation of native method `createOperation`

takes two arguments `camType` and `camSubType` in addition to the `JNIEnv` variable and the `jObject` variable. The `camType` refers to the setup of the Unigraphics CAM
session. For example, the setup for a part which requires a milling or drilling operation, is set to mill_planar or drill. The \textit{camSubType} that is also passed as an argument to the \textit{createOperation} method depends on the \textit{camType} that is chosen. A mill_planar \textit{camType} has the \textit{camSubType} as FINISH_WALLS, FACE_MILLING, etc. The native \textit{createOperation} method on the Java side then calls the native source implementation for the \textit{createOperation} method, which returns an integer value once the operation is created successfully inside Unigraphics. A return value of 0 for this method call indicates the operation has been successfully created in Unigraphics, whereas -1 indicates failure to create an operation.

For the native \textit{dll} method \textit{createOperation}, the parameters that are passed from the Java side are the \textit{type} and \textit{subtype}. Each of these parameters is represented as \textit{jstring} in the native “C” language. A variable \textit{new_oper} is declared as a type tag object as Unigraphics treats operations as tags. This tag corresponds to the new operation that is created every time the user decides to create an operation. Next, for accessing the string that is passed from the Java side the \textit{GetStringUTFChars} method is called on the Java \textit{type} variable. The \textit{GetStringUTFChars} converts the built-in unicode representation of a Java String into UTF-8 string. This results in a pointer to an array of Unicode characters that comprise the string \textit{type} is assigned to a pointer \textit{strTyp}. Similarly, the method \textit{GetStringUTFChars} is called on the \textit{subtype} java variable and returns a pointer \textit{strSubTyp} on an array of characters.
Native methods access and manipulate Java objects such as strings through the *env* interface pointer. In “C” this requires using the *env* pointer to reference the JNI method. Next, the UG routine *UF_OPER_create* method is called with the variables *strTyp* and *strSubTyp* as arguments. This method call results in the operations being created inside the Unigraphics environment. The operation that is created depends on the arguments *strTyp* and *strSubTyp* that correspond to the operation *type* and *subtype*. The third argument *new_oper* for the *UF_OPER_create* method refers to the new operation that is created on the part file when the *UF_OPER_create* method executes successfully. This method returns a value of 0 when executed successfully, and -1 gets returned if the operation cannot be created for the part.

The values returned by each of the method calls above are assigned to a variable *err*. The value of the *err* variable is checked to see if all the method calls were executed successfully. For saving the operation into the Unigraphics part file, the part model with UG routine *UF_PART_save()* is called. This method saves the newly created operation to the part file. When the native code has finished using the UTF-8 string, it calls the *ReleaseStringUTFChars* on the pointer variables *strTyp* and *strSubTyp*. *ReleaseStringUTFChars* informs the virtual machine (VM) that the native method is finished with the string so that the memory taken up by the UTF-8 can be freed. Finally, the *new_oper* tag is returned by type casting it to a Java integer type variable. This results in a *CAMOperation* being
created on the Java application side. The CAMOperation is then added to the CAMPartModel. Also, the name, type, subtype and groups are retrieved.

5.5 Saving Process Model in XML File

This section describes the procedure for storing and retrieving process information by using XML document.

5.5.1 Writing an XML File

Upon clicking the SaveXmlFile button, the writeXMLFile method is called on an instance of NetExample netPart with a file object featureXMLFile as an argument. As a result, MfgPartModel’s writeXMLFile is called as the NetExample class extends the MfgPartmodel class. Method writeXMLFile, in turn, calls the method writeXML Figure 5-22, Figure 5-23 for the MfgProcess and MfgFeature

```java
public void writeXML(StringBuffer buffer, String indent)
{
    String firstIndent=indent + ImpConstants.INDENT_INCREMENT;
    String SecondIndent=firstIndent + ImpConstants.INDENT_INCREMENT;
    System.out.println("process" + buffer);
    String className = this.getClass().getName();
    buffer.append(indent).append("<"+ className + " procName="
    .append(this.getProcessName())
    .append("" processingTime =""+this.getProcessTime()+"">\n")

    if (this.getMachine()!=null){
        buffer.append(firstIndent).append("<Machine Name="
        +this.getMachine().getMachineName()+"/>\n")
    }
    if (this.procActivity !=null){
        buffer.append(firstIndent).append("<MfgActivity
        Name="
        +this.procActivity.toString()+"/>\n")
        buffer.append("</MfgActivity>\n");
    }
}
```

Figure 5-22 Method writeXML implemented in the MfgProcess class part 1
```java
if (toolConstraint != null){
    buffer.append(firstIndent).append("<ToolConstraint ConstraintClass=" +
    this.getToolConstraint().getConstraintClass()+ "]"">\n");
    toolConstraint.writeXML(buffer,SecondIndent);
    buffer.append(firstIndent).append("</ToolConstraint>\n");
}
if (processTool != null)
    processTool.writeXML(buffer,firstIndent);
if (cutParameter != null)
    cutParameter.writeXML(buffer,firstIndent);
buffer.append(indent).append("</" +className+ "]"">\n");
```

Figure 5-23 Method `writeXML` implemented in the `MfgProcess` class part 2

class. The `writeXML` method creates an XML file containing the details of processes.

### 5.5.2 Writing the Element Tags

The Class name of the process is set to `className` string variable. Each element contains starting and closing tags, as shown in Figure 4-7. Attributes are created for these elements from the different class parameters that relate to a particular process. For example, the method `getProcessName` and `getProcessing Time` are called on a process element and the values returned are assigned to the attributes `procName` and `processingTime` of the process element. Similarly, different element tags are created for `ToolConstraint`, `Process Constraint`, `Constraint` and `CuttingParameter` classes by calling the `writeXML` method for each class separately, as shown in Figure 5-24. This results in a well-formed XML document being created.
if (toolConstraint != null) {
  buffer.append(firstIndent).append("<ToolConstraint ConstraintClass=" +
    this.getToolConstraint().getConstraintClass() + "">
  toolConstraint.writeXML(buffer, SecondIndent);
  buffer.append(firstIndent).append("</ToolConstraint>");
}

Figure 5-24 Method writeXML for ToolConstraint class

The ProcessList element consists of all processes associated with a particular feature. The Element edu.ohiou.implanner.processes.FaceMilling shows the entire package structure to which a FaceMilling class belongs. This is useful when parsing back the XML file. The XML file saved for the process generated in 3I-PP system and passed using data exchange Figure 5-25 and Figure 5-26.

<?xml version='1.0'?>
<?xml-stylesheet type="text/xsl" href="c:\Interface\xmlcommFolder\features.xsl"?>
<LISTOFFEATURES>
  <PROCESS NAME = "P68-F2_CENTER_SLOT-END-M-VERTI-END-M"
    INSTANCEOF = "END-MILLING-SLOTTING">
    <param name="TOOLAXIS">NIL</param>
    <param name="POSITIVE-TOL">NIL</param>
    <param name="NEGATIVE-TOL">NIL</param>
    <param name="FLATNESS">NIL</param>
    <param name="SURFACE-FINISH">NIL</param>
    <param name="SWEEDIRECTION">NIL</param>
    <param name="SWEPTSOLID">MAKE-ENDMILLING-SOLID</param>
    <param name="DISP-PROCESS">DISP-CUTTING</param>
    <param name="DISP-PROCFEATURES">DISP-CUTTING-FEATURES</param>
    <param name="PLANNING-STATUS">SELECT</param>
    <param name="PREFERRED-TO">NIL</param>
    <param name="IS-MACHINED-BY+INV">F2_CENTER_SLOT</param>
    <param name="MAY-BE-MACHINED-BY+INV">NIL</param>
    <param name="PREFERRED-TO+INV">NIL</param>
  </PROCESS>
<PROCESS NAME = "P68-F2_CENTER_SLOT-END-M-VERTI-END-M"
  INSTANCEOF = "END-MILLING-SLOTTING">
  <param name="TOOLAXIS">NIL</param>
  <param name="POSITIVE-TOL">NIL</param>
  <param name="NEGATIVE-TOL">NIL</param>
  <param name="FLATNESS">NIL</param>
  <param name="SURFACE-FINISH">NIL</param>
  <param name="SWEEDIRECTION">NIL</param>
  <param name="SWEPTSOLID">MAKE-ENDMILLING-SOLID</param>
  <param name="DISP-PROCESS">DISP-CUTTING</param>
  <param name="DISP-PROCFEATURES">DISP-CUTTING-FEATURES</param>
  <param name="PLANNING-STATUS">SELECT</param>
  <param name="PREFERRED-TO">NIL</param>
  <param name="IS-MACHINED-BY+INV">F2_CENTER_SLOT</param>
  <param name="MAY-BE-MACHINED-BY+INV">NIL</param>
  <param name="PREFERRED-TO+INV">NIL</param>
</PROCESS>
</LISTOFFEATURES>

Figure 5-25 Process data generated by 3I-PP system part 1
<param name="MAY-USE-TOOL">END-MILLING-TOOL</param>
<MACHINE-CONSTRAINT NAME = "M58-F2_CE-VERTI" INSTANCEOF = "VERTICAL-MILL">
<param name="TOOLHEAD">NIL</param>
<param name="BED-SIZE-X">NIL</param>
<param name="BED-SIZE-Y">NIL</param>
<param name="BED-SIZE-Z">NIL</param>
<param name="PROCESS">NIL</param>
<param name="SETUP-TIME">60</param>
<constraint name="POWER" operator="\geq" value="3.0"/>
<param name="MAT-HANDLING-TIME">0.4</param>
<param name="SPEED-EFFICIENCY">NIL</param>
<param name="TOOL-CHANGE-TIME">0.2</param>
<param name="UNIT-COST">1.0</param>
<param name="MAY-USE-MACH+INV">NIL</param>
<param name="SHOULD-USE-MACH+INV">P68-F2_CENTER SLOT-END-M-VERTI-END-M</param>
</MACHINE-CONSTRAINT>
<TOOL-CONSTRAINT NAME = "T58-F2_CE-END-M" INSTANCEOF = "END-MILLING-TOOL">
<constraint name="FLUTE-LENGTH" operator="\geq" value="30.16"/>
<param name="TOTAL-LENGTH">NIL</param>
<param name="SHANK-DIAMETER">NIL</param>
<constraint name="NUMBER-OF-TEETH" operator="\=" value="4"/>
<constraint name="MATERIAL" operator="\"EQL\" value="HSS"/>
<constraint name="DIAMETER" operator="\=" value="25.5"/>
<param name="LIFE-CYCLE">220</param>
<param name="COST">30</param>
<param name="MAY-USE-TOOL+INV">NIL</param>
<param name="SHOULD-USE-TOOL+INV">P68-F2_CENTER SLOT-END-M-VERTI-END-M</param>
</TOOL-CONSTRAINT>
<CUT-PARAMETERS NAME = "CUT-DATA47" INSTANCEOF = "CP-END-MILLING-SLOTTING">
<param name="FOR-PROCESS">P70-F2_CENTER SLOT-END-M-CNC-END-M</param>
<param name="TOOL-DIAMETER">NIL</param>
<param name="SPEED">450</param>
<param name="FEED">0.004</param>
<param name="PART-MATERIAL">ALUMINUM</param>
<param name="TOOL-MATERIAL">HSS</param>
</CUT-PARAMETERS>
</PROCESS>
</LISTOFFEATURES>

Figure 5-26 Process data generated by 3I-PP system part 2
5.5.3 Reading an XML File

The SAX parser reads the XML document and fires events as it encounters start Element tags and end Element tags. The events are fired as the SAX parser reads the XML document from top to bottom, a tag at a time. SAX parser uses a reader object from ImpXmlReader to parse an XML file into its equivalent Java objects. This involves calling appropriate content handlers to delegate parsing based on the starting or ending element that is read. Each of the classes implements an inner class that extends SaxHandler, which parses the starting and ending elements, as the elements that are being read from the XML file have attributes and no content. A SaxHandler for startElement of the Process constraint’s inner class ConstraintSaxHandler is shown in Figure 5-27.

```java
public void startElement
(String namespaceURI, String localName, String qName, Attributes atts) {
    String name = verifyName(localName, qName);
    reader.logParsing("starting element <" + name + ">");

    if(name.equalsIgnoreCase("ConstraintClass"))
    {
        model = new ProcessConstraint(atts.getValue("Name"), new LinkedList());
        reader.addObject(model);
        return;
    }

    if(name.equalsIgnoreCase("Constraint"))
    {
        Constraint constraint = new Constraint(atts.getValue("Name"),
                                                atts.getValue("Operator"),
                                                atts.getValue("Value"));
        model.addConstraint (constraint);
        return;
    }
}
```

Figure 5-27 Method startElement for constraintSaxHandler Class
First the element name is checked to see if it is of type \textit{ConstraintClass}. An instance of Process Constraint class is created for each process element that is read by the SAX parser, and the attributes are assigned to the processes by calling the \textit{getValue} method on the attributes. The SAX parser is event-based; it checks for instances of a class and parses the element into \textit{JTree} object. Each element that is read from the XML file appears as a node in the \textit{JTree}. Next, a valid \textit{MfgPartModel} is created and the \textit{TreeModel} is populated from this hierarchy. If the starting element is \textit{Constraint} class, the \textit{getValue} method is called on each attribute of the element and their values are retrieved. Finally, the constraint object is added to the MfgPartModel. The ending element for the \textit{ConstraintSaxHandler Class} is shown in Figure 5-28.

```java
public void endElement(String namespaceURI, String localName, String qName) {
    String name = verifyName(localName, qName);
    reader.logParsing("ending element </" + name + ">, object " + model);
    reader.logParsing("Handler is " + this);
    reader.logParsing("objects" + reader.getObjects());

    if (name.equalsIgnoreCase("ToolConstraint")) {
        ContentHandler handler = reader.getNextHandler();
        reader.logParsing("same element, handler " + handler);
        if (handler != null) {
            try { 
                reader.getXMLReader().setContentHandler(handler );
                return;
            }
            catch (Exception ex) {ex.printStackTrace();}
            }
            }
}
```

Figure 5-28 Method endElement for \textit{constraintSaxHandler} class
The endElement retrieves the content handler for the current class and delegates it to the next higher element in the hierarchy of the TreeModel whenever the parser encounters a closing element tag. Similarly startElement and endElement are implemented for the \textit{MfgProcess}, \textit{Cutting Parameter}, \textit{Tool}, and \textit{Tool Constraint} classes.
6 TESTING

This chapter describes the test examples on which the process data exchange mechanism was implemented.

6.1 Testing Examples

This section explains the working of process data exchange application, which is responsible for the data exchange between the Java process-plan model and the Unigraphics environment. Each step involved in the data exchange process is explained below.

6.1.1 NetExample

The *NetExample* application will provide an application interface once the application is run. A screen shot of *NetExample* interface is shown in Figure 6-1.

![Figure 6-1 NetExample’s Process Data exchange Interface](image)
A default root node corresponding to a part is shown in the interface when the \textit{NetExample} application is run for the first time. The \textit{NetExample} Java application was run for different part files. The explanation on a few of these test cases is covered in the subsequent portion of this section.

6.1.2 Loading Part File

The \textit{NetExample}'s process-data-exchange interface provides the command for loading a Unigraphics part file. The part file is then selected by the user.

6.1.3 Querying CAM Operation Data

The Anc101_Manuf.prt file, as shown in Figure 6-2, is first loaded through the process data exchange interface.

![Anc101_Manuf.prt file](image)
At this point the *NetExample* interface is ready to interact with the Unigraphics environment. The Anc101_Manuf.prt file already has predefined set of operations for its part model. Figure 6-2 shows the predefined operations on the Anc101_Manuf.prt file. Now the information for all the operations for the Anc101_Manuf.prt part file has to be sent to the Java application side so that all the Unigraphics operations can be made available at the *NetExample*’s interface.

The *Cam Operation* button in the *NetExample*’s interface, when clicked, will cause all the operations for the selected part file to be displayed inside an applet. The is made possible by creating an instance of *CAMPartModel* when the *Cam Operation* button is clicked. This *CAMPartModel* instance calls the *populateOperations* method. The *populateOperations* method will in turn call the *getOperationsTags* and *getOperationDetails* methods to obtain all the information about the Unigraphics Operations. The getOperationTags will retrieve all the tags associated with each operation and the *getOperationDetails* retrieves information about the *type* and *subtype* for identifying an operation.

A screen shot of the operations from Unigraphics that are available at the *NetExample*’s or Java application interface is shown in Figure 6-3. The operation objects that are retrieved from the Unigraphics are shown as a node object in the *JTree* on the Java application side. For example, the operations ROUGH_FOLLOW, DRILLING, etc. are the objects in the *JTree* corresponding to the ROUGH_FOLLOW and DRILLING operations in Unigraphics. The *JTree* now has all the operations from the Unigraphics environment onto the Java side.
Figure 6-3 List of operations from the Anc101_Manuf.prt file

Another example, switcharm_manuf.prt, is a complex part with many features like slot, hole, and pocket, as shown in Figure 6-4. The NetExample interface will load the switcharm_manuf.prt file and calls the populateOperations method and makes the operation information available to the Java application side, as shown in Figure 6-5. The operations in the JTree in Figure 6-5 reflect the operations that exist inside the switcharm_manuf.prt file.
Figure 6-4 Switcharm.prt file

Figure 6-5 List of Operations from the switcharm_manuf.prt file
6.2 Creating Operations in Unigraphics

The NetExample application interface has the capability of creating operations inside Unigraphics. This is demonstrated on the Block.prt file shown in Figure 6-6. The Block.prt file is a part model, which has the features slot, pocket, and holes. At first the Block.prt file is selected by the user from the NetExample interface. The Block.prt file has no operations associated with it, as the part is in the Unigraphics modeling environment. The Block.prt file has to be initialized to the manufacturing environment to allow operations to be created on it. Initialization of the Block.prt file is done by setting the Unigraphics file to the manufacturing mode. This causes all

![Figure 6-6 Block.prt file with list of features](image-url)
the manufacturing templates and necessary setups and Configuration files to be loaded inside the Unigraphics environment.

The Block.prt file has no operations at this stage. The NetExample then loads the Block.prt file. A set of predefined features and processes are created inside the NetExample Class. The processes created inside the NetExample Class are REAMING, FACE_MILLING, SPOT_DRILLING, etc. The NetExample interface shows a default root node as part before the part file is loaded, as shown in Figure 6-7.

![NetExample interface before Block.prt file is loaded](image)

Figure 6-7 NetExample interface before Block.prt file is loaded
When the Block.prt file is loaded, the part panel of the *NetExample* interface displays the list of all the features for the Block.prt file, as shown in Figure 6-8.

![Figure 6-8 NetExample interface after Block.prt file is loaded](image)

The part panel displays a *JTree* of feature objects in Java. Each of the feature objects Figure 6-9 is associated with one or more processes.

![Figure 6-9 Feature panel for the slot feature](image)
The process instances are created in the *NetExample* class for these features. When the feature node in the part panel *JTree* is clicked, the Feature panel is displayed.

The feature parameters are displayed in the feature panel on the right side of the *NetExample*’s interface split-pane. When the feature node in the part panel *JTree* is clicked, it expands and shows the processes which are associated with a feature. Figure 6-10 shows the expanded processes and feature tree nodes.

![Figure 6-10 Features and Process are shown in a JTree](image)

The features Slot, Hole, and Pocket shown in Figure 6-10 have processes END MILL SLOTTING, ROUGH MILLING, SPOT DRILLING, etc. The next step is...
to start the CAM Session by clicking the `StartCAMSession` button. This results in the native method `startCAMSession` to be called, which initializes the CAM session on the part that was earlier set to manufacturing mode. The next step involves creating operations inside the Block.prt file. By right clicking at the process node in the `JTree` a pop-up menu, `CreateOperation`, appears as shown in Figure 6-11.

![JTree with Create Operation option](image)

Figure 6-11 Create operation pop-up menu for Block.prt file

When the “Create Operation” option is clicked the `createUGProcess_action Performed` event gets called. The `createUGProcess` calls the method `createOperation` of the `CAMSystem` class, which in turn calls the native method for `createOperation` from the native `dll` application. This results in an operation being created on the Block.prt file. All the subsequent operations, which are created through the pop-menu on right clicking a particular process, are added to a linked
list of operations inside the `addOperations` method that is called for the instance of `CAMPartModel` corresponding to the Block.prt file.

The “Create Operation” menu item calls the `createOperation` method on an instance of the `MfgProcess` class. The `createOperation` method in turn retrieves the `CAMType` and `CAMSubType` for the END MILL SLOTTING process Figure 6-12. The `type` and `subtype` that are used are based on the processes, which are mapped in the mfgproperties file.
For the END MILL SLOTTING process the `CAMType` that is returned is `mill_planar` and `CAMSubType` is `PLANAR_MILL`. `CAMType` and `CAMSubType` then are passed as arguments to the native `createOperation` method. As a result of this, the operation is created for the END MILL SLOTTING process inside the Block.prt file as shown in Figure 6-12.

The same operations can be retrieved as explained earlier by clicking on the `CamOperationButton`, which displays the operations for the Block.prt file in a `JTree` inside a Java applet. Figure 6-13 shows the list of all operations created in the Block.prt file and stored in `CAMPartModel` object.

![Figure 6-13 List of Operations retrieved from Block.prt file](image)
6.3 Saving Process Data

The process data that are generated for the NetExample Java application are saved in an XML format for later retrieval. After a part file is loaded for which the operations have to be created, the part panel or the operation tree shows a partial list of features and processes. The information for the features and their associated processes can be saved to an XML file, as shown in Figure 6-14. This is done by clicking on the **SaveXml File** button in the NetExample interface.

---

**Figure 6-14 Process and Feature Data saved to BlockXml File**
When the `SaveXMLFile` button is clicked, it calls the `writeXMLFile` method on an instance of `MfgPartModel` with a file object as argument, which in turn calls the `writeXML` method from the `MfgProcess` Class. This causes an XML file to be created for saving the process and the feature information. The XML file that is saved for the features and processes defined in the `NetExample` application.

### 6.4 Reading the XML

The `OpenXMLFile` button in the `NetExample` interface allows the XML file that is saved by using the `writeXML` file method to be retrieved back into a `JTree` of feature and operations. Figure 6-15 shows the BlockXml.xml file that is read by the Java application interface.

![Figure 6-15 List of Features read from BlockXML.xml file.](image-url)
When the *OpenXMLFile* button is clicked, a XML reader object is created for reading the XML file into Java Objects. Each element in the XML file is parsed into features and processes. Parsing of XML elements is done by calling appropriate SaxHandlers for each element that is read from the XML file. This results in all the features and process information to be retrieved back into a *JTree* of Java objects.
7 CONTRIBUTIONS, LIMITATIONS AND FUTURE WORK

This section highlights the contributions made by this thesis towards the process data exchange mechanism along with its limitations and the direction for future work.

7.1 Contributions

This thesis provides a mechanism to control the CAD / CAM application environment, such as Unigraphics, with the process-modeling interface represented as a Java application.

1. Native application integration is achieved using Java Native Interface which allows the Process Data exchange Application to interact with the UG/Open API.

2. The Processes from the Java application side get mapped as operations in the Unigraphics environment through Java's platform independent integration capability.

3. Manufacturing and process planning information can be sent to and retrieved from Unigraphics which would otherwise put significant overhead on the user to manually generate all the operations on a part file and supply the operation parameters.

4. Application interoperability is achieved across native platforms.

The process data exchange mechanism that has been adopted in this thesis can in general be applied in any CAD / CAM application environment that supports native application integration.
7.2 Limitations

The limitations of the process data exchange mechanism used in this thesis are:

1. The API that links features to operations is not disclosed by Unigraphics and as a result, operations which are created are incomplete inside the CAM System. The operations are not associated with the features or any part geometry necessary.

2. Process mapping is possible for only several processes using the process data exchange mechanism because all the processes of MfgProcess type cannot be mapped directly to Unigraphics operations. This specific process or feature parameters should be used to uniquely map a process to an operation, which was outside the scope of the thesis and the CAM system.

3. Synchronization between 3I-PP and the process-plan model is incomplete because of the limitation of the XML interface as a result of operations that cannot be generated directly in Unigraphics.

7.3 Future work

Once the operations are created in the part model, related information can also be sent to the Unigraphics environment. This will allow visualizing of all the manufacturing operations on the part within the Unigraphics environment. In addition, the process data exchange mechanism can be applied to remote applications across a network, which will allow processes to be controlled and
visualized remotely. The limitations of the process data exchange mechanism mentioned in the previous section may also be treated as ideas for future work.
REFERENCES


   http://java.sun.com/docs/books/tutorial/native1.1/


[23] XML Development Goals (November 2001),

www.xml.com/pub/a/98/10/guide1.html