AN APPLICATION OF EXTENSIBLE MARKUP LANGUAGE FOR
INTEGRATION OF KNOWLEDGE-BASED SYSTEM WITH JAVA
APPLICATIONS

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirement for the Degree
Master of Science

By
Sachin Jain

November, 2002
ACKNOWLEDGEMENTS

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. David Koonce and Dr. Constantinos Vassiliadis. 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, Sridharan Thiruppalli, Jaikumar Arumugam and Prashant Borse for their excellent cooperation and suggestions. A lot of information useful to the work was found via the World Wide Web; I thank those who made their material available on the Web and those who kindly responded back to my questions over the news-groups.

Finally, it has been pleasure to pursue graduate studies at IMSE department at Ohio University, an 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

APPENDIX ...................................................................................................................... viii

LIST OF FIGURES ........................................................................................................... x

1 Introduction .................................................................................................................. 1

1.1 Application Integration .......................................................................................... 1

1.2 Motivation ................................................................................................................. 1

1.3 Difficulties in Data Exchange ................................................................................ 3

1.4 Why XML for Data Exchange / Application Integration? .................................... 3

1.5 Research Goals ........................................................................................................ 4

1.6 Summary of Data Exchange Operation .................................................................. 5

1.7 Structure of the Thesis ............................................................................................ 6

2 Background and Related work .................................................................................. 8

2.1 Knowledge-Based Systems .................................................................................... 8

2.2 The 3I-PP Knowledge-Based System ..................................................................... 9

2.2.1 Knowledge base and data base ......................................................................... 10

2.2.2 Feature Completion module ............................................................................. 11

2.2.3 Process Selection module ............................................................................... 11

2.2.4 Part Models ...................................................................................................... 11
4.3.5 Document rules ................................................................. 38

4.4 Interpreting XML Documents ............................................. 38
  4.4.1 DOM parser ............................................................... 38
  4.4.2 SAX parser ............................................................... 40

4.5 Feature Display ............................................................... 42

4.6 Process Visualization ....................................................... 44

4.7 Presenting and storing XML documents ............................ 44

5 Implementation ..................................................................... 46

5.1 Generating XML Document ................................................ 46
  5.1.1 XML Body, Header and Footer ..................................... 46
  5.1.2 Writing Element Tags ................................................ 47
  5.1.3 Dynamic object type headers ....................................... 47

5.2 Data Exchange Interface ................................................... 48
  5.2.1 Connecting to 3I-PP on remote server .......................... 50
  5.2.2 Loading 3I-PP ............................................................ 50
  5.2.3 Loading Example (part model) ..................................... 51
  5.2.4 Retrieving features for the part model ......................... 51
  5.2.5 Getting data for features ............................................ 52
  5.2.6 Get processes for features ......................................... 52
  5.2.7 Creating feature Java object ....................................... 53
APPENDIX

[A] Screen shots of different stages of data exchange mechanism ............... 68

I. Connecting to 3I-PP server .................................................................. 68

II. Loading 3I-PP Knowledge Based System ........................................... 68

III. Loading NetExample Part Model and Obtaining List of Features ......... 69

IV. Generating Feature Data for THOLE_F6_1 feature ............................. 69

V. Generating List of Processes for LSLOT_F5_2 feature ....................... 70

VI. Creating Feature Objects ..................................................................... 70

VII. Showing Process Animation ............................................................... 71

VIII. Displaying XML on the browser ...................................................... 71

[B] Samples of XML documents generated ............................................. 72

I. XML document generated for a feature ................................................. 72

II. XML document generated for a process ............................................... 73

III. XML document generated for a feature including all processes ........... 74

IV. XSL file written for showing XML data in browser ............................ 78

[C] Samples of code for implementation of Data Exchanger .................... 79
I. Sample code for a Socket Connection on the Lisp and Java Side ....................... 79

II. Snippet of Lisp code for generating XML document ........................................ 80

III. Snippet of XML DOM parser code ................................................................. 81

IV. Snippet of XML SAX parser code ................................................................. 82

[D] Input data files for 3I-PP system..................................................................... 84

I. Snippet of feature file for a part model loaded in 3I-PP ..................................... 84

II. Snippet of part file for a part model loaded in 3I-PP ....................................... 87
LIST OF FIGURES

Fig. 2.1 Geometry of NetExample model ................................................................. 12
Fig. 2.2 An example XML document ....................................................................... 18
Fig. 2.3 Child elements of XML document ............................................................... 18
Fig. 3.1 Architecture Of Process Visualization System ........................................... 21
Fig. 3.2 Data Exchange Architecture ....................................................................... 25
Fig. 4.1 Approaches for communication between Lisp and Java application ........ 27
Fig. 4.2 Data for features of NetExample model generated from XML document .... 29
Fig. 4.3 XML Writer Procedure .............................................................................. 32
Fig. 4.4 Body of XML document generated by XML writer ................................... 35
Fig. 4.5 A sample XML document generated by XML writer ................................. 35
Fig. 4.6 Different XML document object type headers .......................................... 37
Fig. 4.7 SAX / DOM Selection ................................................................................ 42
Fig. 5.1 An XML Writer function to generate body of XML document ................. 46
Fig. 5.2 Functions for writing header and footer of the XML document .................. 47
Fig. 5.3 Writing XML element tags ........................................................................ 47
Fig. 5.4 Returning object type for XML document header ..................................... 48
Fig. 5.5 Generating XML document object type header ....................................... 48
Fig. 5.6 Initial state of Data Exchange Interface ..................................................... 49
Fig. 6.1 Screen Capture of Data shown in process frame of NetExample model ...... 57
Fig. 6.2 Process data inside JTree node .................................................................. 58
1 INTRODUCTION

1.1 Application Integration

The application interfaces being developed for industry functions differ from system to system due to the lack of a standard data format for the use in representing information. The need to tie together incompatible systems has made application integration a main focus of research. Successfully executed, such research can combine the data or information from one application with data or information of another application to provide real-time integration.

This research describes the development of an automated data exchange mechanism that can provide data mobility between a knowledge-based system and Java applications leading to an integrated system. The systems used for this research consist of several independent or semi-independent modules, each performing a particular task in the product development cycle. Some examples of modules are feature modeling, feature recognition, tolerance modeling, process selection, machine selection, cost estimation and sequencing. In this research, the focus is on the machining process selection module in a 3I-PP system and process visualization in the IMPlanner system.

1.2 Motivation

3I-PP (Integrated, Incremental, and Intelligent Process Planner) [5], which is a Lisp-based process planning expert system, uses its extensive knowledge-base for feature
completion and process selection, and its space search algorithm for process sequencing. Some of the limitations of the 3I-PP system are cited in [5]. The 3I-PP system is not capable of showing a process-manufacturing operation visually. Additionally, the feature file data needs to be entered manually in the 3I-PP, which prevents automation for different part models. Since the 3I-PP system is developed in Lisp, support for animation and 3D visualization is not provided. Also, the limited support of Lisp for web applications makes it inappropriate for developing future web-based applications. One other major disadvantage of using Lisp is the incompatibility of its source code with those used on different Lisp processors or tools of choice (e.g. LispWorks GUI routines cannot be used within the Franz Lisp Environment).

Java, a language used in production and automation systems, provides enhanced functionality such as interoperability, animation and web deployment, to our current software applications. Applications that are developed with Java can use data from the 3I-PP system to show visualization of process plans in Java applets, making it also possible to view animation within browsers across networks. Also by using Java, products feature geometry can be graphically represented, enabling better understanding of a product and a process by the user.

These facts have motivated us to develop an automated data integration mechanism that will use the existing data from the 3I-PP knowledge base in Java-based applications. This will allow us to use the extensive knowledge base of the 3I-PP legacy system, already present, and at the same time, develop interoperable new applications using the tremendous potential of Java.
1.3 Difficulties in Data Exchange

Currently, it is very difficult to find a complete off-the-shelf tool that will allow data exchange mechanism between Lisp applications and Java applications. The difficulty in developing such a mechanism is the fact that the data is tightly bound to the applications that created the data, which means a conversion or translation process is required to move data between applications.

The data generated by the 3I-PP system is rudimentary data in Lisp format. However, it is required by the process visualization module of the IMPlanner system that is developed in Java environment. The problem is that the 3I-PP system uses the Lisp format to store data whereas the IMPlanner system uses Java objects. Since the two modules are implemented in different formats, a common data format is needed that can be understood on both sides. Therefore, an interface is required that implements data exchange using a common data format to allow efficient mapping of data from Lisp object model to Java objects.

1.4 Why XML for Data Exchange / Application Integration?

XML is becoming widely accepted as the neutral and standard data format for information storage. It not only describes how to show data but also describes the content of the data. Therefore, interchange of data between disparate systems will become simpler using XML. The data present in one legacy system could be written and stored in XML natively. This data could then be accessed by other systems using standard XML
tools / parsers developed for other systems. One of the prime reasons for the tremendous growth of XML for data exchange is its wide support by W3C (World Wide Web Consortium) as an open industry standard for all application integration issues. Because of this, XML is being selected and used by everyone for data representation/information storage. This should also make it easier to find tools, libraries and future applications compatible with currently developed systems.

1.5 Research Goals

The goal of this thesis is to develop an automated data exchange mechanism capable of transferring data between the 3I-PP and IMPlanner systems and, thus, making data from the 3I-PP visible to modules within the IMPlanner and vice-versa. The product representation file would be loaded from the IMPlanner to the 3I-PP. After receiving the file, the 3I-PP would generate manufacturing processes and send them back to the IMPlanner for further processing. Upon receiving the information must be sent immediately in real time requests from clients. Granular exchange of data (which means different levels of detail) is needed for sending only the required information from server. Thus it will not be necessary for the 3I-PP server to dispatch an entire subsection of a generated process plan to a client every time there is a request. This mechanism should allow data mobility between systems on remote computers.
1.6 Summary of Data Exchange Operation

A summary of operation of the system is given below. The knowledge server in the Lisp environment and Java applications within the IMPlanner will start as two separate processes, which may be on two different platforms. The first action is the request from the IMPlanner to load the knowledge base of the 3I-PP into the knowledge server. The second action from the client is product representation loading into the 3I-PP environment.

Next the IMPlanner requests feature data from the 3I-PP, which calls the XML Writer that generates the XML stream from the 3I-PP data and sends it via TCP/IP sockets to the client system. The client system regenerates feature objects and presents them to the user. Upon inspecting feature data in a visual environment, the user on the client side decides the feature on which he/she wishes to perform process selection. The client then sends the request to the knowledge server, which performs all the necessary verifications and computations (according to its knowledge base) and generates process candidates for the feature. For example, in the case of a slot, end milling and side milling, instances may be generated, or in the case of pockets, processes with different tool dimensions may be generated. The server notifies the client side about the results.

The user on the client side may then select an individual process candidate and request its data. The request is sent to the server. It generates an XML stream for a given object using the XML Writer and sends it back to the client. The client parses the stream, generates the necessary process plan objects and may directly perform the visual
animation of the selected process. The procedure may be repeated for other features and process candidates. Feature data may be modified and the request for repeated process selection can be performed upon a user’s request.

1.7 Structure of the 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 data exchange tools, difficulties in data exchange, XML as a tool for application integration, research goals and brief summary of the data exchange mechanism developed.

Chapter 2 provides the background work related to this research and provides an explanation of the technologies and terminologies used in this research. It describes recent research in areas covered by the thesis. It also explains knowledge-based systems and in particular the 3I-PP knowledge-based system used for this research.

Chapter 3 presents the architecture of a process plan modeling and visualization system (IMPlanner), a Java client in this research that is used for data exchange with the 3I-PP system. It also describes the complete data exchange architecture that was developed for exchanging data between the IMPlanner and the 3I-PP system.

In Chapter 4 and 5, the complete methodology of stages in the development of the data exchange mechanism and its implementation are described.

Chapter 6 describes test cases with various complexities involving different modules of the IMPlanner system used for verification of the modeling and visualization system.
Finally, Chapter 7 provides a conclusion for this research and suggests future work related to this area. It also gives some insight to possible extensions of this research work for different applications.
2 BACKGROUND AND RELATED WORK

In this section, we review recent research in the areas covered by the thesis. First, the knowledge-based system used in this research is explained. After that the various approaches, pursued by the industry for building data exchange mechanism for integration of disparate and heterogeneous information systems are discussed. The section ends with the description of some of the related terminologies and definitions used in the research.

2.1 Knowledge-Based Systems

As products increase in complexity, as product development becomes more distributed and as software tools cover a broader spectrum of product development activities, the need for an effective means to formally capture additional types of knowledge become essential. A knowledge-based system consists of a set of rules and user-supplied data that interact through an inference engine from which new facts or data can be derived or deduced from existing facts and conditions. Thus, a knowledge-based system (expert system) could be defined as a knowledge-based computer program that provides "expert quality" solutions to problems in a specific domain [2]. Generally, knowledge is extracted from human experts in a domain and attempts to emulate their methodology and performance.
2.2 The 31-PP Knowledge-Based System

The 31-PP knowledge-based system is used for this research. Šormaz, D. et al., had described the architecture of the 31-PP in [5]. The 31-PP system generates process plans by accepting the product description in the form of a set of features that specify geometrical, tolerance and material information for the part.

The system consists of the following parts: knowledge base, database, feature completion module, process selector, and process sequencing module. It is implemented as a hybrid rule-based and frame-based system utilizing an object-oriented approach for describing process planning related entities and the relations between them [6]. The system has been initially implemented using the KnowledgeCraft™ environment and its underlying language, Lisp, is used for the implementation of object methods. The machining knowledge is represented as a hierarchical tree of frames (schemas in KnowledgeCraft terminology) for features, machining processes, machines, and tools. Data-dependent machining knowledge is implemented as methods for the frames and is inherited by all children frames. Relationships between features, machining processes, machines and tools are implemented at an appropriate level of abstraction. The relation inheritance and transitivity are defined using KnowledgeCraft's CRL language. Since the object-oriented approach is utilized, the resulting process plan is represented as a set of interrelated objects. These objects are stored in the system as the process plan network of alternatives, making it most suitable for integration with scheduling and a shop-floor control system.
Process sequencing is implemented as a state space search algorithm. However, states and state transitions operate on frames, and the final process plan is stored as a connected sequence of frames. The process clustering procedure is embedded into the space search algorithm in such a way that it is executed on a pre-selected set of machining processes defined for the current state.

A menu-driven user interface is created using LispWorks library for GUI development. The menu system allows easier additions to the feature model. It also allows the user to monitor and inspect the system performance at various phases of process planning.

2.2.1 Knowledge base and data base

The knowledge base consists of feature knowledge base, process knowledge base, machine knowledge base, and tool knowledge base. For each category of objects a hierarchy of classes is built by including more specific knowledge on the lower levels of the hierarchy. Methods that include manufacturing rules and decisions are incorporated at the appropriate level of specialization.

The database is connected to the knowledge base in such a way that data represent individual instances (objects) of classes from the knowledge base. This assures that objects inherit all of the related knowledge through methods that belong to their parent classes. The data base consists of necessary data for an individual part for which the process plan is being generated, and encompasses: feature model, process candidates,
machine constraints, tool constraints, cutting parameters, and process activities. Separate databases are built for the library of available tools and machines.

2.2.2 Feature Completion module

The feature completion module accepts the feature model of the part from the feature recognizer and completes it for process planning. The feature completion procedure consists of two parts. In the first part, each individual feature is considered and its data are augmented to include explicit data necessary for process planning. The second task of the feature completion module is the creation of a feature precedence network. For this task the system performs an analysis of feature interactions.

2.2.3 Process Selection module

The process selection module uses the augmented feature data and generates all available process instances for the features. The procedure starts by generating all possible process instances for various process types that may be used on the feature. After that process capability is verified for each instance by comparing its tolerance capability data with feature tolerance requirements. Only instances that pass this verification are kept as possible alternative processes for the analyzed feature.

2.2.4 Part Models

Paper [5] shows several part models like Bendix, NetExample and Testnik, where this process planning system is used for generating process plans. Fig. 2.1 shows the geometry of NetExample model used for this research.
2.3 Previous Work in Data Exchange

Independent applications may exchange data in several ways: through file interface (one module writes file and the other reads it), database interface (both modules access the same database), object interface (the modules exchange objects using object exchange protocol like CORBA or COM), or XML interface (the modules exchange tagged text streams).

Exchange of data where applications use file interface for storage of information is shown in [21]. Paper [4] presents a shared database approach for storing data to database. The traditional object interface approach using CORBA is described in [22] where data are exchanged as objects. Papers [11] and [15] show how the XML-based approach is used for data representation.

Ash, D., et al. [11] showed how XML could be used as a Language Interface for AI Applications. The paper shows how developers can design their own languages suited to particular AI problems allowing a seamless interface with the rule engine. The input,
output and rules for an AI application are given using XML files. To solve new problems a developer must change the language used for solving the problem. The commercially available product ART*Enterprise offers cross-platform support for most operating systems and seamless integration with programming languages like C, C++, Java and CORBA. The HTML request, after being processed by servlet, is passed as an XML document and is finally converted to ART*Enterprise objects which are then loaded into the ART*Enterprise application. This work shows how XML could be used for AI applications but fails to explain the communication mechanism or the data exchange intricacies between different systems using XML.

Burkett, W., [4] describes the use of Product Data Markup Language (PDML) for product data exchange that facilitates the integration and interoperability of business processes across organizations for process control and product data management. It uses Internet for connectivity and the basic structure of STEP for designing the Integration Schemas needed to exchange product data. STEP is the principal data exchange standard in the world, pioneering in the interpretation and use of generic data structures in different application domains [4]. To solve data redundancy and duplication problems a procedure to obtain an integrated, accessible, and consistent data is suggested. Data repository that allows users to access and import product data into their own applications is created. This solution shows how a shared database should be created for use in multiple applications; however, it does not focus on data that changes dynamically at different stages of application. Also this paper assumes communication mechanism
needed to interact between disparate systems to be present with existing Internet Protocols and Languages.

Hayes, C., et al. [15] presents an architecture in which case-based intelligent applications could be distributed over network. They proposed Case Based Markup Language (CBML) for representing data in the form of cases using XML. Using cases based on the proposed CBML standard allows other applications such as Intelligent Agents to access large repositories of case storage. In addition, they present a distributed architecture for case-based reasoning (CBR) motivated by a need to move processing to the client side in order to improve interactive response times. The CBR engine is downloaded to the client side in order to allow processing of cases on the client side in later stages without further interaction across the network. This paper only describes how cases could be represented using XML while working across network, but fails to describe the communication model for using XML between intelligent assistants that use CBR and other applications.

Limitations of the CBML approach are discussed by Hayes, C., and Cunningham, P., [16] in their work on presenting a standard case view in XML, making a case for techniques that can integrate easily with existing mark-up structures. They show that when using CBML, the data must be marked up in CBR specific format. Instead of using generic DTDs they propose Namespaces and Schemas that provide the appropriate CBR perspective on the XML data.
Franz Inc.'s Jlinker tool [23] develops a dynamic link between Lisp programs and Java class libraries. The communication between Lisp and Java is achieved using remote interfaces supported by a proprietary socket connection. It allows dynamic un-pre-mediated access to Java objects and methods from the Lisp runtime environment. The end result is that the Lisp application may call Java methods as if they were Lisp functions. This approach does not handle problems where Java applications need to call Lisp methods. Also the features implemented are available only with Allegro CL 6.0. Furthermore, this work assumes client-server interface because of presence of proprietary socket connection.

Franz Inc. discusses how Allegro ORBLink enables Common Lisp developers to create CORBA-compliant applications using Allegro CL [22]. The Object Request Broker (ORB) manages the communications between client and server using the Internet Inter-ORB protocol (IIOP). The Allegro ORBLink is an in-house product that helps in automatic mapping of the CORBA IDL into Allegro CL. The CORBA interface definitions are written in a file, compiled into two phases: first into a Java stub on the client side using a Java CORBA compiler and then into a Common Lisp skeleton on the server side using Allegro ORBLink. This work enables truly interoperable distributed applications with support for diverse platforms and existing systems but requires Allegro CL and Allegro ORBLink for implementation.

Menke, L., and Miller, W., [21] developed the B2 Interface and Server. The B2 Interface has a Java client that communicates with the Java server that, in turn, communicates with a Lisp process by grabbing the process's standard input and output
streams. The Java client creates a socket to communicate with the server. In the implementation of a server, a Lisp process is created for each client and communication is accomplished through streams. The thread is then started to allow the server to begin listening for connections. Once a connection is accepted, the server then creates a new socket object at a different port that then communicates with the client at the new port created. This approach uses an additional Java server to communicate with the Lisp process, which adds complexity to the problem. Also, the activities such as sending a message to the server to open, to read and then to send back the information are all done using files. More then one stream is needed to transfer all this information. Since this data is unstructured, it needs to be parsed after it is received on the client side.

After analyzing previous work and current technological advances in the field of data exchange, XML interface appears ideal because of its data independence and usage with different languages. By using XML, data is no longer dependent on a specific application for creating, viewing, or editing. XML allows data to be used by any application.

2.4 Overview of XML technology

The Extensible Markup Language (XML) is a simple, very flexible text format derived from Standard Generalized Markup Languages (SGML) [24]. It was developed by the World Wide Web Consortium (W3C) XML Working Group in 1996 to facilitate easy transmission of structured data over existing network protocol [13]. It is not a markup language. Instead, XML is a set of rules for creating new markup languages. As compared to HTML, XML tags not only describe the data but also include information
about the data. So it is easier for a computer to understand the meaning of the tags. Also by applying a different Stylesheet to the same XML document, an XML document can be rendered in different formats. The key is that with XML the information is in the document, while the rendering instructions are elsewhere. This means that content and presentation are separate.

XML’s strongest point is its ability to do data interchange [3]. Because different applications rarely have the same standards on a set of tools, it takes a significant amount of work for two groups to communicate. XML makes it easy to send structured data across different systems so that nothing is lost in translation. If the data being sent is structured with XML, it is much easier for the computer to understand exactly what the data means and how it relates to other pieces of data. In other words, two applications can exchange data between them, and neither of them needs to know how the other system is organized. Additionally, both applications are insulated from any changes in the architecture or data that might occur in the other system.

XML changes the way data moves across networks. XML encapsulates data inside custom tags that carry semantic information about the data. By formatting our data in a markup language, we allow computer applications to process and present this data to us in different views. The processing of data into multiple views is the perfect role for a Java-powered client that receives the XML-formatted data from the server. This is the key idea behind Jon Bosak’s prophetic statement, “XML gives Java something to do” [12]. We come to conclusion that XML makes it much easier for two computers to exchange data.
2.4.1 XML document structure

The first line in the document - the XML declaration, defines the XML version as shown in Fig. 2.2. The next line describes the root element of the document <books>.

<?xml version='1.0'?>
<books>
    <book category = "farm">
        <author>Greg Campbell</author>
        <title> Last Garlic</title>
        <price>12.95</price>
    </book>
    <book category = "computers">
        <author>Jack Stewart</author>
        <title> Networking Chains</title>
        <price>14.15</price>
    </book>
</books>

Fig. 2.2 An example XML document

The next line describes the child element of root called <book>. The root element can have multiple <book> elements. The next 3 lines describe 3 child elements of the <book element> as shown in Fig. 2.3.

<author>Greg Campbell</author>
<title> Last Garlic</title>
<price>12.95</price>

Fig. 2.3 Child elements of XML document

The next line defines the end of book element </book>. The next 5 lines similarly describe another book element. And finally the last line defines the end of the root element and the XML document </books>. In the above example, one piece of data is string 12.95, and it represents price of an element to which it belongs. Thus, from this
books example we see that the XML document simply self-describes itself. So XML documents carries both data and the meaning (or semantics) of data.

2.4.2 Components of XML

An XML-compliant document can be separated into markup and content. The markup is meta-data or the data that is outside the original data; it is present for the purpose of enhancing or clarifying the original data. The content is the original data on which markup is used. In the above example as shown in Fig. 2.3, <author>, <title> and <price> makes the markup. And the data inside these elements Greg Campbell, Last Garlic and 12.95 makes the content. The elements and attributes represent the key indicators of the structure or the purpose of the content. An element is a logical construct of a document. A normal element is composed of start and end tags that surround content, other elements, or both. The tags of an element are delimited by angle brackets. The elements in the above example are <books>, <book>, <author>, <title> and <price>. An element may have attributes that are specified in name/value pairs and are placed inside the start-tag name. Attributes allow attaching characteristics to an element [1]. For instance, an attribute in the above example would be category in <book> tag. Attributes values must always be enclosed in quotes as “farm” and “computers” shown in Fig. 2.2.

2.4.3 The XML Stylesheet Language (XSL)

In order to display XML documents, it is necessary to have the mechanism to describe how the document should be displayed. XSL is a language that can filter and sort XML
data, transform XML into HTML, address parts of an XML document and format XML data based on data value. It is a standard recommended by the World Wide Consortium.

XSL consists of three parts: a method for transforming XML documents (XSL Transformations-XSLT), a method for defining XML parts and patterns (XPath) and a method for formatting XML documents (XSL Formatting objects). XSLT and XPath are used in this research for showing XML file in browser.
3 IMPLANNER SYSTEM ARCHITECTURE

3.1 Architecture of Process Plan Modeling and Visualization System (IMPlanner)

The overall structure of the IMPlanner system as presented in details by Šormaz, D., Arumugam, J., Borse, P., Jain, S., & Thirupalli, S. [10] is described in Fig. 3.1. It shows how the 3I-PP and visualization systems are integrated together.

Fig. 3.1 Architecture Of Process Visualization System
It is important to discuss the architecture of the IMPlanner system here since it is the core module used for integration in this research work. The properties of this IMPlanner system are:

1) It relies on existing software tools for CAD/CAM and CAPP,

2) It provides for distributed processing of process plans across enterprise and virtual enterprise in intranet/internet environment,

3) It utilizes available technology and emerging standards for internet computing (namely Java and its tools JNI, Sockets, and RMI), and

4) It utilizes standards for data storage and exchange (relational databases and XML).

Those properties enable evolutionary development of the system and incremental transfer of the technology into enterprises.

The dark shaded modules in Fig. 3.1 represent existing software tools. Two modules grouped with rectangle in the middle represent a process plan representation model. The three lightly shaded modules represent applications that are being developed in this project. Sharp rectangles on arrows represent interfaces utilized between the different modules. Also, because of the software development tools selected for this system (namely, Java, HTML, JNI and XML), all of these modules (existing and those under development) may exist on different computers with various operating systems, and on geographically distant locations.
3.1.1 System components

The main components of the system are as follows: a process plan representation, existing CAD and CAPP systems, component interfaces, machine/tool databases and process planning applications.

The process plan representation model is a key component of the system. It currently encompasses two related segments: a machining feature model and a process plan model. Both of these models are built using object-oriented modeling. For features and processes a hierarchy of classes has been implemented in order to provide smooth links with existing knowledge based systems.

Existing systems for design (CAD), and process planning (CAPP) are linked into the system in order to utilize existing expertise in geometric modeling, part programming and/or process planning built into these systems. Each existing system provides specialized knowledge in a given domain and implementation of desired functionality. For example, any system need related to geometric computations on solid model may be satisfied by a CAD system (current prototype is built for Unigraphics), manufacturing process selection knowledge may be supplied by the 3I-PP system, and so on.

Component interfaces provide means for transferring data between modules, for remote invocation of methods, and for communication of modules through the Internet. For each application the most appropriate method is selected. For example, the TCP/IP protocol is selected for transfer of process selection results from the 3I-PP to process visualization.
The visualization module exchanges data with the 3I-PP supplied process plan representation, which consists of information about the selected process for a feature, corresponding machine, tool and cutting parameters and process plan sequencing for the part. The visualization system interacts with the user by displaying the selected process with the process parameters for a feature.

The approach used in this architecture is based on utilizing existing tools on both fronts: special purpose applications, and distributed communication and collaboration. Thus, it provides very a flexible environment for different users of process plans.

3.2 Data Exchange Architecture

The architecture proposed for building a data exchange mechanism between the IMPlanner and the 3I-PP Knowledge Based System [7] is shown in Fig. 3.2. The communication between the 3I-PP and the IMPlanner is achieved using a TCP/IP socket connection. Upon receiving the request from the process visualization module, the Lisp server generates corresponding data using the 3I-PP process planner. The data exchange mechanism gives control to the 3I-PP to send only data that is relevant for the current request. The XML Writer captures the data and, then, generates appropriate the XML document stream. This XML document stream is sent to Java client side through sockets. At the client side, either a SAX or a DOM parser parses the XML document from the InputStream. The client decides which parser to use depending on the result required. The DOM Parser is used first for parsing the XML document. It loads the whole XML document into its memory and creates its model displayed in a Java JTree model.
Fig. 3.2 Data Exchange Architecture

The SAX parser is used for parsing the XML document and converting it to Java objects. When a particular feature or process is sent from the Java application to the Lisp Server, then Lisp should generate the appropriate XML document for that feature. Attributes of that particular feature should be accessible on Java side. These objects are then used by different Java client applications developed using the JFC/Swing. The metadata exchanged over streams is shown on the web browser using the XSL Stylesheet.
This interface allows users to control the 3I-PP knowledge based system for any particular feature from the IMPlanner system.
4 METHODOLOGY

4.1 Data Exchange Approach

The following four approaches, shown in Fig. 4.1, using client/server architecture were studied for communication between Lisp and Java application.

![Four Approaches for Communication between Lisp and Java Application](image)

**Fig. 4.1 Approaches for communication between Lisp and Java application**

The four approaches differ in the way they communicate between the Lisp and Java applications. The first two approaches need a Java client or a server on the Lisp side whereas the third and the fourth approach do not. Also, in the third approach we have the
Lisp server communicating with the Java client whereas in the fourth approach, it is the Lisp client communicating with the Java server.

After investigating all four approaches the third approach, with the Lisp server and the Java client, was selected. The server was on the Lisp side since the 31-PP process planning system would have to frequently respond to client request. The Java application would act as a client side and would make requests to the server for parameters details.

We rejected the first two approaches since to have communication between Lisp and Java applications; an extra Java module on the Lisp side is not required. Also our approach was more challenging and more generic to real-life cases.

A socket connection was selected for communication between the Java client and the Lisp server. A fixed port was chosen for making this connection. Also the IP address or server name needs to be specified for communication between the two remote PCs.

Once the communication mode between the Lisp server and the Java client has been determined, the next question is how to send data from the Lisp to Java. In the past, strings and Lisp lists have been used for storing and sending these data. But these methods would be very long and tedious, since they need to be parsed and saved on the Java side. Therefore, XML technology, which facilitates the traffic of complex hierarchical data structure over network applications, was used for exchanging of data between the 31-PP and the IMPlanner system. However, in order to send requests from Java to Lisp we used simpler approach, we send Lisp formatted lists from Java. The reason for this was that lists are the most natural way the Lisp can interpret generic input.
To achieve the task of developing an automated data exchange mechanism, the following approach divided into six distinct phases, was selected:

(1) Establish interactions between the 3I-PP and the IMPlanner

(2) Create XML documents within the 3I-PP application (XML Writer)

(3) Send the XML document to the IMPlanner applications using sockets

(4) Convert the XML document to a Java object model (XML Handler)

(5) Work with the Java object model for use in different JFC/Swing applications

(6) Build an interface allowing data exchange control at granularity level

In order to verify feasibility of this approach, a sample XML document was created from data generated by the 3IPP process planner for a given feature and was shown in the browser using XSL. Fig. 4.2 shows sample data generated from the XML document for one of the features of NetExample model.

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTANCE</td>
<td>CYL-BLIND-HOLE</td>
</tr>
<tr>
<td>DO-AFTER</td>
<td>Cocket_F4_1</td>
</tr>
<tr>
<td>MAY-BE-MACHINED-BY</td>
<td>CORE-MAKING</td>
</tr>
<tr>
<td>AXIS #</td>
<td>(1.0 0.0 0.0)</td>
</tr>
<tr>
<td>ACCESS</td>
<td>(1.25 ; #(1.0 0.0 0.0))</td>
</tr>
<tr>
<td>RADIUS</td>
<td>0.125</td>
</tr>
<tr>
<td>BELONGS-TO</td>
<td>NETEXAMPLE</td>
</tr>
</tbody>
</table>

Fig. 4.2 Data for features of NetExample model generated from XML document
On the basis of these initial results, the decision to use XML technology for building a
data exchange system was further justified. This XML document was generated in Lisp.
Use of the XML Writer is explained later in section 4.3.

4.1.1 Data Exchange Modules

This section discusses procedures and methods used for developing an automated data
exchange mechanism. The automated data exchange mechanism connects two modules:
the manufacturing process selection module in the 31-PP system & the process
visualization module in the IMPlanner system using the Data Exchange Interface, which
interacts between these two modules. The data exchange interface is described in detail in
section 5.2.

4.1.2 Manufacturing Process Selection

The first module manufacturing process selection is based on data generated from the
31-PP system developed in Lisp/CLOS. It generates process plans by accepting the
product description given in the form of a set of features that describe geometrical,
tolerancing, and material information for the part. The description of this module was
presented earlier in section 2.2.

4.1.3 Process Visualization

The architecture of the second module process visualization used in this research is
described in detail in [9]. This work in the process visualization module of the IMPlanner
system is not a part of this thesis work, but is a core module used to perform integration.
It performs machining process animation and visualization using Java and Java3d APIs. The data structures from the Java classes (for feature, process and tool) are converted into geometric objects using Java3d utilities for generation of geometry in virtual 3D space. For process animation, the necessary behavior and time interpolation for changing the geometry are generated from the process data (cutting speed and feed) and a single timer object is used to generate realistic virtual machining process. The architecture provides flexibility to display every individual component (tool, feature, work piece) in 3D animation as well as 3D wire frame model. A prototype has been implemented so that it accepts data (tool, feature, work piece geometries) from knowledge based process planner and generates data driven animation in a distributed environment.

4.2 Inter-Process Communication

For Inter-process communication between Lisp and Java, sockets are used. Input and Output in Common Lisp and Java is done through objects called streams. When working under a Lisp environment, comm:socket-stream is called on a socket, which returns a stream that can be used with the standard common Lisp functions like read-line, read etc. The stream is buffered for both reads and writes, so when writing to the stream we have to use force-output on the stream to send the data over the socket [14]. Similarly in Java, reading and writing of stream is done using InputStreamReader and BufferedOutputStream respectively. Appendix C [1] shows a sample code for a socket connection on the Lisp and Java side.
4.3 XML Writer

The XML Writer is developed to capture needed elements and their attributes for exchange of manufacturing features and machining processes. In Lisp/CLOS the XML Writer is written to generate a XML document dynamically from the process plan data on the server side. The procedure developed by the XML Writer is shown in Fig. 4.3.
The implementation of the XML Writer is done in Lisp. Since the 3I-PP represents feature and process data as a set of interrelated objects, the XML Writer is implemented as a method for each object of interest (feature, manufacturing process, machine and tool constraints). The data content of the XML document is generated from the 3I-PP system for the object that corresponds to a particular request from a client. This data is wrapped between the XML header and the XML footer inside XML tags and nested at a proper location inside the XML document structure. The XML document is written step-by-step starting with the request, interpreting it and generating XML stream that represents the response to a given request. The data exchange mechanism begins with a request from the IMPlanner client. In order to handle only the required data these requests are classified depending on what level of detail they require. The XML Writer creates appropriate an XML document stream depending on these requests. The detailed working of the XML Writer is explained in following subsections.

4.3.1 Request types

After receiving a request from the Java side for a particular feature or process, the 3I-PP is used to obtain corresponding data. The request can be classified as

1. List of Features
2. Feature Data
3. List of Processes for a given feature
4. Process Data
The first request is to get a list of features for a particular part model, which is loaded into the 31-PP system. The second request is made when the list of features is available and the user wants to obtain data from any feature. This request will get complete feature data (like feature’s height, depth, axis etc) for the feature selected. The third request is to get list of processes for a selected feature. It is similar to the first request, but here the request depends on the feature selected by the user. The fourth request gets complete process data available for the process selected by the user. The implementation of each of the requests is explained later in section 5.2.4, 5.2.5 and 5.2.6.

The initial request is for List of Features for a given part model. From this request the user gets the option to perform further tasks on the Java side. The complete list of features is obtained from the 31-PP using (get-instances 'machining feature) function defined in 31-PP.

4.3.2 Simple example

The list of features for a NetExample part model inside the 31-PP is the following (thole_f6_1 bhole_f7_2, lslot_f1_1, lslot_f5_2, cpocket_f7_1). Now XML Writer can be applied to this list to make this list, the body of the XML document as shown in Fig. 4.4. In addition the header and footer of the XML document have to be added in front and back of this structure. The result will generate a well-formed XML document as shown in Fig. 4.5.
The procedure described above shows a simple example of obtaining data from the 3I-PP and then putting XML tags inside to generate an XML document.

### 4.3.3 Nesting of elements

The `<FEATURE>` element is nested inside `<DEFSHEMA>`, which are, in turn, contained within `<LISTOFFEATURES>`. Element `<DEFSHEMA>` is added only when a request for List of Features for a particular part model is made. The XML elements, thus, follow the basic law of physical containment. `<LISTOFFEATURES>` can contain complete `<FEATURE>` but a `<FEATURE>` cannot have `<LISTOFFEATURES>` inside
it. This nesting of elements shows the hierarchical structure of the XML document. This document conforms to all the rules required to be a well-formed document.

The `<FEATURE>` element can be expanded to include details about feature objects when requested. Appendix B [I] shows the nesting of elements within the `<FEATURE>` element. The `<param>` element nested inside the `<FEATURE>` element contains the parameters of the feature. These parameters of feature are added under the name attribute of `<param>` element. Their values are then added between start and end of the `<param>` element. In the case of parameters that represent 3D vector or point, the value is written in another element called `<param-vector>`. This element `<param-vector>` is written within `<param>` element, and has three attributes x, y and z for describing the vector data. Complete data from the 3I-PP system includes the `<param>`, `<param-vector>` and `<constraint>` elements.

The nesting of elements in the content model becomes more complex when the features include process data within them as shown in Appendix B [III]. Each of the process data is included under `<param-child>`. The `<param-child>` element is now added under the `<FEATURE>` element. To include data from multiple processes a `<PROCESS>` element is added under `<param-child>`. The `<PROCESS>` contains the same data structure as explained for a feature above.

The depth of data content still goes deeper when processes are involved. Few parameters inside the processes like machine-constraint, tool-constraint and cutting parameters contain more data inside them and, thus, objects needs to be expanded. These
objects are added within the <PROCESS> element. They are put instead of <param> tags when encountered. The data content inside these objects are added within their respective objects. We have three new elements <MACHINE-CONSTRAINT> that describes machining constraint for selection of each process, <TOOL-CONSTRAINT> for tool constraints and <CUT-PARAMETERS> for cutting parameters involved while machining. Each of these three elements is expanded and their parameters are added inside them. A new element <constraint> is added within the elements for expanding data, which are in vector type. The <constraint> element has three attributes name of constraint, operator and value. The whole hierarchy of nesting of elements, forming structure of the XML document is shown in Appendix B [III].

4.3.4 Element headers

Fig. 4.6 shows the different XML element headers having attributes NAME and INSTANCE-OF. Depending on the object requested, this header code would be dynamically created in XML document and used.

1. <FEATURE NAME = "LSLOT_F5_2" INSTANCEOF = "LIN-OPEN-SLOT">
2. <PROCESS NAME = "P42-LSLOT_F5_2-END-M-CNC-H-END-M" INSTANCEOF = "END-MILLING-SLOTTING">
3. <MACHINE-CONSTRAINT NAME = "M36-LSLOT-CNC-H" INSTANCEOF = "CNC-HORIZONTAL-MILL">
4. <TOOL-CONSTRAINT NAME = "T36-LSLOT-END-M" INSTANCEOF = "END-MILLING-TOOL">
5. <CUT-PARAMETERS NAME = "CUT-DATA55" INSTANCEOF = "CP-END-MILLING-SLOTTING">

Fig. 4.6 Different XML document object type headers
Appendix B [III] shows use of all object headers in a single created XML document. Each feature is then subjected to a series of methods to obtain process and machining data.

4.3.5 Document rules

The XML document generated satisfies all XML-tag rules for a Well-Formed XML Document. The data formats defined in the 3I-PP system are used for generation of XML data. Since the data format is fixed and we only need to transmit well-formed document to the XML parser on the Java side, the XML schemas are not needed.

Thus, all the data from the 3I-PP system to the IMPlanner system are sent via the XML documents generated by the XML writer. The type of data written into the XML document is controlled by the user who has the option to send a selected list of features or processes, feature data or complete hierarchy of related objects.

4.4 Interpreting XML Documents

The XML document, passed to the IMPlanner system is interpreted by two parsers: either the SAX (Simple API for XML) or the DOM (Document Object model) parser.

4.4.1 DOM parser

The DOM, a standard of the World Wide Web Consortium, creates a tree view of the XML Document parsed. The DOM provides standard functions for manipulating the elements in the XML document. The DOM parser is used to provide an overview of all the features or processes and show them in the JTree structure since it creates a tree-
based representation for the data inside the XML document. This allows for easier selection of features and processes from the JTree. Each data element of the XML document is added to the JTree as a child node. The value in each element of the XML tags is added under the parent node. Since the DOM parser keeps the whole XML document structure in memory, all the elements of the XML documents can be accessed at a later stage. The DOM object is instantiated using the Sun ProjectX parser. DOM 1.0 interfaces are used as defined by W3C (org.w3c.dom package). Once the DOM object is instantiated, the W3C interfaces are used to access the DOM object. The XML Document object (com.sun.xml(XmlDocument) created by Sun parser provides several static methods for creating a DOM document from an InputStream. The root of the inheritance structure for this DOM document is the Node. All other interfaces are descended from Node and inherit a number of its method. So a document object is a simple container of nodes. To extract information from the document object, we first ask the document object for all the elements with process tag. The element objects obtained are then iterated to get the attributes inside each child node. The attributes and elements object are then added to TreeModel in Java. The TreeModel interface allows easy access to data inside the DOM document via JTree.

The DOM parser handles the data coming from all four of the requests mentioned above i.e. to get list of features, features data, list of processes and processes data. Requests for feature or process objects fall under the second and fourth request i.e. feature data and process data.
4.4.2 SAX parser

The SAX parser fires events based on relevant data it encounters while reading the XML document in order to create a particular Java object. Any data not specifically stored is discarded. Since the SAX parser does not come with a default object model representation for the data in the XML document, these objects are stored in the IMPlanner object model written in Java. These feature and process objects are stored in the corresponding JTree nodes. Different IMPlanner modules then use these Java objects for visualization and animation of the process plans. The SAX parser extends the HandlerBase class (org.xml.sax.HandlerBase) as a listener for the various events fired by the SAX parser, depending on the object data received from the XML document. The Java object model used with the SAX parser is described further in sections 5.2.7 and 5.2.8. We have used the SUN XML parser (available in com.sun.xml.parser) for this purpose. The InputStream is passed to the InputSource class (org.xml.sax.InputSource), which encapsulates the character stream coming from 3I-PP server side. The created SAX parser reads the XML document and fires events as it encounters start Element tags, end Element tags and character data. These events are fired as the SAX parser reads the XML document from top to bottom, a tag at a time. A document handler is used as an interface between the SAX parser and the IMPlanner objects on the occurrence of these events while reading the XML document. This will invoke the appropriate methods in the correct sequence on the process class to construct IMPlanner objects and populate their data. In short, the SAX parser triggers the XML document into sets of events, which are then translated into a corresponding set of method calls, defined in the parser class. The
Interface handler is registered with the parser using setDocumentHandler method. We use five methods startDocument (documentName), endDocument (documentName), startElement (processName, attributes), endElement (tagName) and characters (attributeName) available in HandlerBase class. In the startElement method, if it encounters a new Process tag, it will create a new Process object. In between the startElement and endElement methods, the values we encounter, i.e. the data that is not markup are sent to characters method for retrieving its value. These values are also added in the process object inside each corresponding element.

The endElement method is called when the SAX parser encounters a close element tag. It will then create a string object and place it inside the process object. The switch between the DOM and the SAX parser is provided to the user depending on the data requested from the 3I-PP system. The selection criteria for the switch between the SAX and the DOM parser are shown in Fig. 4.7. The incoming XML document is parsed either by the SAX or the DOM parser as shown in Fig. 4.7. The DOM parser is selected when we need to go deeper into data structure of the 3I-PP system. This data is added to the JTree that resembles data stored in 3I-PP the system. This allows the user to have complete access to data present in the knowledge base of the 3I-PP system. From the JTree, user can select to obtain more data about the node or obtain the object of that node. If the user wants feature or process objects to be created, the SAX parser is then called. The SAX parser obtains only relevant information required by the user that is stored then in the form of Java objects.
4.5 Feature Display

The feature data obtained from the 3I-PP that contains part design information in the form of XML document is used to create feature objects. The parsing of data for creating feature object is explained below. The SaxFeatureHandler class handles the parsing of the XML document for creating the feature object. It extends HandlerBase class to use its different methods available for parsing the XML document. In the startElement method, if the element starts with <feature> tag, it checks whether it ends with Hole, Slot, Slab or Pocket. Depending on the tag it encounters it creates instance of Hole, Slot, Slab or Pocket feature object. In the startElement method if the element tag starts with <param>,
then it gets the value of the element and writes function, which will set its value to the current feature object created. The element can also start with <param-vector>. In this case, the parser knows that it needs to expand the element. The parser gets the value of the element in vector3d form and writes function, which will set its x, y and z value to the current feature object created.

Once the instance of feature object is created and functions are written to set the data obtained from the 3I-PP to the feature object, the next step is to call these functions, which will set the required elements and vectors. Methods are written in each of the feature classes, which will set these parameters for the feature. If the element is “RADIUS” then its value is set to the feature object using setAttribute. Similarly, for vectors “AXIS” and “PO”, setVector method is written in corresponding feature classes.

After parsing of the XML document, the feature object opens in a new Java applet. The user can select to continue further interaction with either feature applet or data exchange applet. Through this feature applet, a user can create a new feature file or enquire about existing feature files. A user can also change information about a current feature file and view the feature inside the graphic interface of the applet, making new modifications. This visualization tool allows the user to view the model from different viewpoints and at varied zoom factors. The work done in feature display module of the IMPlanner is not a part of this thesis work. The feature display module is described in detail in [6].
4.6 Process Visualization

After retrieving the data for process, feature and work piece geometry from 3I-PP as explained above, the process plan objects can be created. The parsing of data for creating process object is explained below. Unlike feature object, the process object is created when the end element is encountered, since no blank constructor is created for process object.

After the creation of the process plan objects, the control for displaying process is passed over to them. Using the data supplied to these objects, visualization of each component of machining process can be provided. The components such as work piece geometry, before and after machining, a slot feature, tool path for the slot, tool and slot feature can be observed in 3D animation as well as 3D wire frame model. This animation and visualization provide enhanced understanding and data representation of a machining process, which makes engineers more productive, and helps them to more quickly optimize their designs. The work done in the process visualization module of the IMPlanner is not a part of this thesis work. The process visualization module is described in detail in [9].

4.7 Presenting and storing XML documents

In order to present the XML documents in the way we desire, Extensible Stylesheet Language (XSL) was written. The XSL transforms the XML file into HTML format so
that browsers can display it. The implementation of XSL is described later in section 5.2.9.

The XML document stream generated by the XML Writer is also written to a local file. This provides a clear understanding of what data is getting written to the stream. This file is written at the same time the data is written to the stream. This file will be stored on the computer where the 3I-PP server is running.
5 IMPLEMENTATION

5.1 Generating XML Document

This section talks about the implementation of XML Writer. It describes how the XML Writer would work for the simple example explained in section 4.3.2.

5.1.1 XML Body, Header and Footer

The 31-PP function call (get-instances 'machining feature) will return a list of features in the current part model as list (thole_f6_1 bhole_f7_2, lslot_f1_1, lslot_f5_2, c pocket_f7_1). Each element of this function is obtained and added in XML tags using the mapcar function. The feature-file-body function as shown in Fig. 5.1 will insert all features in the feature list inside the XML tags. The output of this function is shown in Fig. 4.4.

(defun feature-file-body (str)
  (mapcar #'(lambda (el) (format str "I-Yo-8TCFEATURE NAME="-a">" el)
    (writexml str el t nil)))
  (format str "-%-8T</FEATURE>I1 el)
  (getinstances 'machining-feature)))

Fig. 5.1 An XML Writer function to generate body of XML document

The functions for writing the prolog (header) and the footer for this XML Document in which the above feature body will be written is shown below in Fig. 5.2. The result of all the three functions together will generate a well-formed XML document as shown in Fig. 4.5.
(defun feature-file-header (str)
  (let* ((str (format str
  "<?xml version='1'?>%<xml-stylesheet type="texWxsl" href="J:\user\labimp\ProcessPlan\xmlcommFolder\features.xsl"%<LISTOFFEATURES>'%<DEFSCHEMA PART_NAME = \"-a\"%>% (get-value (first (get-instances 'machining-feature)) 'belongs-to)))))

(defun feature-file-footer (str)
  (let* ((str (format str
  "%<LISTOFFATURES>-%<IDEFSCHEMA>-%<LISTOFFEATURES>"))))

Fig. 5.2 Functions for writing header and footer of the XML document

5.1.2 Writing Element Tags

Similar to the example explained above, to get data from different objects within 31-PP, several Lisp methods are written. Fig. 5.3 shows snippet of the Lisp code written to get data from features, processes and machine & tool constraints.

(defun get-param-detail (str input val)
  (format str "%<param name="-a"%>" val)
  (format str "-%a" (get-value input val))
  (format str ">%"))

(defun expand-vector (input val)
  (format str "<param-vector x="-a" y="-a" z="-a"/>
    (aref (get-value input val) 0 ) (aref (get-value input val) 1 ) (aref (get-value input val) 2 )))

(defun get-constraint-detail (str input val)
  (if (consp (get-value input val))
    (format str "%<constraint name="-a" operator="-a" value="-a"%>" val (first (get-value input val))(rest (get-value input val)))
    (get-param-detail str input val)))

Fig. 5.3 Writing XML element tags

5.1.3 Dynamic object type headers

Methods write-element-header and element-name are written to generate dynamic XML document object type header code. Different objects that are present within the 31-PP model are manufacturing-feature, cutting, cutting-parameters, machine and tool. Fig. 5.4 and Fig. 5.5 show the Lisp code for generating dynamic the XML document header.
Method `element-name` is called within method `write-element-header` for generating dynamic object names in XML document object type headers.

```lisp
(defun write-element-header (str input inputname)
  (format str "-%t<NAME = "-a\" INSTANCEOF = "-a\">
           (element-name input) inputname
           (get-value input 'instance)))

(defun write-element-header (str (m machine) machinename)
  (format str "-%t<NAME = "-a\" INSTANCEOF = "-a\">
           (element-name m) machinename
           (get-value m 'is-a)))

(defun write-element-header (str (tl tool) toolname)
  (format str "-%t<NAME = "-a\" INSTANCEOF = "-a\">
           (element-name tl) toolname
           (get-value tl 'is-a)))
```

Fig. 5.5 Generating XML document object type header

### 5.2 Data Exchange Interface

The interface for Data Exchange mechanism is implemented through a Java applet. Fig. 5.6 shows the initial state of the Data Exchange Applet. The Data Exchange Applet provides an interface for exchanging data between the 3I-PP and the IMPlanner systems. Different components of this applet are mentioned below. This applet consists of
*JButtons*, on left side of the applet. This *JButtons*, are used for connecting to the server, loading the 3I-PP, loading part models and getting a list of features. Selection for obtaining data or process information is available through the *JRadioButton* present in the center of the Applet.

Fig. 5.6 Initial state of Data Exchange Interface
A JCheckBox option adjacent to the JRadioButton must be selected for showing XML data over the browser. Another JCheckBox on the right side of the applet is selected when complete data from the 3I-PP system is to be obtained.

The applet consists of two JTextAreas for displaying data. The JTextArea at the top right corner shows data obtained from the 3I-PP inside the JTree. The other JTextArea at the bottom of the applet is used for showing messages and notifications while the data exchange operation is performed.

5.2.1 Connecting to 3I-PP on remote server

The connection between the IMPlanner client and the 3I-PP server is established through the Connect JButton. The connection is made at a specified port number and a new socket is created every time a new request is send to the 3I-PP server. Once a connection has been made, the statusArea JTextArea will show the current status of the connection as connected. It will also show a port number, a machine name and the IP address of the machine to which it is connected. The statusArea JTextArea will append data that is generated in the console to show what is happening during the entire application session. The first stage of Data Exchange Interface when connected to the 3I-PP server is shown in Appendix A [I].

5.2.2 Loading 3I-PP

After establishing the connection between the IMPlanner client and the 3I-PP server, the 3I-PP system can be loaded through the Load 3I-PP JButton. The 3I-PP system is loaded at the machine where the server is running. After receiving the request the 3I-PP
system is loaded and is displayed on the server side. Appendix A [II] shows the 31-PP loaded in the Data Exchange environment. An acknowledgement is sent to the IMPlanner client after successful loading of the 31-PP system.

5.2.3 Loading Example (part model)

Once the 31-PP is loaded, a part model can be loaded in the 31-PP system. The name of the part model to be loaded is to be entered in exampleArea JTextArea. The default value used for this research is NetExample. This part model is loaded though the Load Example JButton. The part model is then loaded into the 31-PP system. On successful loading of the part model an acknowledgement is sent to the IMPlanner client.

5.2.4 Retrieving features for the part model

Once the part model is loaded all of its features are retrieved using Get Features JButton. The list of features generated through the 31-PP system is stored in the XML document and sent to the Java client through the TCP/IP socket stream. This data is then displayed in the featureJTree JTree of the Data Exchange Interface. Once the list of features is received in the featureJTree, the Connect, Load 31-PP, Load Example and Get Features JButton are disabled since further interaction between the user and the applet is only through featureJTree. At the same time the remaining transfer controls like Processes and Node Data JRadioButton Controls and Browser JCheckBox are enabled for further data exchange operations. Appendix A [III] shows the List of Features obtained in JTree after NetExample is loaded. The figure also shows the List of Features and the Feature Graph generated from 31-PP.
5.2.5 Getting data for features

The user is given an option to select a particular feature for the part model in 3I-PP from the JTree. After a node is selected in featureJTree, it is sent to 3I-PP server with the request for its data. The 3I-PP system stores data for each feature in the form of objects. After performing a series of operations the 3I-PP sends the complete feature object to the IMPlanner client in XML format. This feature object after receiving on Java side is stored as DOM object. This object is then added to the featureJTree and the feature data is added to the node. All the feature parameters are added as child elements and their attributes are added under the child element. Here, we don’t have to change JRadioButton option since the default value for JRadioButton is Node Data. A particular value for attribute could now be obtained by invoking a method that searches through the tree of the DOM objects in the document. When a new attribute is searched the same tree is used as long as the XML document used is the same. Similarly, data about each feature is obtained via communication between the 3I-PP server and the IMPlanner client. 3I-PP system stores data for each feature in the form of objects. Appendix A [IV] shows data expanded in JTree for feature THOLE_F6_1.

5.2.6 Get processes for features

A List of Processes is obtained for each individual feature, in a similar way as the List of Features is obtained. First we have to change the Transfer Control JRadioButton value to Processes from Node Data. Once this is done, a list of processes is obtained in similar fashion as Node Data option explained in previous section. Note here there are no child
elements in the process list. To obtain data for each process, we again change the 
\texttt{JRadioButton} option to \textit{Node Data}. Now when we left click on any process, its object is received and added to \texttt{featureJTree} as is done for features. Appendix A [V] shows List of Processes expanded for L SLOT_F5_2 feature. Also we can see the same List of Processes inside the 3I-PP.

5.2.7 \textbf{Creating feature Java object}

\textit{Right Click Event} is implemented on each node to create Java Object for that particular node. \texttt{Javax.swing.SwingUtilities.isRightMouseButton MouseEvent} is used to recognize the Right click event inside \texttt{JTree}. When a feature node is right clicked in \texttt{featureJTree}, a SAX parser is initiated to parse the incoming XML document. Thus, Right Click and Left Click events provide a switch between the SAX and DOM parsers to the user.

This parser now generates new instance of \texttt{MfgFeature} class and data are passed to a corresponding feature class. This initiates generation of a Java object for that particular feature. Similarly, Java objects for all features can be created. The created Java objects are then stored inside \texttt{featureJTree} by adding these objects to that particular node using \texttt{setUserObject()}. Once the feature Java object is created, the control is then transferred to the applet generated by the feature object. This feature object can be obtained later in the application by using \texttt{getUserObject()}. The feature object applet is called when a feature is right clicked as shown in Appendix A [VI].
5.2.8 Creating process Java object

Process objects are generated similarly to feature object as explained above. A Right Click Event is developed similarly for generating a process Java object. Since for generating processes, a feature object needs to be passed to MfgProcess class, the first feature object is generated for a particular feature. Once a feature object is created and set to that particular node in JTree, this feature object can then be retrieved from inside the SAX parser while generating the process object. The process visualization requires feature data to be passed to them to compute all 3D animation parameters. The process object is then created, which opens a corresponding process applet. The control is then passed to this applet. From this process applet as shown in Appendix A [VII], a user can view 3D animation and process visualization of the particular process.

5.2.9 Rendering XML document in browser using XSL

The XML document that is created while generating data for features/processes can also be shown inside the Web Browser. By checking Show XML in the Browser JCheckBox in Display Control and then clicking on particular feature/process node in feature JTree, the XML document is shown in the Internet Explorer Browser. The IE browser is opened using run.exec and specifying the XML document name and IE browser path. The XSL is used to display the XML document in table format in the browser.

Since XSL is an XML document itself, the document begins with <?xml version="1.0"?>. An XSL Stylesheet contains the templates that define how the elements
in an XML document should be transformed. A *match pattern* is used to associate the template with an XML element. The XSL `<xsl:for-each>` element is called to locate `<param>` and `<param-vector>` elements in XML document. The `<xsl:for-each>` element selects every XML element into the output stream of the XML transformation. The *select* attribute describes the element in the source document. It is called with *name* attribute and "." specifying value of current template. Some of the attributes require special processing for example, AXIS and PO. Since AXIS (AXIS #(-1.0 0.0 0.0)) and PO (PO #(1.0 0.0 0.0)) are lists of values representing x, y and z values, they are expanded using `<xsl:choose>`, `<xsl:when>` and `<xsl:otherwise>` for obtaining their values, as shown in Appendix B [IV].
6 TESTING AND EXAMPLES

6.1 Testing Example

The whole architecture for data exchange used for testing, deals with integrating a feature generation system (3I-PP) with a process visualization system (IMPlanner). A Lisp server is started on one computer having the 3I-PP knowledge-based system. A Java client running the IMPlanner process plan system on another computer is connected to the Lisp server using a TCP/IP socket connection. Once the connection is made, the 3I-PP system is loaded from the process plan system. A user can verify that 3I-PP system gets loaded from the IMPlanner Java client. After the 3I-PP system is loaded on the Lisp server side, a part model example is loaded.

6.1.1 NetExample Part Model

The NetExample model as shown in Fig. 2.1 is used for getting feature information. Files loaded for this example are feature file and part file, shown in Appendix D [I] and D [II] respectively. The process planning data (part and feature information) initially present inside the 3I-PP process planner are shown inside 3I-PP window in Fig. 6.1.

6.1.2 Obtaining Data

After loading the example, a list of features present in NetExample model is obtained by clicking Get Features command. A list of features in the form of strings is sent to the
client side. Once the list of features is received on the client side, it signifies the data from the 3I-PP system is obtained into the Data Exchange Applet.

A Complete feature object is not sent at this stage. The strings obtained are then added to featureJTree model as shown above in Data Exchange window in Appendix A [III]. Once a feature-list becomes available, JTree becomes the sole interface for further interaction between the Java client and the Lisp server.

![Data Exchange Window](image)

**Fig. 6.1 Screen Capture of Data shown in process frame of NetExample model**

Each feature object can be obtained after selecting it in JTree. A request for a feature object is sent to the 3I-PP, which sends the feature object containing information about its parameters, geometry and tolerance requirements back to client side. The XML document received at Java side is parsed by the DOM parser and then added to the JTree nodes.
Each node has information about the different attribute name and their corresponding values. The JTree can be opened as it increases in size. A complete hierarchy of features and processes is, thus, represented inside JTree.

After the features are obtained in JTree, a list of processes available for the particular feature can be obtained. This list of processes is then sent to the client side and added under the feature node in JTree. Similarly, the process data containing tool constraints, machine constraints and cutting parameters, can be obtained after selecting the process node in JTree. Fig. 6.2 shows data being selected for processes and feature. This action is implemented on the left click of selected feature and process in JTree.

Fig. 6.2 Process data inside JTree node
6.1.3 Verifying Data

In order to create Java objects for feature and process, each node is right clicked. This event triggers the SAX parser, which controls feature and process object generation. The example of this action is the process object shown in Appendix A [VII]. This figure shows a snapshot of a process visualization module that animates the process in 3D space. The visualization of the processes, verifies that data from the 3I-PP system is been exchanged to the IMPlanner system and is consistent with the original data.

6.1.4 Presenting Data

The interface also allows the data exchanged via the XML document to be presented to the user inside the Internet browser. Extensible Stylesheet Language (XSL) style definition for features and processes are written for displaying this data in table form in the browser. This representation is shown in Appendix A [VIII].

6.2 Different Feature File Test Cases

In the above example, we passed the feature file to the 3I-PP in which the tool and the work piece where parallel to each other. Appendix D [I] shows an example feature file for a part model. Similarly, another test case with a different feature file is used to test the data exchange mechanism. In this feature file, the work piece (stock) is set at an angle to the tool position for creating an inclined slot in the NetExample part model. The result showed the workings of visualization of the machining process of a slot for an inclined work piece. The snapshot of the process object created is shown in Appendix A [VII].
6.3 Verification

The testing with the NetExample model shows that the Data Exchange Mechanism could exchange data between the 3I-PP and the IMPlanner system. The data exchange is done in real-time immediately upon receiving a request from the IMPlanner client. The visualization of machining processes verifies that the data has been passed to the IMPlanner client and, thus, verifies the developed data exchange mechanism. The Integration allows data from the 3I-PP to be completely visible to modules within the IMPlanner, just as if they were local routines.

At all stages in data exchange, the 3I-PP generates only the data that is requested. This proves our claim for granular exchange of data, which means generating and sending only the required information from the server. The mechanism is verified to allow data mobility between systems on remote computers. Different modules were loaded on different machines to test its validity. Thus, this interface connects different modules: manufacturing process selection of the 3I-PP system and process visualization of the IMPlanner system allowing real-time data exchange between them. The integration between two disparate applications, here the 3I-PP and the IMPlanner system is, thus, tested and verified.
7 CONCLUSION AND FUTURE WORK

7.1 Research Summary

This research has developed an XML based data exchange mechanism for integrating disparate systems to allow the systems to communicate and allow data interchange to take place. It allows data exchange between systems over the network without making changes in existing systems. The architecture presented facilitates integration between the process selection module written in Lisp and the process visualization module written in Java to exchange data at run-time and, thus, allow a real time data integration environment.

7.2 Research Contributions and Limitations

This work provides a new horizon where product designers and developers, could come together to work collaboratively at a virtual space on product development cycle; thus increasing overall productivity. Similar integrations would provide advantages and capabilities for other disparate modules, thus increasing scalability of newly developed applications. Customized solutions for solving interoperability issues between disparate systems could be similarly implemented.

The contributions provided by this research are mentioned below:

- An XML Writer is developed which captures required data dynamically and generates appropriate XML document.
• Real-time communication is achieved between Lisp and Java applications. In addition, interaction between them is also possible on remote computers.

• Data integration between Knowledge-based system and Process Visualization was successfully implemented.

The limitations of this research are explained below:

• The approach does not present two-way communication using XML. Only Lisp to Java data transfer is done using XML. The requests sent from Java to Lisp applications could be sent in XML format instead of list. Also, this type of communication may be required if feature objects are generated in Java environment. These objects will need to be sent to 3I-PP again for performing process selection. Requirements for implementing this would be to write an XML Writer on the Java side and an XML Parser for Lisp.

• The XML Writer generates an XML file on the basis of object type of attribute passed, which makes it generic for objects defined inside the 3I-PP system. Another approach could be to have an XML Writer generate XML document on the basis of Lisp attributes directly, instead of getting them from its object type. Each of the attributes and variables would be defined inside the XML Writer. With this approach, the XML Writer will be independent of underlying applications it is working on.
7.3 Future Extensions

Future work in this area would be to extend this architecture to allow additional families of process design and planning systems to integrate and exchange data between themselves. The integration methodology used in this research, could be used for integration of different heterogeneous system. This would solve current interoperability issues in different CAD/CAM / Process Planning models.

The need for such integration mechanism extends to where the collaboration of various different industry modules is required. Different industry modules like tolerance modeling, cost estimation, etc could be similarly collaborated.
REFERENCES


[22] The Franz Inc. (February 2001). CORBA-Enabling Allegro CL Applications with Allegro ORBLink, Available from the Internet, URL:
http://www.franz.com/resources/corba/white_papers/corba_white_paper.php3

[23] The Franz Inc. (February 2001). Jlinker – A Dynamic Link between Lisp and Java, Available from the Internet, URL:
http://www.franz.com/support/documentation/6.0/doc/jlinker.htm#intro-1

APPENDIX

[A] Screen shots of different stages of data exchange mechanism

I. Connecting to 3I-PP server

II. Loading 3I-PP Knowledge Based System
III. Loading NetExample Part Model and Obtaining List of Features

![Image of part model with features]

IV. Generating Feature Data for THOLE_F6_1 feature

![Image of feature data generation]

- Feature List:
  - THOLE_F6_1
  - BSA_LF_F2_1
  - LBS_LF_F1_2
  - LSP_LF_F1_3
  - SLO_Pocket_F3_2
  - SLAB_F2_1
  - CPOCKET_F4_1

- Feature Data:
  - RADIUS: 0.125
  - MIS: ...
  - P0_14X: ...
  - RAD-LST: ...
  - TRUE POSITION: ...
  - DEPTH: ...

- Feature Parameters:
  - RADUS: 0.125
  - COPY: ...
  - EXTEND: ...
  - EXTENDED-CYL-NO: ...
  - TRUE-POSITION: ...
  - DEPTH: ...
  - PARALLELM and its value is 1.125
V. Generating List of Processes for L SLOT_F5_2 feature

VI. Creating Feature Objects
VII. Showing Process Animation

VIII. Displaying XML on the browser
[B] Samples of XML documents generated

I. XML document generated for a feature

```xml
<?xml version='1.0'?>
<?xml-stylesheet type="text/xsl" href="c:\Interface\xmlcommFolder\features.xsl"?>
<ListOfFeatures>
    <Feature NAME = "SLAB_F2_1" INSTANCEOF = "SLAB">
        <param name="PROF-RADIUS">NIL</param>
        <param name="MAY-BE-MACHINED-BY">END-MILLING-PERIPHERAL</param>
        <param name="FLOOR">NIL</param>
        <param name="PROFILE">NIL</param>
        <param name="ON-LST">NIL</param>
        <param name="ORIGINE">NIL</param>
        <param name="AXIS"><param-vector x="0.0" y="1.0" z="0.0"/></param>
        <param name="LENGTH">1.25</param>
        <param name="WIDTH">1.25</param>
        <param name="HEIGHT">0.125</param>
        <param name="POSITIVE-TOL">NIL</param>
        <param name="NEGATIVE-TOL">NIL</param>
        <param name="PERPENDICULARITY">NIL</param>
        <param name="EU">NIL</param>
        <param name="EV">NIL</param>
        <param name="EW">NIL</param>
        <param name="PO"><param-vector x="1.0" y="0.0" z="0.0"/></param>
        <param name="COMP-OF">NIL</param>
        <param name="STATUS">NIL</param>
        <param name="ACCESS">(1.375 , #(0.0 1.0 0.0))</param>
        <param name="SURF-FEAT">NIL</param>
        <param name="CSG">SCHEMA->CSG</param>
        <param name="DISP">DISP-FEATURE</param>
        <param name="SOLID-NAME">NIL</param>
        <param name="BELONGS-TO">NETEXAMPLE</param>
        <param name="TOLERANCE">NIL</param>
        <param name="DO-AFTER">PPR-TREE-START</param>
        <param name="DO-BEFORE">THOLE_F6_l</param>
        <param name="SURFQUALITY">NIL</param>
        <param name="MIN-COST">NIL</param>
        <param name="PLANNING-STATUS">NIL</param>
    </Feature>
</ListOfFeatures>
```
II. XML document generated for a process

<?xml version='1.0'?><!--xml-stylesheet type="text/xsl" href="J:\user\labimp\ProcessPlan\xmlcommFolder\processes.xsl"-->
<ListOfProcesses>
  <Feature Name="LSLOT_F5_2">
    <Process Name="P37-LSLOT_F5_2-SIDE--CNC-H-SLOTT"/></Process>
    <Process Name="P35-LSLOT_F5_2-SIDE--CNC-V-SLOTT"/></Process>
    <Process Name="P33-LSLOT_F5_2-SIDE--PLAIN-SLOTT"/></Process>
    <Process Name="P31-LSLOT_F5_2-SIDE--UNIVE-SLOTT"/></Process>
    <Process Name="P42-LSLOT_F5_2-END-M-CNC-H-END-M"/></Process>
    <Process Name="P41-LSLOT_F5_2-END-M-CNC-V-END-M"/></Process>
    <Process Name="P40-LSLOT_F5_2-END-M-VERTI-END-M"/></Process>
    <Process Name="P39-LSLOT_F5_2-END-M-UNIVE-END-M"/></Process>
  </Feature>
</ListOfProcesses>
III. XML document generated for a feature including all processes

```xml
<?xml version='1.0'?>
<features>
  <feature name="LSLOT_F5_2" instanceof="LIN-OPEN-SLOT">
    <param name="MAY-BE-MACHINED-BY">SIDE-MILLING</param>
    <param name="F-NORM"><param-vector x="0.0" y="1.0" z="0.0"/></param>
    <param name="CTR-PL">NIL</param>
    <param name="WIDTH">0.25</param>
    <param name="LENGTH">1.25</param>
    <param name="FLR-POS">NIL</param>
    <param name="FLOOR">NIL</param>
    <param name="WALLS">NIL</param>
    <param name="CEILING">NIL</param>
    <param name="ENDS">NIL</param>
    <param name="HEIGHT">0.25</param>
    <param name="CONCAT">NIL</param>
    <param name="CFS">NIL</param>
    <param name="UP-CLASS">NIL</param>
    <param name="STEP-OF-CUT">ROUGH-MILLING</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="PERPENDICULARITY">NIL</param>
    <param name="EU">NIL</param>
    <param name="EV"><param-vector x="-1.0" y="0.0" z="0.0"/></param>
    <param name="EW">NIL</param>
    <param name="PO"><param-vector x="1.0" y="0.0" z="0.0"/></param>
    <param name="COMP-OF">NIL</param>
    <param name="STATUS">NIL</param>
    <param name="ACCESS">(0.0 , #(-1.0 0.0 0.0))</param>
    <param name="SURF-FEAT">NIL</param>
    <param name="CSG">LIN-SLOT->CSG</param>
    <param name="DISP">DISP-FEATURE</param>
    <param name="SOLID-NAME">NIL</param>
    <param name="BELONGS-TO">NETEXAMPLE</param>
    <param name="TOLERANCE">NIL</param>
    <param name="DO-AFTER">LSLOT_F1_1</param>
    <param name="DO-BEFORE">NIL</param>
    <param name="SURFQUALITY">NIL</param>
    <param name="MIN-COST">NIL</param>
    <param name="PLANNING-STATUS">NIL</param>
  </feature>
  <process name="P42-LSLOT_F5_2-END-M-CNC-H-END-M" instanceof="END-MILLING-SLOTTING">
    ...
  </process>
</features>
```
<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="SWEPPDIRECTION">NIL</param>
<param name="SWEPTSOLID">MAKE-ENDMILLING-SOLID</param>
<param name="DISP-PROCESS">DISP-CUTTING</param>
<param name="DISP-PROC-FEATURES">DISP-CUTTING-FEATURES</param>
<param name="PLANNING-STATUS">SELECT</param>
<param name="PREFERRED-TO">NIL</param>
<param name="IS-MACHINED-BY">LSLOT_F5_2</param>
<param name="MAY-BE-MACHINED-BY">NIL</param>
<param name="PREFERRED-TO+INV">NIL</param>
<param name="MEMBER-OF-PROCESS">NIL</param>
<param name="TOOL-AXIS">X-NEGATIVE</param>
<param name="PROCESS-TIME">0.155832968928065</param>
<param name="PROCESS-COST">0.22383280991485716</param>
<param name="SETUP-TIME">NIL</param>
<param name="MAY-USE-MACH">UNIVERSAL-MILL</param>
<param name="MAY-USE-TOOL">END-MILLING-TOOL</param>

<MACHINE-CONSTRAINT NAME = "M36-LSLOT-CNC-H" INSTANCEOF = "CNC-HORIZONTAL-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">120</param>
  <constraint name="POWER" operator="=" value="3.0"/>
  <param name="MAT-HANDLING-TIME">0.15</param>
  <param name="SPEED-EFFICIENCY">NIL</param>
  <param name="TOOL-CHANGE-TIME">0.15</param>
  <param name="UNIT-COST">1.3</param>
  <param name="MAY-USE-MACH+INV">NIL</param>
  <param name="SHOULD-USE-MACH+INV">P42-LSLOT_F5_2-END-M-CNC-H-END-M</param>
</MACHINE-CONSTRAINT>

<TOOL-CONSTRAINT NAME = "T36-LSLOT-END-M" INSTANCEOF = "END-MILLING-TOOL">
  <constraint name="FLUTE-LENGTH" operator="" value="0.25"/>
  <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="0.25"/>
  <param name="LIFE-CYCLE">220</param>
  <param name="COST">30</param>
</TOOL-CONSTRAINT>
<param name="MAY-USE-TOOL+INV">NIL</param>
<param name="SHOULD-USE-TOOL+INV">P42-LSLOT_F5_2-END-M-CNC-H-END-M</param>
</TOOL-CONSTRAIN>
</CUT-PARAMETERS>
</PROCESS>
</param-child>
<param-child name="P31-LSLOT_F5_2-SIDE--UNIVE-SLOTT">
<PROCESS NAME="P31-LSLOT_F5_2-SIDE--UNIVE-SLOTT" INSTANCEOF="SIDE-MILLING">
<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="SWEEPDIRECTION">NIL</param>
<param name="SWEPTSOLID">MAKE-SIDEMILLING-SOLID</param>
<param name="DISP-PROCESS">DISP-CUTTING</param>
<param name="DISP-PROC-FEATURES">DISP-CUTTING-FEATURES</param>
<param name="PLANNING-STATUS">SELECT</param>
<param name="IS-MACHINED-BY+INV">LSLOT_F5_2</param>
<param name="MAY-BE-MACHINED-BY+INV">NIL</param>
<param name="PREFERRED-TO+INV">NIL</param>
<param name="MEMBER-OF-PROCESS">NIL</param>
<param name="TOOL-AXIS">Y-POSITIVE</param>
<param name="PROCESS-TIME">0.06592766472923337</param>
<param name="PROCESS-COST">0.08646666797180222</param>
<param name="SETUP-TIME">NIL</param>
<param name="MAY-USE-MACH">UNIVERSAL-MILL</param>
<br>
</MACHINE-CONSTRAINT NAME="M25-LSLOT-UNIVE" INSTANCEOF="UNIVERSAL-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">75</param>
<constraint name="POWER" operator="\"=>\" value="3.0"/>
<param name="MAT-HANDLING-TIME">0.4</param>
<param name="SPEED-EFFICIENCY">NIL</param>
<param name="TOOL-CHANGE-TIME">0.25</param>
<param name="UNIT-COST">1.1</param>
<param name="MAY-USE-MACH+INV">NIL</param>
<param name="SHOULD-USE-MACH+INV">P31-LSLOT_F5_2-SIDE--UNIVE-SLOTT</param>
</MACHINE-CONSTRAINT>
<TOOL-CONSTRAINT NAME = "T25-LSLOT-SIDE-" INSTANCEOF = "SLOTTING-TOOL">
  <constraint name="WIDTH" operator="=" value="0.25"/>
  <constraint name="NUMBER-OF-TEETH" operator="=" value="6"/>
  <constraint name="MATERIAL" operator="EQL" value="HSS"/>
  <constraint name="DIAMETER" operator="=" value="0.75"/>
  <param name="LIFE-CYCLE">260</param>
  <param name="COST">55</param>
  <param name="MAY-USE-TOOL+INV">NIL</param>
  <param name="SHOULD-USE-TOOL+INV">P31-LSLOT_F5_2-SIDE--UNIVE-SLOTT</param>
</TOOL-CONSTRAINT>
<CUT-PARAMETERS NAME = "CUT-DATA19" INSTANCEOF = "CP-SIDE-MILLING">
  <param name="FOR-PROCESS">P37-LSLOT_F5_2-SIDE--CNC-H-SLOTT</param>
  <param name="TOOL-DIAMETER">NIL</param>
  <param name="SPEED">120</param>
  <param name="FEED">0.008</param>
  <param name="PART-MATERIAL">CARBON-STEEL</param>
  <param name="TOOL-MATERIAL">HSS</param>
</CUT-PARAMETERS>
</PROCESS>
</param-child>
</FEATURE>
</LISTOFFEATURES>
IV. XSL file written for showing XML data in browser

```xml
<?xml version='1.0'?>
<!-- features.xsl
    Description: This program displays features in formatted and structured
    form using XML and XSL -->
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/TR/WD-xsl"
    xmlns="http://www.w3.org/TR/REC-html40"
    result-ns="">
<!-- default rule -->
<xsl:template><xsl:apply-templates/></xsl:template>
<!-- FEATURES -->
<xsl:template match="LISTOFFEATURES"><xsl:apply-templates/></xsl:template>
<xsl:template match="FEATURE">
    <xsl:for-each select="param">
        <xsl:if test='@name[.='BELONGS-TO']'>
            <h3>Model is <xsl:value-of select="."/></h3>
        </xsl:if>
    </xsl:for-each>
    <table border="1">
        <tr bgcolor="#308030" align="center">
            <td>NAME</td>
            <td>VALUE</td>
        </tr>
        <h3>Feature is <xsl:value-of select="@NAME"/></h3>
        <xsl:for-each select="param">
            <xsl:choose>
                <xsl:when test="@name='AXIS' or @name='PO'">
                    <xsl:for-each select="param-vector">
                        <xsl:value-of select="@x">,</xsl:value-of select="@y">,</xsl:value-of select="@z"/>
                    </xsl:for-each>
                </xsl:when>
                <xsl:otherwise><xsl:value-of select="."/></xsl:otherwise>
            </xsl:choose>
        </xsl:for-each>
    </table>
</xsl:template>
</xsl:stylesheet>
```
[C] Samples of code for implementation of Data Exchanger

I. Sample code for a Socket Connection on the Lisp and Java Side

```
(setf *pipe* (comm:start-up-server :function 'make-stream-and-talk
 :service 1246))

(defun make-stream-and-talk (handle)
  (let* ((stream (make-instance 'cornrn:socket-stream
                                 :socket handle
                                 :direction :io
                                 :element-type 'base-char))
         (mp:process-run-function (format nil "talk-D" handle)
                                   ()
                                   'talk-on-stream stream

  ))

private static final int PORTNUM = 1246;
public static Socket socket = null;
InputStreamReader isr = null; // converting stream to string by reading from stream
BufferedReader inBuffer = null;

socket = new Socket (serverField.getText(), PORTNUM);
out = new PrintWriter (socket.getOutputStream(), false);

public InputStream getStreamAndSocket() throws IOException{
  socket = new Socket (serverField.getText(), PORTNUM);
inStrNew = socket.getInputStream();
return inStrNew;
}
```
II. Snippet of Lisp code for generating XML document

(defun write-element-header (str input inputname)
  (format str "%-10T<-a NAME = \"-a\" INSTANCEOF = \"-a\">
    (element-name input) inputname
    (get-value input 'instance)))

(defun write-file-footer (str input)
  (let* ((str
    (format str "%-2T</LISTOFFEATURES>"))
  )))

(defun expand-element-values (str input val)
  (format str "%-20T<param name="-a">" val)
  (expand-vector str input val)
  (format str "</param>"))

(defun expand-object ((f manufacturing-feature) str targetsymbol exp-slots)
  (mapcar #'(lambda (el)
    (format str "%-6T<param-child name="-a">" el)
    (writexml str el t t)
    (format str "%-6T</param-child>"))
  (get-values f exp-slots))
)

(defun get-constraint-detail (str input val)
  (if (consp (get-value input val))
    (format str "%-20T<constraint name="-a" operator="-a" value="-a"/>" val (first (get-value input val))(rest (get-value input val)))
    (get-param-detail str input val) )
)

(defun get-param-detail (str input val)
III. Snippet of XML DOM parser code

```java
import com.sun.xml.tree.*; //Implements and extends DOM
import org.w3c.dom.*; //Xml Document

public static void createXmlDoc(InputStream inStr, DefaultMutableTreeNode node, String dataType) {
    fileDocMap.put(fullFileName,XmlDocument.createXmlDocument(inStr,false));

    /*
     * Returns a Tag Value given node and file name.
     *
     * @param keyfeature node name
     * @param nodeKey XML file name
     * @return an XML Document
     */
    public static String getTagValue(String keyfeature,String nodeKey) {
        Element row;
        String fileName;
        fileName = nodeKey;
        getFileMap(fileName + "data");
        size = XmlUtils.getSize(xDoc, ROOT_ELEMENT_TAG);
        String sxml = "";
       DataExchangeApplet.statusArea.append("size here is "+size);
        for (int i = 0; i < size; i++) {
            row = XmlUtils.getElementById(xDoc, ROOT_ELEMENT_TAG, i);
            sxml = row.getFirstChild().toString();
            String sname = row.getAttribute("name");
            if(sname.equalsIgnoreCase(keyfeature)){
                return sxml;
            }
        }
        return null;
    }
```
IV. Snippet of XML SAX parser code

```java
import org.xml.sax.*;
import org.xml.sax.helpers.ParserFactory;
import com.sun.xml.parser.Resolver;

public void parseFeature(InputStream inStr) {
    try {
        InputStreamReader isr = new InputStreamReader(inStr);
        InputSource is = new InputSource(isr);
        dea.statusArea.append("parsing sax xml file");
        handler = new SaxFeatureHandler();
        parser = getParser(parserClassName);
        parser.setDocumentHandler(handler);
        parser.setErrorHandler(handler);
        parser.parse(is);
        dea.statusArea.append("xml file parsed");
    } catch (Throwable t) {
        System.out.println(t);
        t.printStackTrace();
    }
}

public void startElement(String name, AttributeList atts) {
    debugDisplay("<start elem:"+name+">");
    if (name.equalsIgnoreCase("FEATURE")) {
        // check feature instance (Attribute of given element and create appropriate feature object
        instanceElement = atts.getValue("INSTANCEOF");

        if (instanceElement.endsWith("HOLE")) {
            f = new Hole();
            f.setFeatureName("enterfromxml");
        }

        if (instanceElement.endsWith("POCKET")) {
            f = new Pocket();
        }
    }
    if (name.equalsIgnoreCase("PARAM")) {
        currentElement = atts.getValue("name");
        this setCurrentElement(currentElement);
    }
```
/**
 * Return Sax Feature Directory Handler given Current Vector3d
 *
 * The Current Vector3d passed through parameter will be used to set
 * to the corresponding feature object in SaxFeatureHandler
 *
 * @param v Current Vector3d to be set
 * @return SaxFeatureHandler with Vector added to it.
 */

public SaxFeatureHandler setCurrentVector(Vector3d v) {
    currentVector = new Vector3d(v);
    return this;
}

public void endElement(String name) {
    debugDisplay("</" + name + ">");
    if (name.equalsIgnoreCase("FEATURE")) {
        debugDisplay(f.toString());
        f.setApplet(new GUIApplet(f));
        f.display("Feature Object");
        f.setFeatureName(dea.node.getUserObject().toString());
        f.setVector(this);
        f = null;
    } else {
        if(f != null){
            f.setVector(this);
        }
    }
}

public void characters(char ch[], int start, int length) {
    value = new String(ch, start, length);
    if(!value.trim().equals("") ) {
        if(f != null ){
            this.setValue(value);
            f.setAttribute(this);
        }
    }
}
[D] Input data files for 31-PP system

I. Snippet of feature file for a part model loaded in 31-PP

;;; -* Mode:Lisp; Syntax:Common-Lisp; Package:crl-user; Base:10; Patch-file: YES. -*
; (in-package 'crl-user)

(if (not (contextp 'HINTS)) (create-context 'HINTS '$ROOT-CONTEXT))
(setq *USER-CONTEXT* (get-context))
(assert-context 'HINTS)

; --- Trial for overall model ---
; ----------------------------------------
(DEFSchema lslot_F1_1
 :PARALLEL
 :NOTIFY
 (INSTANCE LIN-OPEN-SLOT)
 (DO-AFTER PPR-TREE-START)
 (DO-BEFORE lslot_F5_2)
 (AXIS #(-1.0 0.0 0.0))
 (po #(1.0 0.0 0.0))
 (ev #(-1.0 0.0 0.0))
 (f-norm #(0.0 1.0 0.0))
 (ACCESS
  (0.0 0.0
   #(-1.0 0.0 0.0)))
 (LENGTH 1.25)
 (WIDTH 0.5)
 (BOTTOM-DIST 0.0)
 (HEIGHT 0.0)
 (BELONGS-TO NETEXAMPLE)
)

(DEFSchema slab_F2_1
 :PARALLEL
 :NOTIFY
 (INSTANCE SLAB)
 (DO-AFTER PPR-TREE-START)
 (DO-BEFORE thole_F6_1)
 (po #(1.0 0.0 0.0))
 (LENGTH 1.25)
 (WIDTH 1.25)
 (HEIGHT 0.125)
 (BELONGS-TO NETEXAMPLE)
 (STEP-OF-CUT ROUGH-MILLING)
 (AXIS #(0.0 1.0 0.0))
 (ACCESS
  (1.375 0.0
   #0.0 1.0 0.0)))
(DEFSHEMA cpocket_F3_1
  :PARALLEL
  :NOTIFY
  (INSTANCE REG-OPEN-POCKET)
  (DO-AFTER PPR-TREE-START)
  (DO-BEFORE thole_F6_1)
  (AXIS #(0.0 -1.0 0.0))
  (po #(1.0 0.0 0.0))
  (ACCESS
   (0.0, #(0.0 -1.0 0.0))
  (BELONGS-TO NETEXAMPLE)
  (PROF-RADIUS 0.125)
  (STEP-OF-CUT ROUGH-MILLING)
  (LENGTH 2.875)
  (WIDTH 1.25)
  (HEIGHT 0.125)
)

(DEFSHEMA cpocket_F4_1
  :PARALLEL
  :NOTIFY
  (INSTANCE REG-CLOSED-POCKET)
  (DO-AFTER PPR-TREE-START)
  (DO-BEFORE bhole_F7_1)
  (AXIS #(1.0 0.0 0.0))
  (po #(1.0 0.0 0.0))
  (ACCESS
   (1.25, #(1.0 0.0 0.0))
  (BELONGS-TO NETEXAMPLE)
  (PROF-RADIUS 0.125)
  (STEP-OF-CUT ROUGH-MILLING)
  (LENGTH 1.75)
  (WIDTH 1.25)
  (HEIGHT 0.125)
)

(DEFSHEMA lslot_F5_2
  :PARALLEL
  :NOTIFY
  (INSTANCE LIN-OPEN-SLOT)
  (DO-AFTER lslot_F1_1)
  (AXIS #(-1.0 0.0 0.0))
  (po #(1.0 0.0 0.0))
  (ev #(-1.0 0.0 0.0))
  (f-norm #(0.0 1.0 0.0))
  (ACCESS

(0.0 . #(-1.0 0.0 0.0)))
(LENGTH 1.25)
(WIDTH 1.0)
(HEIGHT 0.0)
(BELONGS-TO NETEXAMPLE)
)

; --------------------------------------------------------------------------

(DEFSHEMA thole_F6_1
 :PARALLEL
 :NOTIFY
 (INSTANCE CYL-THROUGH-HOLE)
 (DO-AFTER slab_F2_1
     opocket_F3_1)
 (MAY-BE-MACHINED-BY CORE-MAKING)
 (AXIS #(0.0 1.0 0.0))
 (po #(1.0 0.0 0.0))
 (ACCESS
     (0.0 . #(0.0 -1.0 0.0))
     (1.375 . #(0.0 1.0 0.0)))
 (RADIUS 0.125)
 (ON-LST
     (0.0 . OUT->IN)
     (1.0 . IN->ON)
     (1.125 . ON->OUT))
 (BELONGS-TO NETEXAMPLE)
 (DEPTH 1.125)
)

; --------------------------------------------------------------------------

(DEFSHEMA bhole_F7_1
 :PARALLEL
 :NOTIFY
 (INSTANCE CYL-BLIND-HOLE)
 (DO-AFTER cpocket_F4_1)
 (MAY-BE-MACHINED-BY CORE-MAKING)
 (AXIS #(1.0 0.0 0.0))
 (po #(1.0 0.0 0.0))
 (ACCESS
     (1.25 . #(1.0 0.0 0.0)))
 (RADIUS 0.125)
 (ON-LST
     (1.25 . OUT->IN)
     (1.125 . IN->ON)
     (1.125 . ON->OUT))
 )
 (BELONGS-TO NETEXAMPLE)
 (DEPTH 0.5)
)

; ---------------------End of file---------------------------------------------

II. Snippet of part file for a part model loaded in 3I-PP

;;; -*- Mode:Lisp; Syntax:Common-Lisp; Package:crl-user; Base:10; Patch-file: YES. -*-
;(in-package 'crl-user)

(if (not (contextp 'HINTS)) (create-context 'HINTS '$ROOT-CONTEXT))

(setq *USER-CONTEXT* (get-context))
(assert-context 'HINTS)

(DEFSHEMA NETEXAMPLE
  :PARALLEL

    (INSTANCE PART)
    (MATERIAL CARBON-STEEL)
    (PART-DIM-X 10)
    (PART-DIM-Y 11)
    (PART-DIM-Z 10)
    (batch-size 50)

  )

(assert-context *USER-CONTEXT*)