Object-oriented File Translator for the MEMCAD System

by

King Chung Yu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1995

Certified by ....
Stephen D. Senturia
Barton L. Weller Professor of Electrical Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Object-oriented File Translator for the MEMCAD System

by

King Chung Yu

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The MEMCAD system is a system of CAD/CAE tools which are configured to support the design, simulation, and analysis of microelectromechanical systems (MEMS). In version 1.0 of the system, PATRAN is used as the chief modeling tool, whereas in version 2.0, I-DEAS is used. In this thesis, I designed and implemented a file translator which translates modeling data from a format supported by I-DEAS to a format supported by PATRAN. This enables MEMBase, the core of the MEMCAD system which in version 1.0 reads data from PATRAN, to read data seamlessly from I-DEAS. The translator was written entirely in C++, with extensive use of stream manipulations. It was designed in such a way that subsequent extension of the translator would be straightforward.

Thesis Supervisor: Stephen D. Senturia
Title: Barton L. Weller Professor of Electrical Engineering

Acknowledgments

May I extend my gratitude towards Steve, for his supportive and enthusiastic supervision; John, for his help in my C++ and in the integration of UNV2PNF with MemBase; Peter, for teaching me I-DEAS, solid modeling and meshing. I would also like to acknowledge ARPA for its financial support (contract J-FBI-92-196).

I am also in debt to the following people, whose company has immensely enriched my life, both at MIT and *beyond*.

To Dad and Mom, for their love and support throughout these twenty-four years.

To Katherine, for being a special partner and friend.

To Sunman, Thomas and Felix, for being friends and comrades on the table. Those precious and glorious moments together are unforgettable.

To Jocelyn, for sharing our many walks together, and being a passionate encourager and supporter throughout.

To Bernard, for being a special roommate, friend and coworker. "Ba Bit Tab", "Jui Yik", "Log Ha" etc. are songs I will never forget!

To Albert, Jerome, Leo, Pchan and Yuk, for being friends and project partners in various memorable classes together.

To Brian, two David's, Jimmy and Kin, for walking together to find and grasp the right priorities in life.

And to countless others: Ben Suk, Ernie, Gideon, Kai, Zachary, etc.

Last but not least, to the Lord Jesus Christ, who has made all of the above a wonderful reality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is part of the program undertaken by the MEMCAD group under Professor Stephen D. Senturia in the Microsystems Technology Laboratories of MIT, which is referred to as "Application-Driven CAD for Microdynamical Systems" [1]. This introductory chapter describes the program, what motivates the development of the Object-oriented File Translator, and the organization of this thesis.

1.1 Application-Driven CAD for Microdynamical Systems

The primary goal of the program is the development of a system of CAD/CAE tools (MEMCAD) which are configured to support the design, simulation, and analysis of Microelectromechanical Systems (MEMS) [2].

For the "mechanical" part of electromechanical simulation, MEMCAD uses, whenever possible, commercially available finite-element modeling (FEM) and CAD packages. The focus is the creation of the 3-D model from mask and process specifications (incorporating material properties, loads, and boundary conditions) and the development of new simulation capabilities not routinely available elsewhere. These are to be assembled into an overall system which is sophisticated in its capabilities, but still usable by MEMS designers who might not be experts in numerical methods.

1.1.1 Program Organization

The overall program is organized into four major tasks, as explained below:

1. **MEMCAD System**

   Development of the MEMCAD system, including provision for coupling to existing technology CAD and mechanical CAD tools, and for accessibility to users.

2. **Material Property Test Structures**

   Design and analysis of material property test structures, and demonstration of their use to determine material properties in practical MEMS process sequences.

3. **Simulation and Analysis**

   Development of new simulation tools capable of analyzing large mechanical deformations in the presence of electroquasistatic and fluid-induced forces, with emphasis on boundary-element methods and accelerated algorithms to reduce computational costs.

4. **Demonstration System**

   Design, fabrication, and testing of an experimental fluid-metering microdynamical system, including a high-speed microvalve with integrated actuation, a flow sensor, and an electronic feedback control system.

1.1.2 MEMCAD System

This thesis is part of the MEMCAD System task. The MEMCAD system development is an evolutionary process. Each successive version will incorporate new capabilities and/or user enhancements. Each version is installed at MIT and tested first by the MIT MEMS community (the alpha-test), and then transferred to an off-site beta-test prior to release.

MEMCAD 2.0 is the second version of MEMCAD. In MEMCAD 2.0, I-DEAS Master Series 1 [3] is selected as the core commercial package, replacing both Pro/Engineer [4] and PATRAN [5] in MEMCAD 1.0. ABAQUS [6] and FASTCAP [7], continue to be used in MEMCAD 2.0, for structural analysis and mechanical simulation, and for electrostatic simulation, respectively. See Figure 1-1 for a block diagram showing the structure of MEMCAD 2.0.

One of the tasks which remains in order to complete a beta-test version of MEMCAD 2.0 is the creation of a translator within MemBase which can read and write I-DEAS universal files. That translation is the subject of this thesis.

1.2 Purpose of the File Translator

1.2.1 Data Incompatibility

As with the development of many software systems, the development of MEMCAD 2.0 involves the usage of existing (commercial or non-commercial) software packages. Incompatibility of data file formats between different packages used at the same time or between different packages used in different versions of the system often creates a data incompatibility problem. In the case of MEMCAD 2.0, the switching from using PATRAN as the main solid modeling tool in version 1.0 to using I-DEAS in version 2.0 creates this problem.

1.2.2 Purpose

As described in Section 1.1.2, there is a need within MEMCAD for a translator within MemBase which can read and write I-DEAS universal files. Previously, in version 1.0, MemBase read in files from PATRAN, which were in the PATRAN neutral file format. With the switching to using I-DEAS in version 2.0, there comes the need for MemBase to be able to read in files from I-DEAS, which are in the universal file format. The task of the file translator is thus to translate output universal files from I-DEAS into PATRAN neutral files, so that the old MemBase can read in the same information. Hence the file translator is called UNV2PNF ("UNV" standing for universal files and "PNF" standing for PATRAN neutral files, and the "2" in the middle being a common homonym for "to"), and will be referred to as such in the remaining part of

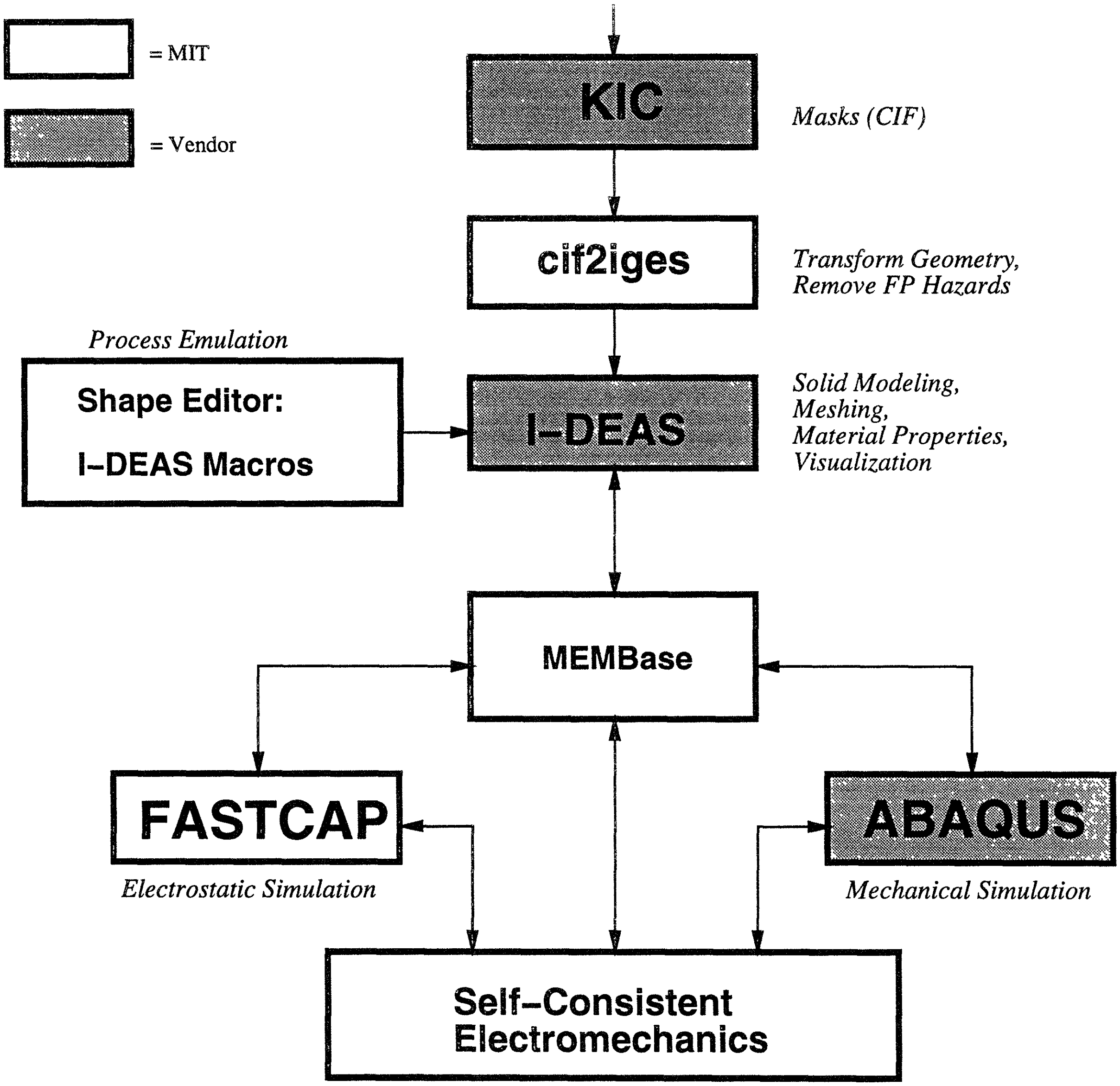

Figure 1-1: Structure of the MEMCAD 2.0 system.

this thesis.

1.3 Organization of Thesis

Chapters 2 and 3 describe the structures of I-DEAS universal files and PATRAN neutral files respectively. Chapter 4 is a specification of the translation problem at hand. It specifies the problem in terms of the types of physical models that are being dealt with, and the syntax and semantics of typical universal files and PATRAN neutral files being translated. Chapter 5 discusses the design and implementation issues of UNV2PNF, including a discussion of the extensibility of the translator. Finally, Chapter 6 presents some summary information about the implementation, and some suggestions of future work.

Chapter 2

Universal Files

The universal file is one of the file formats to which I-DEAS can export its model data. Most of the information concerning universal files in this chapter are taken from Smart View, the online documentation of I-DEAS [8].

Universal files are plain ASCII files which can be used to store selected information from a model file, which is a database where I-DEAS stores the user's work when he/she gives the SAVE command. They are typically used to interface with user-written programs or to transfer information between different computer platforms. They are designed so that they may be easily read and written using user-written programs. The universal file link is bi-directional; that is, I-DEAS can export universal files which are then read by user-written programs, and I-DEAS can also import universal files written by user-written programs.

Here are some reasons for using universal files:

- For small models, a universal file is smaller than a model file.
- The user (who has access to the universal files) has more control over what is written to the file.
- In case of loss of data in the model file, perhaps due to failure in hardware storage device, a universal file can serve as a backup to the model file.

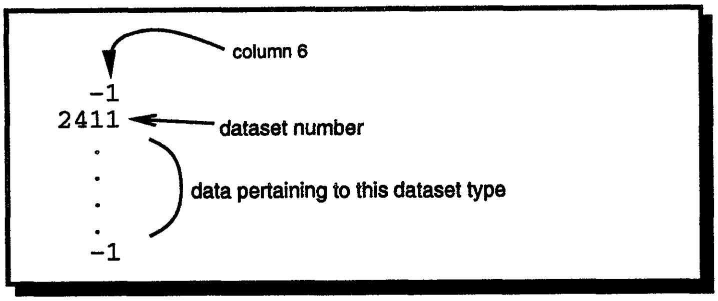

Figure 2-1: **The contents of a dataset.**

2.1 Structure of Universal Files

The basic building block of a universal file is a block of information called a dataset. A universal file is a simple concatenation of a number of datasets. In a typical universal file that is processed by the file translator of this thesis, there are about 50 datasets.

Each dataset begins and ends with a delimiter line, which is a line containing a minus sign in column 5 and a "1" in column 6. The remainder of the line is blank. There are about a thousand different types of datasets in the universal file definition, each with its own purpose, content and format. For a partial list of dataset types, see Table 2.1, which lists the numbers and names of some datasets in the Simulation application of I-DEAS. For a complete list, refer to Smart View [8].

2.1.1 Contents of datasets

Following the beginning delimiter line is the data type record, which is a line containing a number which is the dataset number. For example, finite element nodes with double precision storage are written with dataset number 2411.

Following the data type record, the body of the dataset contains data which is dependent on the dataset type. See Figure 2-1. Each dataset type has its own specification of the format of the body. It typically divides the body into pieces called records, each record ending with a newline character.

Between datasets (that is, after the ending delimiter of a certain dataset and before the beginning delimiter of the next one), the file can contain lines (for example,

Number	*Name*
1700	Material Database Header
1701	Material Database Dimension
1702	Material Database Unit System
1703	Material Database Property
1704	Material Database Material Type
1705	Material Database Variable
1706	Material Database Material Class
1707	Material Database Material Attribute
1708	Material Database Material Component
1709	Material Database Material Specification
1710	Material Database Material
1750	Material Database Property Table
2400	Model Header
2411	Nodes - Double Precision
2412	Elements
2413	Result type definition
2414	Analysis Data
2415	Laminate Definitions
2416	Trace Lines
2417	Permanent Groups
2418	Sorted results associativity
3001	Thermal Coupling
3002	Temperature Boundary Condition
3003	Heat Load Boundary Condition
3004	Heat Flux Boundary Condition
3005	Thermostat
3006	Initial Temperature
3007	Radiation Request
3008	Solar Heating
3009	Radiative Heat Source
3010	Reverse Sides
3011	Forced Convective Coupling
3012	Free Convective Coupling
3014	Network Pressure Boundary Condition
3015	Network Flow Definition
3016	Merge set
3017	Elimination Set
3018	Locked Element Set
3019	Element Deactivation Set
3020	Tabular Data

Table 2.1: **Examples of universal file datasets, chosen from the Simulation application of I-DEAS.**

comments), which are not part of any dataset.

2.1.2 Examples of dataset format specifications

Here are two examples of specifications of the contents of dataset bodies. These are two of the dataset types encountered in a typical universal file that is handled by our translator.

1. Nodes - Double Precision (Universal Dataset Number 2411)

```
Name:   Nodes - Double Precision
Status: Current
Owner:  Simulation
Revision Date: 23-OCT-1992
----------------------------------------------------------------------------


Record 1:        FORMAT(4I10)
                 Field 1       -- node label
                 Field 2       -- export coordinate system number
                 Field 3       -- displacement coordinate system number
                 Field 4       -- color
Record 2:        FORMAT(1P3D25.16)
                 Fields 1-3    -- node coordinates in the part coordinate
                                  system

Records 1 and 2 are repeated for each node in the model.

Example:

    -1
  2411
       121         1         1        11
   5.0000000000000000D+00   1.0000000000000000D+00   0.0000000000000000D+00
       122         1         1        11
   6.0000000000000000D+00   1.0000000000000000D+00   0.0000000000000000D+00
    -1

----------------------------------------------------------------------------
```

2. Elements (Universal Dataset Number 2412)

```
Name:   Elements
Status: Current
Owner:  Simulation
Revision Date: 14-AUG-1992
-----------------------------------------------------------------------

Record 1:        FORMAT(6I10)
                 Field 1       -- element label
                 Field 2       -- fe descriptor id
                 Field 3       -- physical property table number
                 Field 4       -- material property table number
                 Field 5       -- color
                 Field 6       -- number of nodes on element

Record 2:  *** FOR NON-BEAM ELEMENTS ***
                 FORMAT(8I10)
                 Fields 1-n    -- node labels defining element

Record 2:  *** FOR BEAM ELEMENTS ONLY ***
                 FORMAT(3I10)
                 Field 1       -- beam orientation node number
                 Field 2       -- beam fore-end cross section number
                 Field 3       -- beam  aft-end cross section number

Record 3:  *** FOR BEAM ELEMENTS ONLY ***
                 FORMAT(8I10)
                 Fields 1-n    -- node labels defining element

Records 1 and 2 are repeated for each non-beam element in the model.
Records 1 - 3 are repeated for each beam element in the model.

Example:
```

```
    -1
  2412
         1        11         1      5380         7         2
         0         1         1
         1         2
         2        21         2      5380         7         2
         0         1         1
         3         4
         3        22         3      5380         7         2
         0         1         2
         5         6
         6        91         6      5380         7         3
```

```
        11        18        12
         9        95         6      5380         7         8
        22        25        29        30        31        26        24        23
        14       136         8         0         7         2
        53        54
        36       116        16      5380         7        20
       152       159       168       167       166       158       150       151
       154       170       169       153       157       161       173       172
       171       160       155       156
    -1
-----------------------------------------------------------------------------
```

2.2 Processing of the Universal File

Processing of the universal file begins by searching for the first delimiter line. Next, the dataset type line is processed to determine whether or not the reading program should process this dataset. If the dataset is to be processed, the program reads the data per the specifications in the remainder of this section of the manual. If the dataset is not to be processed by the program, the program continues reading until the next delimiter line indicating the end of the dataset is encountered.

The program next searches forward for the next delimiter indicating the beginning of the next dataset. Processing of the next dataset continues as above.

This processing continues dataset by dataset until the end-of-file condition is reached. Note that an end of file condition encountered in the middle of the dataset does not mean that the dataset is complete. The end of file should always be encountered while looking for the beginning of the next dataset to process.

2.3 Writing of Universal File by User-written Program

Universal files written by I-DEAS generally contain more information than is required if the file is written outside of I-DEAS in order to import information. For example, files written by the software contain information about the model file being used and active units. When reading a file, these sections don't need to be present. There is also information on the file describing things like color definitions, viewport layout patterns, and light source definitions. In writing a program to import data into I-DEAS, therefore, any information that does not pertain to the user's application can be skipped.

Chapter 3

PATRAN Neutral Files

The “neutral file” is a key element of the PATRAN “neutral system”, which is PATRAN’s communication link to computer programs in the outside world [9]. Neutral system information flow is bi-directional; that is, data can be transmitted from PATRAN’s database to the outside and data can be transmitted from outside into PATRAN’s database.

3.1 Structure of Neutral Files

Neutral files are in general shorter than universal files because, as pointed out in Section 2.3, universal files generally contain more information than required for translation. The neutral file may contain any of all of the following models:

- An analysis model
- A geometry model
- A conceptual solid model

Just like a universal file, a neutral file is a concatenation of basic building blocks. The building block is this case is called a “packet”. Each packet contains two or more “data cards”, each data card being essentially a data string terminated by a newline character. Each packet contains the data for a fundamental unit of the model, such as

Summary Packets:

Type Number	*Packet Description*
25	File title
26	Summary Data

Finite Element Model Packets:

Type Number	*Packet Description*
1	Node data
2	Element data
3	Material properties
4	Element properties
5	Coordinate frames
6	Distributed loads

Trimmed Surface Solid Model Packets:

Type Number	*Packet Description*
46	Primitive data
47	Primitive face data
99	End of file tag

Table 3.1: **Partial list of data packet types in PATRAN neutral files.**

the coordinates and attributes of a specific node or the definition of a specific finite element. The basis for the term "neutral" is that the formats of the various data packets are not formatted for any particular analysis program.

3.2 Data Packet Types in the Neutral File

Analogous to datasets in universal files, data packets in neutral files come in a large number of varieties. Table 3.1 is a partial list of the types of data packets in neutral files. For a complete list, the reader should refer to Chapter 29 of the PATRAN Plus User Manual [9].

3.3 Data Packet Contents and Format

Each data packet contains two or more data cards, the first of which is always a header card and has a standard format. The rest of the data cards have contents and format which are dependent on the packet type. In this section, the format of the header card is described, and two examples of data packet formats are given, to give a preliminary impression of data packets.

3.3.1 Packet Header

The header card for each data packet can be outlined as in the following table:

Header Card *Format* (I2, 8I8)

IT	ID	IV	KC	N1	N2	N3	N4	N5

IT =	Packet Type
ID =	Identification number. A "0" ID means not applicable (n/a).
IV =	Additional ID. A "0" value means not applicable (n/a).
KC =	Card Count (number of data cards after the header)
N1 to N5	Supplemental integer values used and defined as needed.

3.3.2 Examples of Data Packet Format

Chosen as examples here are packet type 25 and packet type 01, which correspond to Title Card and Node Data respectively.

1. **Packet Type 25: Title Card**

Header Card *Format* (I2,8I8)

25	ID	IV	KC

ID =	0	not/applicable
IV =	0	not/applicable
KC =	1	

User Title Card *Format* (20A4)

TITLE

TITLE = Identifying title may contain up to 80 characters

Actual example of text:

```
25        0         0         1         0         0         0         0         0
default meshed file name
```

2. Packet Type 01: Node Data

Header Card *Format* (I2,8I8)

1 ID IV KC

ID = Node ID

IV = 0 n/a

KC = 2

Data Card 1 *Format* (3E16.9)

X Y Z

X = X Cartesian Coordinate of Node

Y = Y Cartesian Coordinate of Node

Z = Z Cartesian Coordinate of Node

Data Card 2 *Format* (I1, 1A1, I8, I8, I8, 2X, 6I1)

ICF GTYPE NDF CONFIG CID PSPC

ICF = Condensation flag (0 = unreferenced)

GTYPE = Node type

NDF = Number of degrees of freedom

CONFIG = Node configuration

CID = Coordinate frame for analysis results

PSPSC = 6 permanent single point constraint flags (0 or 1)

Actual example of text:

```
1       1       0       2       0       0       0       0       0
5.333069027E+01 1.445868343E+01 0.000000000E+00
```

Chapter 4

Problem Specification

Although there exist a large number of universal file dataset types, the typical universal files being translated by UNV2PNF contain datasets which are of only a subset of all the possible dataset types. Moreover, the sequence of datasets in these universal files follow a similar pattern. This reflects the fact that MEMCAD deals with 3-D solid models in a certain class of problems. In this chapter the translation problem is specified in detail. Specifically, we will describe the typical types of 3-D solid models from which the universal files in question are generated, and we will also look at the syntax and semantics of typical universal files and related PATRAN neutral files. In addition, the existing environment in which UNV2PNF is to be developed is described, namely, the C++ developing environment and the PNF library. Throughout the discussion, an example of a 3-D solid model is used to illustrate a typical scenario of the translation problem.

4.1 3-D Solid Models

The 3-D solid models from which the universal files in question are generated are typically made up of basic parts like blocks, cylinders, cones, spheres and tubes. An example is shown in Figure 4-1, which is a single block of length 100, height 50 and depth 20, created using the Master Modeler Task in the Design Application of I-DEAS.

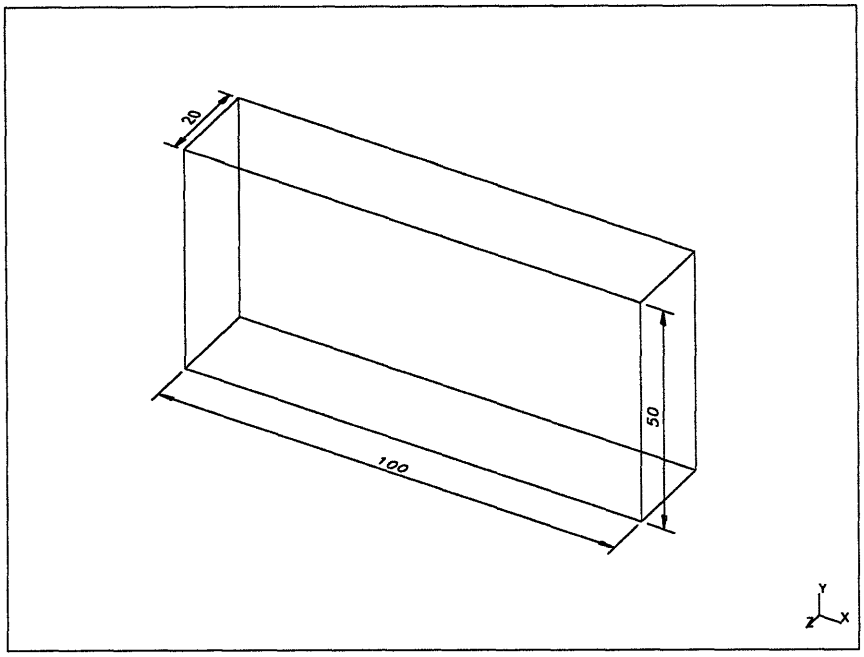

Figure 4-1: **A simple block, created using the Master Modeler Task in the Design Application of I-DEAS.**

A 3-D solid model is typically divided into a grid of "nodes" and "elements" which form a model of the real structure. This process is called Finite Element Modeling (FEM). The resulting elements are then analyzed. This is called Finite Element Analysis (FEA). A typical 3-D solid model that UNV2PNF encounters contains one or more finite element models. The following subsections describe the above concepts in more detail.

4.1.1 Finite Element Modeling and Analysis

Finite Element Analysis (FEA) is a process which predicts deflections and other effects of stress on a structure. Finite Element Modeling (FEM) divides the structure into a grid of "elements" which form a model of the real structure [10]. This process is called "meshing". Each of the elements is a simple shape (such as a square or a cube) for which the finite element program has information to write the governing equations in the form of a stiffness matrix. An example of an element is the 8-noded

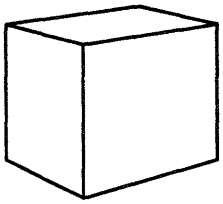

Figure 4-2: **An 8-noded Linear Brick element.**

Linear Brick, which is shown in Figure 4-2.

The unknowns for each element are the displacements at the "node" points, which are the points within an element that form the basis of the finite element model. Nodes also serve as the points at which the elements are connected. The finite element program will assemble the stiffness matrices for these simple elements together to form the global stiffness matrix for the entire model. This stiffness matrix is inverted to find the unknown displacements, given the known forces and boundary conditions. From the displacements at the nodes, the stresses in each element can then be calculated.

As an example, the block in Figure 4-1 has been meshed into 30 (5 by 3 by 2) 8-noded Linear Brick elements, with a total of 72 distinct nodes, using the Meshing Task in the Simulation Application of I-DEAS. This is shown in Figure 4-3.

4.1.2 Element Types

There is a large number of element types used in finite element modeling in I-DEAS. These elements are categorized by family, order, and topology [10]. Family refers to the characteristics of geometry and displacement that the element models. The most common families used for typical structural models are: Beam, Plane Stress, Plane Strain, Axisymmetric Solid, Thin Shell, and Solid. Order refers to the order of the equations used to interpolate the strain between nodes, such as linear, parabolic, or cubic. Linear elements have two nodes along each edge, parabolic and higher order elements have three or more. Some elements have interior nodes in addition to the nodes along the edges. Topology refers to the general shape of the element, such as triangular or quadrilateral.

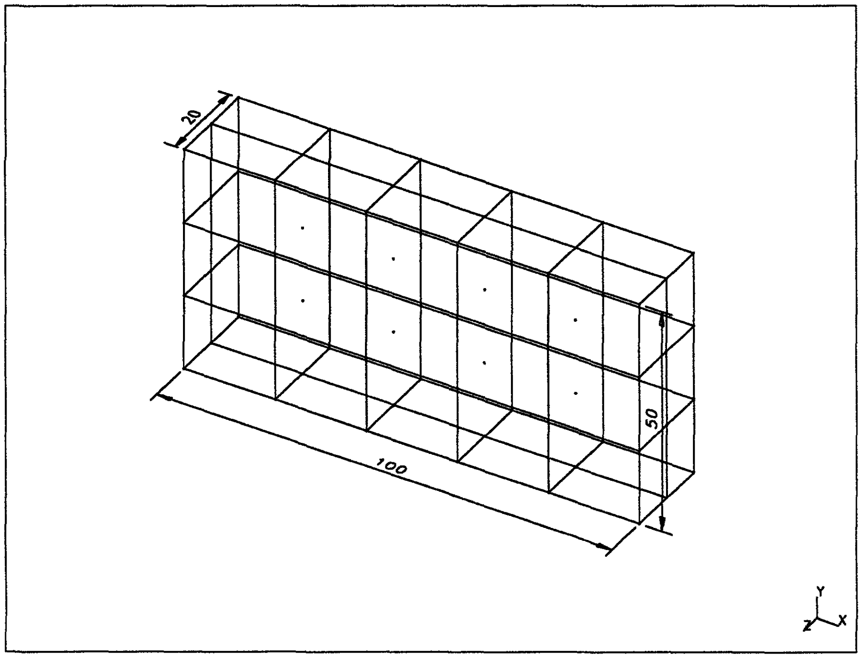

Figure 4-3: **A simple block meshed into 30 8-noded Linear Brick elements, with 72 distinct nodes, created using the Meshing Task in the Simulation Application of I-DEAS.**

FE descriptor ID	*Element type*
81	Axisymmetric Solid: Linear Triangle
84	Axisymmetric Solid: Linear Quadrilateral
82	Axisymmetric Solid: Parabolic Triangle
85	Axisymmetric Solid: Parabolic Quadrilateral
111	Solid: Linear Tetrahedron
112	Solid: Linear Wedge
113	Solid: Parabolic Wedge
115	Solid: Linear Brick
116	Solid: Parabolic Brick
118	Solid: Parabolic Tetrahedron

Table 4.1: **Common element types and their FE descriptor IDs.**

In I-DEAS, each element type has a unique identification number called the FE descriptor ID. This is used as a reference to the element type, for example from an Elements dataset. For a partial list of common element types and their FE descriptor ID's, see Table 4.1. The complete list can be found in Smart View [8].

Figure 4-4 is a visualization of 3-dimensional elements which belong to the Solid family and are used frequently in finite element modeling.

4.2 Syntax and Semantics of Universal Files

A universal file generated from a 3-D solid model, together with associated finite element models, typically has a specific sequence of datasets which together encapsulate information about the model file, the materials used, the nodes and elements in all finite element models, and so on.

As an example, the sequence of datasets of a universal file generated from the block in Figure 4-1 is listed in Table 4.2. Out of all these datasets, only a fraction are relevant[1] in MEMCAD and thus required to be processed in the translation by UNV2PNF. In their order of appearance, the first one is the Header dataset (151), which contains the model file name and model file description. Second is the Units

[1] We use the term "relevant" to refer to information in the universal file which is required for analysis in MEMCAD and thus needs to be included in the translation.

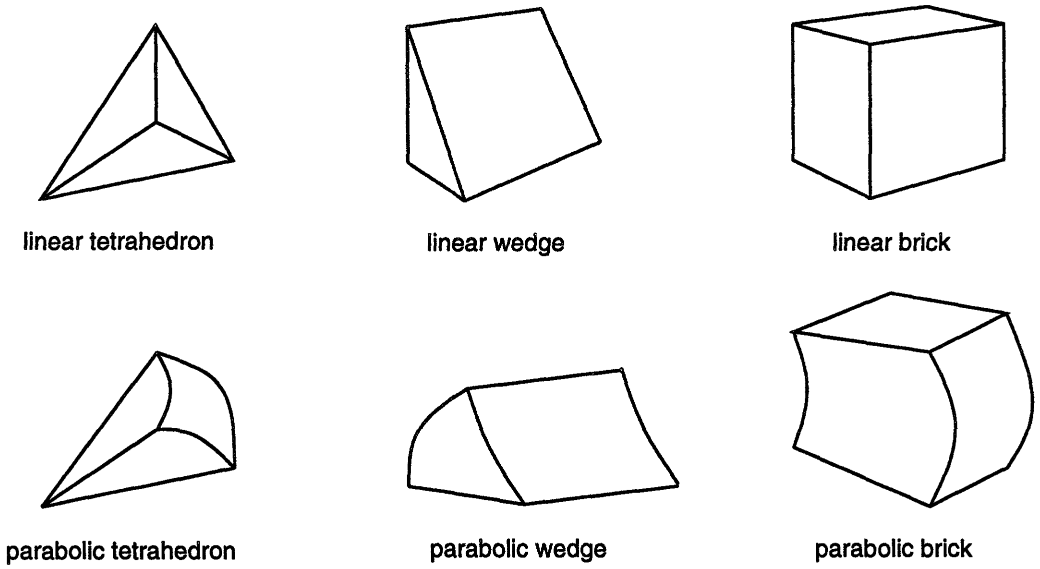

Figure 4-4: **3-dimensional elements in the Solid family in I-DEAS.**

151	164	3211	3202	3212	3212	3212	1700	1703	1705
1710	1712	1101	3209	789	2400	2420	2411	2412	734

Table 4.2: **Sequence of datasets of a universal file generated from a simple block.**

dataset (164), which contains information about the units used in the model file. After a few irrelevant datasets is a group of datasets which contain material information: datasets 1700, 1703, 1705, 1710 and 1712. Each finite element model is represented by a group of datasets: 2400, 2420, 2411 and 2412.

Each of these datasets has its own content and format specification dependent on its dataset number, as described in Section 2.1.1. Refer to Smart View [8] for the exact specifications.

25	26	3	4	1	1	⋯	2	2	⋯	99

Table 4.3: **Sequence of data packets of a PATRAN neutral file generated from a simple block.**

4.3 Syntax and Semantics of PATRAN Neutral Files

Similar to the case of universal files, a PATRAN neutral file representing a 3-D solid model file has a specific sequence of data packets.

Table 4.3 lists the sequence of data packets in a neutral file representing the block in Figure 4-1. At the beginning are a File title packet (25) and a Summary packet (26) which contain general information about the model file. After that are a set of Material properties packets (3) and a set of Element properties packets (4), one for each of the properties. Node data packets (1) and Element data packets (2) follow, one for each node or element. At the end is a End of file tag packet (99) which signifies the end of the neutral file.

Refer to Chapter 29 of the PATRAN Plus User Manual [9] for detailed content and format specifications of these data packets.

4.4 Development Environment of UNV2PNF

The translation of a universal file into a neutral file essentially involves two stages:

1. reading relevant information from the source universal file and storing the information in memory using an appropriate representation, and
2. writing the information into the target neutral file.

UNV2PNF deals chiefly with the first stage of this translation process, that is, the reading of pieces of relevant information from the universal file, storing them in memory. The second stage, the writing of the information into the neutral file, is handled by an existing utility in MEMCAD, the PNF library, which is a collection of C++ class definitions and functions. Implementing UNV2PNF in C++ codes is thus a natural choice which facilitates the flow of data between the two stages of the translation process.

4.5 Extensibility of UNV2PNF

The universal file format has a long and expanding specification; new datasets are continually under construction and modification. Moreover, the future developments of MEMCAD might change or extend the translation problem: more datasets might become relevant, or more element types might need to be handled at a later time. Hence arises the need for extensibility — UNV2PNF needs to be designed such that as new datasets or new element types come into play, it is reasonably easy, efficient and economical for the user or the developer to accommodate these changes.

Chapter 5

Design and Implementation

UNV2PNF was written entirely in C++ [11], and consists of fourteen classes. See Appendix A for the purpose and declaration [11] of each class. This chapter describes the overall strategy in the design of the translator, the translation process and various implementation issues, and also discusses how extensibility is handled.

5.1 Overall strategy

As mentioned in Section 4.4, UNV2PNF deals chiefly with the first stage of the translation process, namely, the reading of information from the universal file and the storing of the information in memory using some appropriate representation. In object-oriented programming methodology, it is natural to create a class [11] to encapsulate the information in a universal file. The class `Unv` is thus created for this purpose. This is analogous to the class `PATRAN_neutral_file` in the PNF library, which encapsulates the information in a PATRAN neutral file.

Since the second stage of the translation process is handled by the PNF library, information needs to be passed from UNV2PNF to the PNF library. This is achieved by a member function [11] `fillPNF()` of `Unv` which transfers information from the `Unv` object to the `PATRAN_neutral_file` object by calling member functions of `PATRAN_neutral_file`.

The overall strategy is thus to:

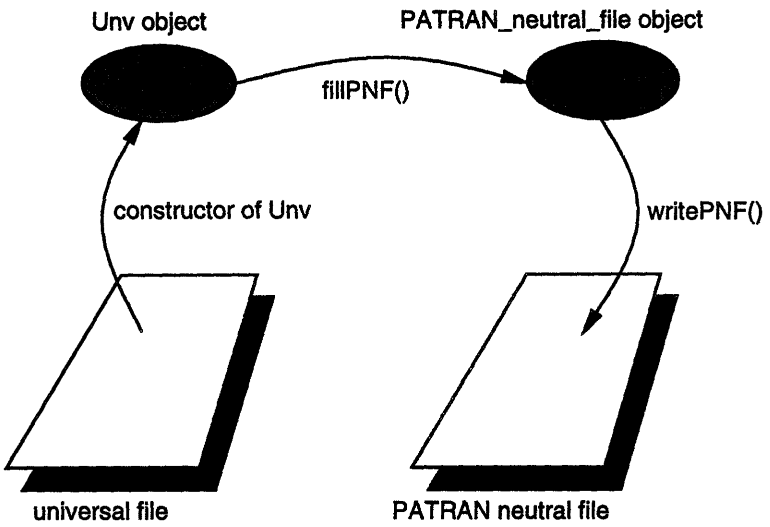

Figure 5-1: **Overall strategy in translation.**

Step 1: read information from the universal file into a `Unv` object,

Step 2: transfer the information from the `Unv` object to a `PATRAN_neutral_file` object, and

Step 3: write the information from the `PATRAN_neutral_file` object to the PATRAN neutral file.

Step 1 is done by a constructor [11] of `Unv`. Step 2 is done by the `fillPNF()` member function of `Unv`. Step 3 is done by the `writePNF()` member function of `PATRAN_neutral_file`. This is illustrated in Figure 5-1.

The next two sections describe in detail the design and implementation of UNV2PNF, corresponding the Step 1 and Step 2 above. Step 3 is handled entirely by the PNF library, and details about the library can be found in MemBase.

The implementation of UNV2PNF consists of fourteen classes. Instead of describing the classes one by one, we will go through the translation process and describe each class as it comes into play. We will also highlight special implementation issues as they become relevant in the discussion.

5.2 Reading from Universal File

5.2.1 Use of Streams

The reading and manipulation of data from the universal file are implemented using streams [11]. The stream input/output library is a powerful I/O facility in C++, and is highly extensible to handle user-defined types.

A file stream is kept in the `Unv` object and is a link to the universal file. All information is read through this file stream. The translation process begins with opening the file through this file stream.

To get a glimpse of how information is read from the universal file into the `Unv` object through a stream, consider a simple example. The following is an excerpt from the format specification of the Elements dataset (universal dataset number 2412):

```
Record 1: FORMAT(6I10)
Field 1 -- element label
Field 2 -- fe descriptor id
Field 3 -- physical property table number
Field 4 -- material property table number
Field 5 -- color
Field 6 -- number of nodes on element
```

This specifies the format of the first record of the dataset. There are 6 fields in the record, each being 10 characters long and representing an integer. An actual example of text is:

```
         1       115         1         1         7         8
```

The following code segment does the job of reading the information into a `Unv` object:

```
s >> u.elementID >> u.type >> u.phyPropTableNum >> u.matPropTableNum
  >> u.color >> u.numNodes;
```

where s is a file stream which has been tied to the universal file previously, and u.elementID, u.type, etc. are members of the Unv object u. The six integers are read one by one into the six member variables of u.

This is a primitive example of how information from a universal file is read through a stream. In UNV2PNF, more sophisticated features of streams have been utilized in the reading of information from universal files, such as operator overloading and manipulators [11].

5.2.2 Scanning of Datasets

After the universal file is opened through the file stream, a prescanning of the entire universal file is done to get a count of the total number of datasets in the file. This count is used to allocate memory for a list of dataset information records, which are objects of the class Dataset. After the prescanning is done, an actual scanning is done to get the list of dataset information:

```
// Counts how many datasets:
numDatasets = 0;
char line[LINE_LENGTH];
while (f.getline(line, LINE_LENGTH))
  if (strcmp(line, delimiter) == 0) {   // if beginning delimiter
    while (f.getline(line, LINE_LENGTH),  // eats up dataset
           strcmp(line, delimiter) != 0)
        ;
    numDatasets++; }
cout << numDatasets << " datasets found.\n";

// Allocates memory for the dataset list:
datasetList = new Dataset[numDatasets];

// Reads in the dataset infos:
cout << "Scanning over all datasets...\n";
f.clear();  // reset the state of f to good()
f.seekg(0, f.beg);
```

```
for (int i = 0; i < numDatasets; i++)
  f >> datasetList[i];
```

The dataset information of each dataset includes its dataset number and location in the file. This information is used later in selecting and accessing the relevant datasets directly.

5.2.3 Units

After the above dataset information is read in, the first relevant dataset in the universal file, namely, the Units dataset (dataset number 164) is read in. There are nine different types of unit systems used in I-DEAS:

```
1 - SI: Meter (newton)
2 - BG: Foot (pound f)
3 - MG: Meter (kilogram f)
4 - BA: Foot (poundal)
5 - MM: mm (milli newton)
6 - CM: cm (centi newton)
7 - IN: Inch (pound f)
8 - GM: mm (kilogram f)
9 - US: USER_DEFINED
```

Information about the unit system is necessary to make sure that all quantities are converted to their proper units before being transfered to the `PATRAN_neutral_file` object. A class `Units_set` is created for storing the unit system information.

5.2.4 Materials

The next relevant datasets are a sequence of the Material Database Material datasets (dataset number 1710), each of which contains information on one material. Each element in a finite element model points to one material, and hence one of these Material Database Material datasets.

The information on a material includes various quantitative properties of the material being represented, of which only the modulus of elasticity, Poisson's Ratio, mass density and shear modulus are relevant in the present version of MEMCAD and need to be included in the translation.

The class `Material_set` is created to store information from the Material Database Material dataset. Analogous to the prescanning to get a count of the number of datasets in Section 5.2.2, a prescanning to get a count of the number of Material Database Material datasets is done prior to the allocation of memory and the actual reading of the datasets:

```
// Counts number of materials and allocates memory:
numMaterials = 0;
for (i = 0; i < numDatasets; i++)
  if (datasetList[i].getDatasetNumber() == MATERIAL_SET_NUMBER)
    numMaterials++;
materials = new Material_set[numMaterials];

// Reads in materials:
int currMat = 0;
for (i = 0; i < numDatasets; i++)
  if (datasetList[i].getDatasetNumber() == MATERIAL_SET_NUMBER) {
    f.seekg(datasetList[i].getBeginPos());
    f >> materials[currMat];
    cout << "Material found:\t" << materials[currMat].getNumber()
          << '\t' << materials[currMat].getName() << endl;
    currMat++;
  }
```

5.2.5 Finite Element Models

Following the Material Database Material datasets is the reading of the most important datasets in the universal file: datasets containing information on the finite element models. Each finite element model corresponds to four consecutive datasets:

1. **Model Header (dataset number 2400)**

This contains the name of the finite element model. The class `Fem_name_set` is created to store this information.

2. **Coordinate Systems (dataset number 2420)**

   This contains the name of the part from which the finite element model is constructed. The class `Part_name_set` is created to store this information.

3. **Nodes - Double Precision (dataset number 2411)**

   This contains information on all the nodes in the finite element model. The classes `Nodes_set` and `Unv_node` are created to store this information. A `Nodes_set` object encapsulates the whole set of nodes in the FEM collectively, while a `Unv_node` object stores information on only one node. A `Nodes_set` object thus contains a list of `Unv_node` objects.

   Each node carries the following information:

   (a) node label

   (b) export coordinate system number

   (c) displacement coordinate system number

   (d) color

   (e) node coordinates in the part coordinate system

   Only the node label and node coordinates are relevant and read into each `Unv_node` object.

   Since the body of this dataset does not explicitly contain a count of the nodes in the dataset, a prescanning of the dataset to get a node count is done prior to the allocation of memory and the actual reading, as in Section 5.2.2.

4. **Elements (dataset number 2412)**

   This contains information on all the elements in the finite element model. Analogous to the classes `Nodes_set` and `Unv_node`, the classes `Elements_set` and `Unv_element` are created to store this information. A `Elements_set` object encapsulates the whole set of elements in the FEM collectively, while a `Unv_element` stores information on only one element. A `Elements_set` object thus contains a list of `Unv_element` objects.

Each element carries the following pieces of information:

(a) element label

(b) fe descriptor

(c) physical property table number

(d) material property table number

(e) color

(f) number of nodes on the element

(g) information on nodes

All of these except physical property table number and color are relevant, though all of them are read into each `Unv_element` object. Physical property table number and color are kept for possible future extension.

As in the case of dataset number 2411, this dataset does not explicitly contain a count of the elements in the dataset. Hence a prescanning of the dataset to get a element count is done prior to the allocation of memory and the actual reading, as in Section 5.2.2.

To encapsulate the relationship between these four datasets, the class `Fem` is created. Each `Fem` object captures a finite element model and contains pointers to four objects storing information from the four datasets representing the model.

The following code segment is where the finite element models are read from the universal file:

```
fns = new Fem_name_set;
pns = new Part_name_set;
```

```
ns = new Nodes_set(fns, pns);
es = new Elements_set(fns, pns);

f.seekg(datasetList[i].getBeginPos());
f >> (*fns) >> (*pns) >> (*ns) >> (*es);
cout << "  FEM found:\tFEM name \"" << fns->getFemName()
     << "\"\n\t\tPart name \"" << pns->getPartName() << "\"\n";

connect(ns, es);
addFem(fns, pns, ns, es);
```

Note that the four datasets are read consecutively into memory using only one simple and elegant statement, demonstrating the expressive power and flexibility of stream operations. The function `connect(ns, es)` links together the `Nodes_set ns` and the `Elements_set es` by hashing their pointers to each other. The function `addFem(fns, pns, ns, es)` creates a new `Fem` object and adds it to the existing list of `Fem` objects.

5.3 Transfer to `PATRAN_neutral_file` Object

Step 2, the transfer of information from the `Unv` object to a `PATRAN_neutral_file` object (cf. Section 5.1), is done by the `fillPNF()` member function of `Unv`, once all relevant information has been read into the `Unv` object. It involves three stages: (1) filling in the nodes, (2) filling in the material properties list, and (3) filling in the elements and element properties. This section describes important implementation issues encountered in the process.

5.3.1 Difference in Storage of Nodes and Elements

There is a fundamental difference between the way a universal file stores nodes and elements and the way a PATRAN neutral file stores them. A universal file organizes all nodes and elements into FEMs, while a PATRAN neutral file puts all nodes and elements in the file onto two single lists, and the concept of FEMs only exists by having

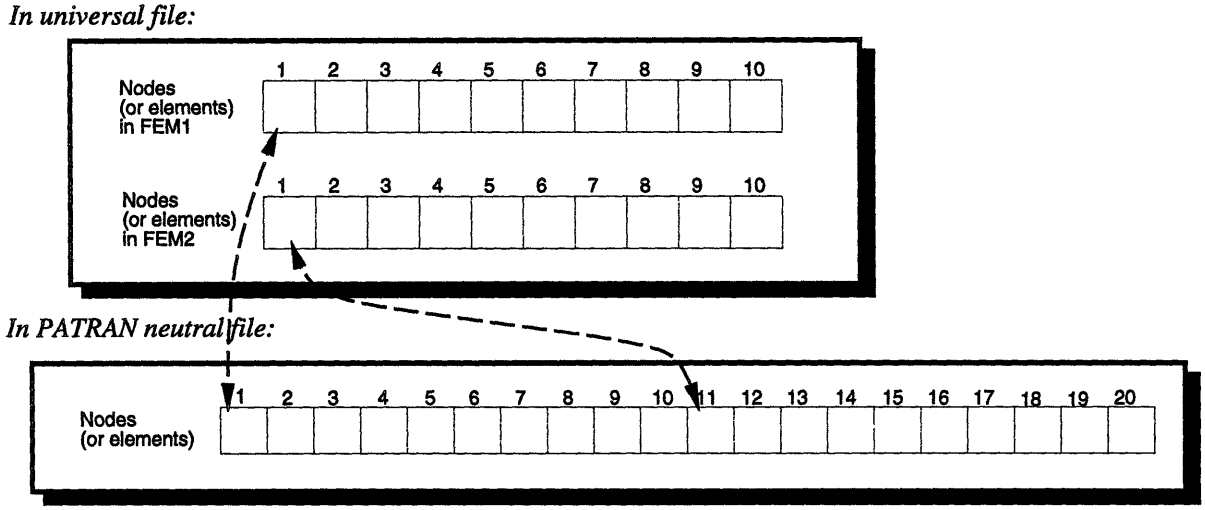

Figure 5-2: **Straightening of node IDs.**

pointers to element properties on elements — elements within one FEM should point to the same element property.

This difference has two important consequences:

1. Node IDs (or labels) and element IDs have to be "straightened out" in the PATRAN neutral file. For example, if in the universal file we have nodes with IDs 1, 2, . . . , 10 in one FEM and also nodes with the same IDs in another FEM, we need to change the IDs of second set of nodes (to 11, 12, . . . , 20, for example) in order to maintain the uniqueness of each node's ID in the PATRAN neutral file, since all nodes are put on a single list in the neutral file. This is illustrated in Figure 5-2.

2. Element properties have to be repeated for two elements having the same material property but being in two different FEMs.

   This is illustrated in Figure 5-3. Note first that:

   - In the universal file, an element points to a material property directly, while in the neutral file, an element points only to an element property which in turn points to a material property.

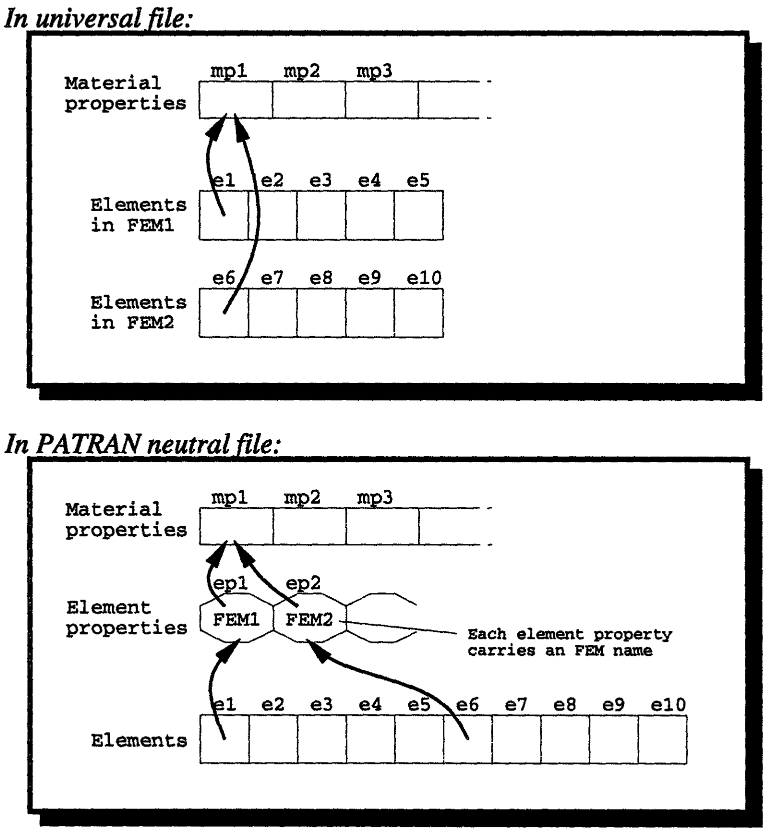

Figure 5-3: **Repeating of element properties in neutral file for elements in different FEMs but having the same material property.**

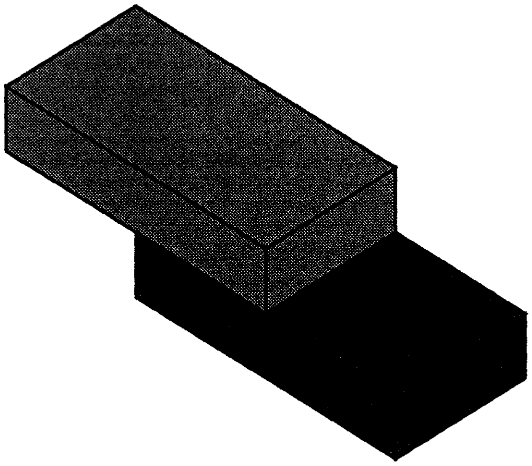

Figure 5-4: **Two partially overlapping blocks of different materials.**

- In the neutral file, an element property carries an FEM name, identifying the FEM the element is in.

The elements $e1$ and $e6$ in the universal file point to the same material, $mp1$, though they are in different FEMs. Since all elements in the neutral file are in a single list, the only distinction we can make between $e1$ and $e6$ in the neutral file is by pointing them to two different element properties, $ep1$ and $ep2$, both pointing to $mp1$ but having different FEM names. The classes `Pnf_elem_prop` and `Pnf_elem_prop_list` are created to keep track of these material properties and element properties.

5.3.2 Shared Nodes

An interesting problem arises when nodes are shared by elements of different materials. An example is shown in Figure 5-4. It shows two rectangular blocks of different materials partially overlapping. Suppose we mesh both blocks into linear bricks of equal size such that at the surface where the two blocks touch, there are nodes which are shared by elements of different materials. The problem which arises is how such nodes are represented in the universal file and PATRAN neutral file while retaining the information about the two different materials.

I-DEAS element type	*PNF library element type*
81	Shell
82	Shell
84	Quad
85	Quad
111	Tetrahedron
112	Wedge
113	Wedge
115	Hex

Table 5.1: **Mapping of I-DEAS element type to PNF library element type.**

This problem is easily solved by the repeating of element properties described in Section 5.3.1. As shown in Figure 5-5, the situation is represented in the universal file as two linear brick elements, $e27$ and $e93$, belonging to the upper block and lower block respectively, and having material properties $mp1$ and $mp2$ respectively. Each of them has an associated node which is shared, but since a universal file organizes nodes into FEMs, this shared node has to appear as two separate nodes, $n21$ and $n45$, in the two FEMs. The result of translation is shown in the lower part of the figure. In the resulting PATRAN neutral file, note that:

- The shared node appears only once as $n21$ in the single list of nodes, eliminating redundancy. $e27$ and $e93$ both point to it.

- Two element properties, $ep1$ and $ep2$, are created, one for $e27$ and one for $e93$. They point to $mp1$ and $mp2$ respectively.

5.3.3 Element Type Mapping

There are a large number of element types in I-DEAS, whereas the PNF library only makes a distinction among six different types: Bar, Shell, Quad, Tetrahedron, Wedge and Hex. Hence a (many-to-one) mapping of element types needs to be done. Table 5.1 shows part of the complete mapping, which is implemented as a static member [11] array in class `Unv`.

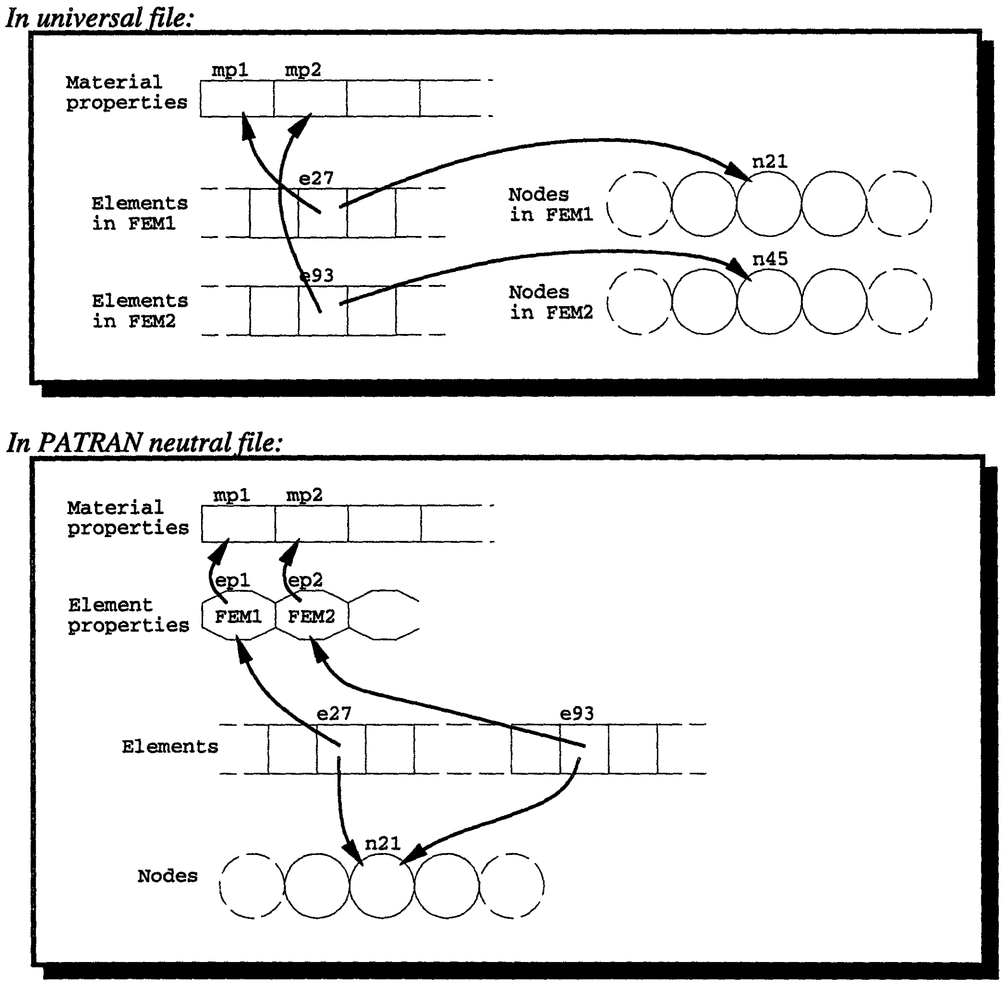

Figure 5-5: **The solution to the problem of shared nodes.**

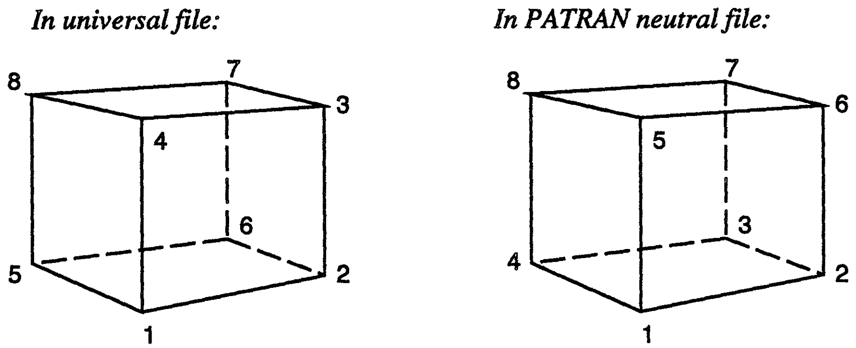

Figure 5-6: **Ordering of nodes on a linear brick in universal files and in PATRAN neutral files.**

Node in universal file	*Node in neutral file*
1	1
2	2
3	6
4	5
5	4
6	3
7	7
8	8

Table 5.2: **Mapping of nodes on a linear brick from universal files to PATRAN neutral files.**

5.3.4 Node Mapping

Universal files and PATRAN neutral files also differ in the ordering of nodes in certain element types. An example is the linear brick. Figure 5-6 shows the orderings of nodes on a linear brick in universal files and in PATRAN neutral files. The mapping of nodes is shown in Table 5.2. Again, this mapping is implemented as a static member array in class `Unv`. In the current version of UNV2PNF, node mapping is done for the element types linear brick and parabolic brick. However, straightforward extension can be made to other element types.

5.4 Extensibility

UNV2PNF is "extensible" because it is object-oriented. Specifically, the fact that it is implemented using an object-oriented programming language like C++ enables new features to be added to the translator easily. Some possibilities of such extensions are:

1. **New element types**

   UNV2PNF currently only handles a subset of all possible element types in I-DEAS. If it is later desirable to handle other types, the following should be done:

   (a) Add a new class in the PNF library representing the corresponding element type, analogous to Bar, Shell, etc.

   (b) Add an entry in the element mapping table `elTypeMap` in class `Unv`.

   (c) Add a case to the `switch` statement in `fillPNF()`, corresponding to the new PNF class for the new element type.

2. **New datasets**

   Only certain datasets in a universal file are currently relevant. If later developments of MEMCAD render more information in the universal file relevant, some of the datasets which are currently ignored must be read. This can be done by adding a new class in UNV2PNF corresponding to the new dataset, analogous to `Nodes_set`, `Units_set`, etc.

Chapter 6

Conclusion

In this thesis, a file translator, UNV2PNF, was designed and implemented to bridge the gap between I-DEAS and MemBase. The gap was caused by a data incompatibility problem due to the switching from PATRAN to I-DEAS. Such data incompatibility or format incompatibility often occurs in software systems, particularly to ones which undergo repeated updates and modifications. Using an object-oriented language proved to be an effective and powerful approach in the development of such a file translator, and made extension of the translator easy.

UNV2PNF has been integrated into MemBase and is part of MEMCAD 2.0.

6.1 Summary Information about Implementation

All descriptions of UNV2PNF in this thesis correspond to the latest working version of UNV2PNF. The source C++ code has a total of 1689 lines, including all function and class declarations and definitions. This count does not include the top-level `main()` function, because UNV2PNF is to be integrated into MemBase as a library.

As an evaluation of the run time of UNV2PNF, it was run on a Sun SPARCstation IPX running SunOS Release 4.1.3 (GENERIC) UNIX, using several input universal files with certain statistics and the run times were measured. The results are shown in Table 6.1.

The run time grows linearly with the number of datasets, number of materials,

	File 1	File 2	File 3	File 4	File 5
File size in bytes	37339	33556	72750	64705	115156
Number of datasets	48	42	40	73	52
Number of materials	1	1	1	1	9
Number of FEMs	1	1	1	2	1
Total number of nodes in all FEMs	45	111	127	111	235
Total number of elements in all FEMs	16	12	403	44	96
Run time in seconds	1.2	1.6	3.4	1.9	3.1

Table 6.1: **Run time statistics of UNV2PNF.**

total number of nodes and total number of elements, of which the total number of nodes and total number of elements have the most significant impact on run time.

6.2 Future Work

Future work on the translator can take the directions of:

- extension of some components of the translator, and
- trying different approaches in certain stages or parts in the translation.

Here are some suggestions:

1. **Disuse of Streams**

   Because C++ and C code are often intermixed, C++ stream I/O must sometimes be mixed with the C `printf()` family of input and output functions [11]. Also, because C functions can be called from C++, some programmers may prefer to use the more familiar C I/O functions.

   Stream output functions have a general advantage over the C standard library function `printf()`: the stream functions are type safe and have a common style for specifying output of objects of built-in and user-defined types. Nevertheless, the usage of streams in this thesis is limited in terms of its robustness in the reading of universal files. The current implementation is loose and the content in input universal files do not need to adhere to exact character locations in order to be read in correctly. If such strict adherence is desired, restrictions must be imposed by modifying the stream functions in the translator. An alternative is using the C standard library function `printf()`, which already has provisions for exact specifications of character locations. This is the way the PNF library currently handles input and output.

2. **Using generic lists**

   The translators keeps and manipulates various kinds of lists, for example:

   - list of nodes in a nodes dataset set
   - list of elements in an elements dataset set
   - list of nodes on each element
   - list of material properties in the universal file
   - list of FEM records

   Currently in UNV2PNF, most of these lists are implemented as arrays, while the list of FEM records is implemented as a linked list. To capture the similarities among all these lists, one might consider creating a generic list class and deriving all specific lists from this generic list. This is actually what is being done in the PNF library. However, computational efficiency might be affected in such use of generic lists and should be taken into consideration if computational speed is crucial.

3. **Extensions**

   As described in Section 5.4, the object-oriented nature of UNV2PNF results in its extensibility. For example, new element types or new datasets can be added to the translator easily. Such and similar extensions can be considered in future endeavor.

Appendix A

Classes in UNV2PNF

The implementation of UNV2PNF consists of fourteen classes. This appendix gives a description of the purpose of each class and the declaration of each class.

A.1 Purpose of Each Class

1. `Unv`

   This is a top level class used to encapsulate all the information which is required for translation in the universal file. It is analogous to the class `PATRAN_neutral_file` in the PNF library.

2. `Dataset`

   This stores information about each dataset in the universal file, including its dataset number, beginning location in the file, and size. These pieces of information are used in the actual reading of data from that dataset in the universal file into an object in memory.

3. `Units_set`

   This stores information from the Units dataset (dataset number 164), which is about the unit system being used in the universal file. This information is necessary to make sure that all quantities are converted to their proper units before being transferred to the `PATRAN_neutral_file` object.

4. `Material_set`

   This stores information from each Material Database Material dataset (dataset number 1710). Of all the quantitative properties of the material available in the dataset, only the modulus of elasticity, Poisson's Ratio, mass density and shear modulus are relevant and are stored.

5. `Fem`

   This encapsulates the relationship between four datasets which together represent a finite element model: Model Header dataset (dataset number 2400), Coordinate Systems dataset (dataset number 2420), Nodes - Double Precision dataset (dataset number 2411), and Elements dataset (dataset number 2412). It does so by keeping pointers to the four objects which store information from the four datasets above respectively.

6. `Fem_name_set`

   This stores information from the Model Header dataset (dataset number 2400), which is simply the name of the finite element model.

7. `Part_name_set`

   This stores information from the Coordinate Systems dataset (dataset number 2420), which is the name of the part from which the finite element model is constructed.

8. `Name_set`

   This is a base class from which `Fem_name_set` and `Part_name_set` are derived.

9. `Nodes_set`

   This stores information from the Nodes - Double Precision dataset (dataset number 2411), which includes the number of nodes in the dataset and the actual information on all the nodes, which are stored as a list of `Unv_node` objects. It also contains pointers to the other three objects in the finite element model sequence: an Fem_name_set object, an Part_name_set object, and an Elements_set object.

10. Unv_node

    This stores information about each node, which includes the node ID and the 3-D coordinates of the node's location.

11. Elements_set

    This stores information from the Elements dataset (dataset number 2412), which includes the number of elements in the dataset and the actual information on all the elements, which are stored as a list of Unv_element objects. It also contains pointers to the other three objects in the finite element model sequence: an Fem_name_set object, an Part_name_set object, and an Nodes_set object.

12. Unv_element

    This stores information about each element, which includes the element ID, type, physical property table number, material property table number, color, number of nodes and ID's of the nodes associated with the element. Of all these data, physical property table number and color are not included in the translation but are kept for possible future extension.

13. Pnf_elem_prop, Pnf_elem_prop_list

    These two classes keep track of the material properties and element properties created to fill the PATRAN_neutral_file object. They aid in the checking of whether a new element property needs to be created for each element encountered in the universal file.

A.2 Declaration of Each Class

Included in this section are the declarations of all fourteen classes. Note that in certain classes some member functions are defined in the class declaration, in which case it is `inline` [11].

1. Unv

```
/*
 * unv.h
 * Unv class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef UNV_H
#define UNV_H

#include <fstream.h>
#include "dataset.h"
#include "material_set.h"
#include "nodes_set.h"
#include "elements_set.h"
#include "fem.h"
#include "fem_name_set.h"
#include "part_name_set.h"
#include "units_set.h"
#include "/fs/blofeld/b/Memcad2/pnf_server/PNF.h"

#define LINE_LENGTH    81
#define MAX_FILENAME_LENGTH 80
#define UNDEFINED  -1
#define UNITS_SET_NUMBER        164
#define MATERIAL_SET_NUMBER     1710
#define FEM_NAME_SET_NUMBER     2400
#define PART_NAME_SET_NUMBER    2420
#define NODES_SET_NUMBER        2411
#define ELEMENTS_SET_NUMBER     2412


class Unv {

  struct DatasetInfo {
    int        datasetNumber;
    streampos  beginPos;
    streampos  bodySize;
  };

  static int      initialized;
  static int      elTypeMap[200];
  static int      nodeMapLB[8];   // node mapping for linear bricks
  static int      nodeMapPB[20];  // node mapping for parabolic bricks

  char           filename[MAX_FILENAME_LENGTH];
  ifstream       f;

  int           numDatasets;
  Dataset*        datasetList;

  int           numMaterials;
```

```
    Material_set*    materials;

    int              totalNumNodes;
    int              totalNumElements;
    int              numFem;
    Fem*               femlistHead;
    Fem*               femlistTail;

    Units_set          units;


    void      init();
    void      addFem(Fem_name_set* femNameSet, Part_name_set* partNameSet,
                        Nodes_set* nodesSet, Elements_set* elementsSet);
    Unv_node*   findNode(Node_ID nodeID);


  public:

    static char      delimiter[7];

    Unv(const char *inFilename);
    ~Unv();
    void      displayDatasetList();
    void      displayUnits()  { cout << units; }
    void      displayFemList();
    void      fillPNF(PATRAN_neutral_file* inPNF);
};

#endif
```

2. Dataset

```
/*
 * dataset.h
 * Dataset class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef DATASET_H
#define DATASET_H

#include <iostream.h>
#include <fstream.h>


class Dataset {

  int        datasetNumber;
  streampos  beginPos;
  streampos  bodySize;

 public:

  int        getDatasetNumber()  { return datasetNumber; }
  int        getBeginPos()   { return beginPos; }
  int        getBodySize()   { return bodySize; }
  ostream&   info(ostream& s);

  friend ifstream&    operator>>(ifstream& s, Dataset& d);
};

ifstream&   operator>>(ifstream& s, Dataset& d);

#endif
```

3. Units_set

```
/*
 * units_set.h
 * Units_set class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef UNITS_SET_H
#define UNITS_SET_H

#include <iostream.h>


class Units_set {

 public:

  int           unitsCode;
               /*
                    1 – SI: Meter (newton)
                    2 – BG: Foot (pound f)
                    3 – MG: Meter (kilogram f)
                    4 – BA: Foot (poundal)
                    5 – MM: mm (milli newton)
                    6 – CM: cm (centi newton)
                    7 – IN: Inch (pound f)
                    8 – GM: mm (kilogram f)
                    9 – US: USER_DEFINED
                    */

  /* Unit factors for converting universal file units to SI.
   * To convert from universal file units to SI divide by the
   * appropriate factor listed below.
   */
  double        length;
  double        force;
  double        temperature;
  double        temperatureOffset;

};

istream& operator>>(istream& s, Units_set& ns);
ostream& operator<<(ostream& s, Units_set& ns);


#endif
```

4. Material_set

```
/*
 * material_set.h
 * Material_set class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef MATERIAL_SET_H
#define MATERIAL_SET_H

#include <iostream.h>
#include <iomanip.h>


// "CONSTANT" type (as determined in Record 38) properties:
struct Property {
  int       propertyVersionNumber;
  char      dimensionName[40];
  char      unitName[40];
  double    propertyValue;
};


class Material_set {

  int               materialNumber;
  char              materialName[80];
  int               numMaterialVariables;
  int               numMaterialProperties;
  int               numReferenceEntities;

  Property          modulusOfElasticity;
  Property          poissonsRatio;
  Property          massDensity;
  Property          shearModulus;

  void              readMaterialVariables(istream& s);
  void              readMaterialProperties(istream& s);

 public:

  Material_set();
  int               getNumber()  { return materialNumber; }
  char*             getName()    { return materialName; }
  friend istream&  operator>>(istream& s, Material_set& ns);
  friend ostream&  operator<<(ostream& s, Material_set& ns);

  double            getModulusOfElasticity()
                    { return modulusOfElasticity.propertyValue; }
  double            getPoissonsRatio()
                    { return poissonsRatio.propertyValue; }
  double            getMassDensity()
```

```
                { return massDensity.propertyValue; }
 double           getShearModulus()
                { return shearModulus.propertyValue; }
};

istream& operator>>(istream& s, Material_set& ns);
ostream& operator<<(ostream& s, Material_set& ns);

#endif
```

5. Fem

```
/*
 * fem.h
 * Fem class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef FEM_H
#define FEM_H

#include "nodes_set.h"
#include "elements_set.h"
#include "fem_name_set.h"
#include "part_name_set.h"


typedef long Fem_ID;


class Fem {

  Fem_ID          femID;
  Fem_name_set*   femNameSet;
  Part_name_set*  partNameSet;
  Nodes_set*      nodesSet;
  Elements_set*   elementsSet;
  Fem*            next;

 public:

  Fem(Fem_ID femID, Fem_name_set* femNameSet, Part_name_set* partNameSet,
      Nodes_set* nodesSet, Elements_set* elementsSet, Fem* next);
  ~Fem();
  void           setNext(Fem* next)  { Fem::next = next; }
  Fem_ID         getFemID()          { return femID; }
  char*          getFemName()        { return femNameSet->getFemName(); }
  char*          getPartName()       { return partNameSet->getPartName(); }
  Nodes_set*     getNodesSet()       { return nodesSet; }
  Elements_set*  getElementsSet()    { return elementsSet; }
  Fem*           getNext()           { return next; }
  void           printInfo();
  void           printNodesSummary()     { nodesSet->printSummary(); }
  void           printElementsSummary()  { elementsSet->printSummary(); }
  void           printNodes()            { nodesSet->printNodes(); }
  void           printElements()         { elementsSet->printElements(); }
};


#endif
```

6. Fem_name_set

```
/*
 * fem_name_set.h
 * Fem_name_set derived class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef FEM_NAME_SET_H
#define FEM_NAME_SET_H

#include "name_set.h"


class Fem_name_set : public Name_set {

 public:

  inline char* getFemName()    { return getName(); }
};

#endif
```

7. Part_name_set

```
/*
 * part_name_set.h
 * Part_name_set derived class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef PART_NAME_SET_H
#define PART_NAME_SET_H

#include "name_set.h"


class Part_name_set : public Name_set {

 public:

  char* getPartName()   { return getName(); }
};

#endif
```

8. Name_set

```
/*
 * name_set.h
 * Name_set base class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef NAME_SET_H
#define NAME_SET_H

#include <iostream.h>

#define NAME_LENGTH 80


class Name_set {

  char name[NAME_LENGTH];

 public:

  char* getName()   { return name; }

  friend istream&   operator>>(istream& s, Name_set& ns);

};

istream&   operator>>(istream& s, Name_set& ns);

#endif
```

9. Nodes_set

```
/*
 * nodes_set.h
 * Nodes_set class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef NODES_SET_H
#define NODES_SET_H

#include "elements_set.h"
#include "unv_node.h"

#define FEM_NAME_LENGTH 80
#define PART_NAME_LENGTH 80


class Nodes_set {

  class Fem_name_set*    femNameSet;
  class Part_name_set*   partNameSet;
  class Elements_set*    elements_link;
  int                    numNodes;
  Unv_node*              node_list;

 public:

  Nodes_set();
  Nodes_set(Fem_name_set* femNameSet, Part_name_set* partNameSet);
  ~Nodes_set()  { delete[] node_list; }
  void          printSummary();
  void          printNodes();
  int           getNumberOfNodes()    { return numNodes; }
  Unv_node*     getNode(int index)    { return node_list+index; }
  Unv_node*     getNode(Node_ID nodeID);
  void          printNode(int index)  { cout << node_list[index]; }
  void          offsetIDs(Node_ID offset);
  Node_ID       getMaxNodeID();

  friend istream&   operator>>(istream& s, Nodes_set& ns);
  friend void       connect(Nodes_set* ns, Elements_set* es);
};

istream& operator>>(istream& s, Nodes_set& ns);

#endif
```

10. Unv_node

```
/*
 * unv_node.h
 * Unv_node class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef UNV_NODE_H
#define UNV_NODE_H


typedef long Node_ID;


class Unv_node {

  Node_ID nodeID;
  double x;
  double y;
  double z;

 public:

  Node_ID    getID()  { return nodeID; }
  double     getx()   { return x; }
  double     gety()   { return y; }
  double     getz()   { return z; }
  void       offsetID(Node_ID offset)  { nodeID += offset; }

  friend ostream&    operator<<(ostream& s, Unv_node& u);
  friend istream&    operator>>(istream& s, Unv_node& u);

};

ostream& operator<<(ostream& s, Unv_node& u);
istream& operator>>(istream& s, Unv_node& u);


#endif
```

11. Elements_set

```
/*
 * elements_set.h
 * Elements_set class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef ELEMENTS_SET_H
#define ELEMENTS_SET_H

#include "nodes_set.h"
#include "unv_element.h"

#define FEM_NAME_LENGTH 80
#define PART_NAME_LENGTH 80


class Elements_set {

  class Fem_name_set*    femNameSet;
  class Part_name_set*   partNameSet;
  class Nodes_set*       nodes_link;
  int                    numElements;
  Unv_element*           element_list;

 public:

  Elements_set();
  Elements_set(Fem_name_set* femNameSet, Part_name_set* partNameSet);
  ~Elements_set()  { delete[] element_list; }
  void          printSummary();
  void          printElements();
  int           getNumberOfElements()  { return numElements; }
  Unv_element*  getElement(int index)  { return element_list+index; }
  void          offsetIDs(Node_ID nodes_offset, Element_ID elements_offset);
  Element_ID    getMaxElementID();

  friend istream&   operator>>(istream& s, Elements_set& es);
  friend void       connect(Nodes_set* ns, Elements_set* es);

};

istream& operator>>(istream& s, Elements_set& es);

#endif
```

12. Unv_element

```
/*
 * unv_element.h
 * Unv_element class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef UNV_ELEMENT_H
#define UNV_ELEMENT_H

#include "unv_node.h"


typedef long Element_ID;


class Unv_element {

  Element_ID     elementID;
  int            type;
  int            phyPropTableNum;
  int            matPropTableNum;
  int            color;
  int            numNodes;
  Node_ID*       nodes;

 public:

  Unv_element()      { nodes = NULL; }
  ~Unv_element()     { delete[] nodes; }
  Element_ID   getID()                  { return elementID; }
  int          getType()                { return type; }
  int          getMatPropTableNum()     { return matPropTableNum; }
  int          getNumNodes()            { return numNodes; }
  Node_ID      getNthNodeID(int n)      { return nodes[n]; }
  void         offsetID(Element_ID offset)  { elementID += offset; }
  void         offsetNthNodeID(int n, Node_ID offset)  { nodes[n] += offset; }

  friend ostream&    operator<<(ostream& s, Unv_element& u);
  friend istream&    operator>>(istream& s, Unv_element& u);

};

ostream&     operator<<(ostream& s, Unv_element& u);
istream&     operator>>(istream& s, Unv_element& u);


#endif UNV_ELEMENT_H
```

13. Pnf_elem_prop

```
/*
 * pnf_elem_prop.h
 * Pnf_elem_prop class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef PNF_ELEM_PROP_H
#define PNF_ELEM_PROP_H

#include "/fs/blofeld/b/Memcad2/pnf_server/member.h"
#include "/fs/blofeld/b/Memcad2/pnf_server/elementProp.h"


struct Pnf_elem_prop : public Member {

  long                  pnfElemPropID;
  ElementProperties*    pnfElemPropObject;
  /* unvMatPropID is represented as PNF_ID in base class Member */

  Pnf_elem_prop(long inPnfElemPropID, ElementProperties* inPnfElemPropObject,
                long inUnvMatPropID);

};

#endif
```

14. Pnf_elem_prop_list

```
/*
 * pnf_elem_prop_list.h
 * Pnf_elem_prop_list class declaration header file
 * King Chung Yu, Microsystems Technology Laboratories, MIT
 */


#ifndef PNF_ELEM_PROP_LIST_H
#define PNF_ELEM_PROP_LIST_H

#include "/fs/blofeld/b/Memcad2/pnf_server/list.h"
#include "pnf_elem_prop.h"


class Pnf_elem_prop_list : public List {

 public:

  Pnf_elem_prop* getItemWithMP(long inUnvMatPropID)
    { return (Pnf_elem_prop *) getMemberWithID(inUnvMatPropID); }

};

#endif
```

Bibliography

[1] Professor Stephen D. Senturia. *Application-Driven CAD for Microdynamical Systems.* ARPA Semiannual Report No. 4 and FBI Progress Report QR-94-2, MIT, Room 39-567, 77 Massachusetts Avenue, Cambridge, MA 02139, USA, July 1994.

[2] J.R. Gilbert, P.M. Osterberg, R.M. Harris, D.O. Ouma, X. Cai, A. Pfajfer, J. White, and S.D. Senturia. *Implementation of a MEMCAD System for Electrostatic and Mechanical Analysis of Complex Structures from Mask Descriptions.* Proceedings MEMS '93, Ft. Landerdale, Feb 1993, p. 207.

[3] Structural Dynamics Research Corporation, 2000 East Dr., Milford, OH 45150, USA.

[4] Parametric Technology, Waltham, MA, USA.

[5] PATRAN Division/PDA Engineering, 2975 Redhill Avenue, Costa Mesa, CA 92626, USA.

[6] Hibbitt, Karlsson and Sorenson, Inc., 1080 Main Street, Pawtucket, RI 02860, USA.

[7] K. Nabors and J. White. *FastCap: A multipole-accelerated 3-D extraction program.* IEEE Transactions on Computer-Aided Design, 10, 1991, pp 1447-1459.

[8] SDRC Software Products Marketing Division. *I-DEAS Smart View Online Information Software*, 1994.

[9] PATRAN Division/PDA Engineering. *PATRAN Plus User Manual.* 2975 Redhill Avenue, Costa Mesa, CA 92626, USA, September 1989.

[10] Mark H. Lawry. *I-DEAS Master Series Student Guide.* Structural Dynamics Research Corporation, 2000 East Dr., Milford, OH 45150, USA, 1993.

[11] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, June 1993.