NBSIR 87-3531

USING THE AMRF PART MODEL REPORT

Sanford Ressler

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Manufacturing Engineering
Factory Automation Division
Machine Intelligence Group
Gaithersburg, MD 20899

February 1987

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

Using the AMRF Part Model Report

by
Sanford Ressler
National Bureau of Standards
October 1986

Introduction

The AMRF (Automated Manufacturing Research Facility) Part Model [Tua,b,86] is a data format used to represent the information characterizing a part to be manufactured in the AMRF. The Part Model representation includes descriptions of a part's geometry, topology, and features. The importance of the Part Model lies not only in its representational capabilities but also in its function as the primary mechanism through which AMRF applications communicate with the AMRF data base [BARK 86]. To facilitate the use of the Part Model by AMRF applications, a parser has been developed which interprets the Part Model and transforms it into a set of data structures. This paper describes the use of the parser and methods for accessing the data structures produced.

The Part Model

The part model report contains several types of information about the part. There are primarily two types of information in the report: first, the geometry and topology description and second: the functionality of the part (i.e. tolerance information). The geometry provides the specific geometric description in terms of points, curves and surfaces. The topology provides the connectivity of the geometry in terms of a boundary representation which contains shells, faces, loops, edges and vertices. The functionality of the part is described as features which refer to faces for use in specifying tolerances. The tolerances provide information about the accuracy required in the manufacturing processes. The feature and tolerance information is added via an interactive graphics system developed at NBS [CLARK 86]. This information can subsequently be used in other AMRF processing systems such as process planning [BROWN 86], deburring and the inspection workstation.

The format of the part model report is a series of nested blocks with particular data in the blocks. The file is a straight ASCII "flat file" readable by humans. Blocks always begin with the format **/NAME** where **NAME** is the block

No approval or endorsement of Sun Microsystems, Silicon Graphics, DEC or AT&T products by the National Bureau of Standards is implied.

title and end with **/END_NAME.** The details of the part model report are discussed in [HOPP 86].

Parsing

Given a part model report, the user (application programmer) must be able to retrieve specific information from the report and manipulate this information. The first major step is to interpret the report and transform it into data which may be manipulated in the application environment. This step is called parsing the report and is the function of the part model parser. The only function of the parser is to treat the report as a source of data and to create a set of initialized data structures in memory. The use of these data structures is up to the application developer; however, some tools to aid in the manipulation of these structures have been developed.

The parser has been implemented using Lex[LESK] and YACC[JOHN], UNIX™ tools which aid in the development of lexical analysers, compilers and translators. [This document is not intended to describe the usage of Lex or YACC, the interested reader can refer to the above mentioned references and an excelant book [SCHR 85] titled "*Introduction to Compiler Construction with UNIX*" for a complete description of these tools.] The BNF description of the part model in [HOPP 86] was used as the basis for the BNF-like description which forms the majority of the YACC source. The lower level implementation language is C. A user of the part model does not need to understand the internal functioning of the parser or YACC however one does need to understand how to use a parser produced by YACC.

Accessing the Parser

In order to include the parser in an application one must call the function `yyparse()` which is produced by YACC. *This function is the parser and upon* execution it allocates space for the data structures and sets a global pointer to the head of the part. By default the parser will look at `stdin` as the source of input. To make the parser look at an arbitrary file one must open that file and assign `yyin` as the pointer to that file or reopen `stdin` and associate it with the file. At present the parser has only been used with C code, however standard mechanisms for calling C subroutines from other languages will presumably work.

The parser is entirely contained in the function `yyparse()` and this function must be called by the particular application which will use the parser. One simply compiles the application source with a library `pmparser` (Part

UNIX is a trademark of Bell Laboratories.

Model Parser) which contains the parser and all utility functions associated with the parser. For example the following line will compile an application with the parser:

```
cc -o application applicationMain.o applUtil.o applDTFns.o \
 -lpmparser -ll -lyacc
```

All of the object files **(applicationMain.o, applicationUtil.o applDTFns.o)** must include (# **include**) the file **parse.d** which contains the definitions for the data structures created and initialized by the parser. In addition the file containing the **main()** function must include the file **parse.h** and all other object file must include the file **parse.x**. The **parse.h and parse.x** files respectivly contain the global variable and external declarations used in the parser and in application code. The file **applDTFns.o** is application specific Dictionary Traversal routines which traverse the parser data structures and is explained later in this document.

Data Structures

The key to effective use of the parser is understanding the data structures. (See Appendix A for the details of the data structures.) The best way to view these structures is as a relatively direct reflection of the part model report itself. A part in the report is composed of a header, geometry, topology, features and functionality blocks. Likewise the **Part** data structure contains entries for each of these blocks. The data structures (Figure 1) are quite complex and were written with several goals in mind. First they must be the repository of the information obtained in the parsing and second they must be manipulable in a variety of ways. Additionally, the extra complexity of a general list structure (Figure 2) was added to allow the writing of generic routines, which have proved to be a great aid in the development of real applications.

Model Parser) which contains the parser and all utility functions associated with the parser. For example the following line will compile an application with the parser:

```
cc -o application applicationMain.o applUtil.o applDTFns.o \
 -lpmparser -ll -lyacc
```

All of the object files `(applicationMain.o, applicationUtil.o applDTFns.o)` must include (`#include`) the file `parse.d` which contains the definitions for the data structures created and initialized by the parser. In addition the file containing the `main()` function must include the file `parse.h` and all other object file must include the file `parse.x`. The `parse.h and parse.x` files respectivly contain the global variable and external declarations used in the parser and in application code. The file `applDTFns.o` is application specific Dictionary Traversal routines which traverse the parser data structures and is explained later in this document.

Data Structures

The key to effective use of the parser is understanding the data structures. (See Appendix A for the details of the data structures.) The best way to view these structures is as a relatively direct reflection of the part model report itself. A part in the report is composed of a header, geometry, topology, features and functionality blocks. Likewise the `Part` data structure contains entries for each of these blocks. The data structures (Figure 1) are quite complex and were written with several goals in mind. First they must be the repository of the information obtained in the parsing and second they must be manipulable in a variety of ways. Additionally, the extra complexity of a general list structure (Figure 2) was added to allow the writing of generic routines, which have proved to be a great aid in the development of real applications.

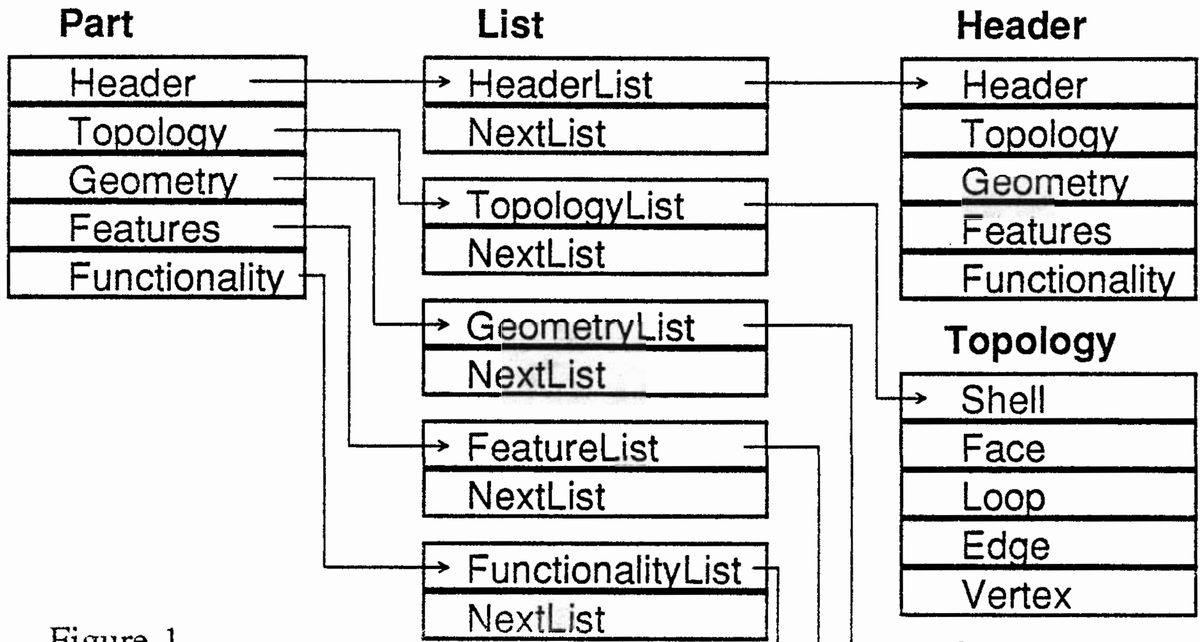
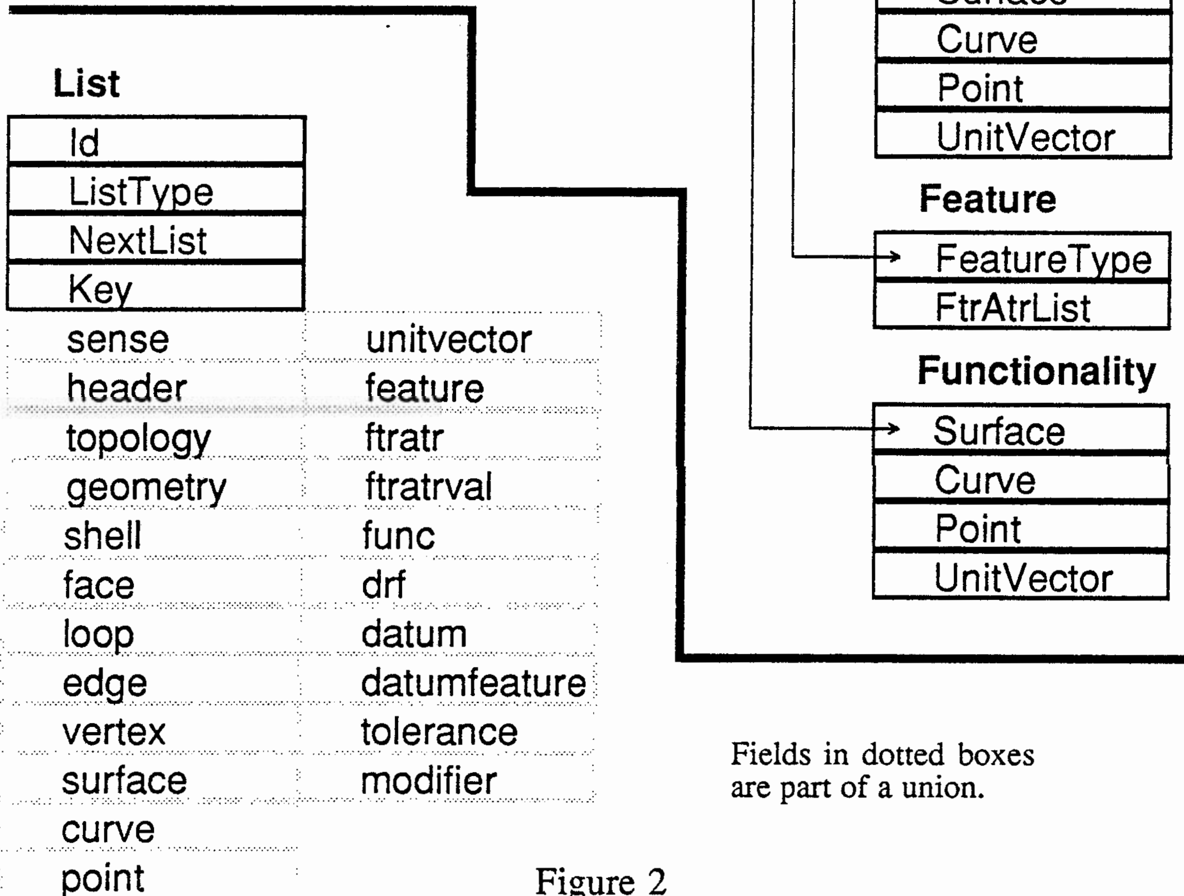


Figure 1



Fields in dotted boxes are part of a union.

Figure 2

When using these data structures in general one must go through one level of indirection to get to the actual data for a particular structure. For example:

If one wanted to get to the **Surface** structure, which is part of the **Geometry** structure, one would use the following C code:

```
surfp =
  Partp->geometryLp->geometryL->surfaceLp->surfaceL;
```

The C code corresponds to the data flow illustrated in figure 3. (Naming conventions: variables ending in Lp are pointers to lists, variables ending in L are List entries.)

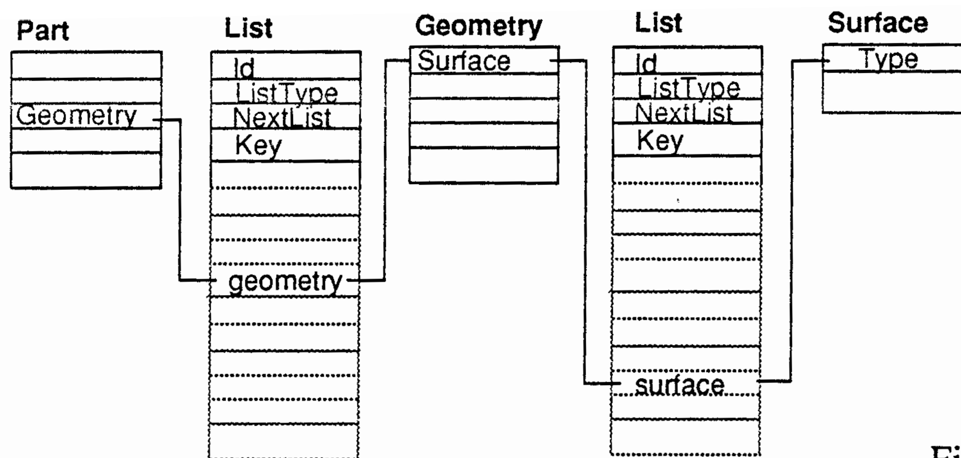


Figure 3

In English this translates as follows: First start at top with the pointer to the part, **Partp**. To get the geometry information we use the geometry list pointer: **geometryLp**. Next, to get the actual geometry information from the union field of the list use: **geometryL**. The geometry structure is also a set of lists so now get the surface list pointer: **surfaceLp**. Finally get the list union field for the surface with: **surfaceL**.

In practice, these long structure references can be annoying, so a set of global variables exists which point to the head of each list. These variables are set by the parser. All of these variables end with the suffix HLp (Header of List pointer). To get to the Surface structure one can simply write:

```
surfp = SurfaceHLp->surfaceL;
```

Utilities

In order to aid in the development of applications which use these data structures, several utilities have been provided. Additionally, an example application (Appendix B) is provided which illustrates the actual usage of the utilities and the data structures. It is in the development of utility functions where the implementation of the general list structure pays off. Generic routines may be developed which can do a variety of manipulations on the structures. The description of a few of these utilities follows.

Most of the data associated with the report contain identifiers which are used as ASCII pointers to data in the report. For example:

> In the following section of a part model report Loop1 consists of four edges. The connections between the loops and the edges are made through the identifiers for the edges. To find the definition of Loop1 start looking down the list of edges which make up Loop1: Edge1, Edge2, Edge3, and Edge4. The list that is searched to find each of these edges is the EdgeId list which is in the edge block: Edge2, Edge3, Edge1, Edge5, Edge10, Edge11 and Edge4.

```
/LOOPS
        Loop1; Edge1 +, Edge2 + , Edge3 -, Edge4 - .
        Loop2; Edge5 +, Edge2 + , Edge10 -, Edge11 - .
/END_LOOPS
/EDGES
        Edge2; Vertex2, Vertex3 ; Curve2 + .
        Edge3; Vertex4; Vertex5 ; Curve3 + .
        Edge1; Vertex1; Vertex2 ; Curve1 + .
        Edge5; Vertex7; Vertex1 ; Curve7 + .
        Edge10; Vertex5; Vertex9 ; Curve9 + .
        Edge11; Vertex3; Vertex10 ; Curve10 + .
        Edge4; Vertex11; Vertex4 ; Curve5 + .
/END_EDGES
```

Please note that the names Edge, Loop, Vertex in the identifiers are not keywords and are used simply for clarity. Foo, froboz, Yomama are all valid identifiers.

The function **FindId** searches for data specified by the an id. **FindId** takes two arguments, both are pointers to lists. The first is the list whose id one wants to find, the second is the list to search through. It returns a pointer to a list which contains the id, or NULL if the id is not found.

```
/* generic list Id searcher */
List *
FindId(idlistp,listp)
     List *idlistp; /* contains the id to look for */
     List *listp;   /* the list to look through */
{
     if (listp == NULL) return(NULL);
     do {
          if (0 == strcmp(idlistp->id,listp->id)) {
               /*then its found */
               return(listp);
          }
          if (listp->nextLp == NULL) {
               /* couldn't find it */
               return(NULL);
          }
     } while (listp = listp->nextLp);
}
```

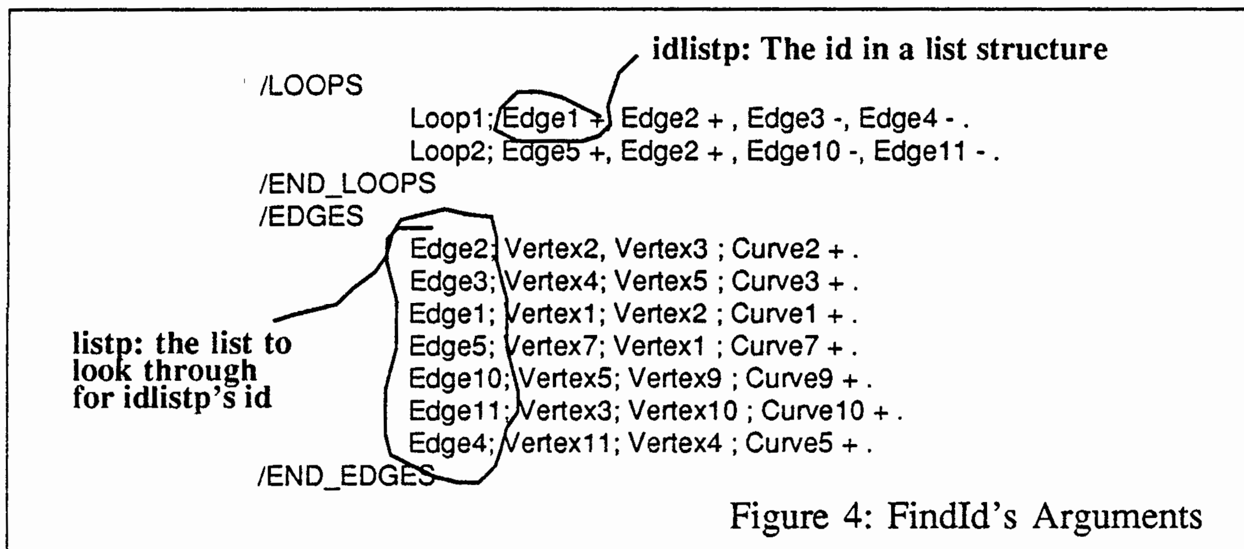


Figure 4: FindId's Arguments

An example function which uses **FindId** is the following function: **DTLoop** (The DT stands for dictionary traversal). This function looks down the loop list looking for a particular edge id.

```
DTLoop(ll)
List *ll;
{
        List *ep;
        List *et;
        List *lh;

        /* get a pointer to the id we want to find */
        ep = ll->loopL->edgeTagLp;
        /*
         * get a pointer to the head of
         * the list we will search: the head of
         * the loop list
         */
        lh = Partp->topologyLp->topologyL->loopLp;
        do {
                et = FindId(ep,lh);
                if (et != NULL){ /* found an id */
                        printf("I found the id: %s\n",et->id);
                        break;
                }
                /* we're at the end */
                if (ep->nextLp == NULL) break;
        } while (ep = ep->nextLp);
}
```

Another useful list function is **TrList** (traverse list). It takes two arguments, the first being a pointer to a list and the second a pointer to a function. **TrList** executes the function upon each node of the list as it traverses down the list.

```
/* general list traversal utility */
TrList(lp,fn)
List *lp;
char *(*fn)(); /* pointer to a function
                    returning pointer to a char
                    (really just a generic pointer) */
{
     List *tp;
     if (lp != NULL) {
          tp = lp;
          do {
               (*fn)(tp); /* execute fn */
               if (tp->nextLp == NULL) break;
          } while (tp = tp->nextLp);
     }
}
```

This has proven quite useful in writing functions which print the contents of the lists. These printing functions are used to create part model reports after modifications to the data structures have been made. The following functions will print the lists of faces contained in the shell data structures.

```
/* print a shell list */
PrShellL(list)
List *list;
{
     printf("\n %s ",list->id);
     TrList(list->shellL->faceIdLp,PrListId);
}

/* print a list id */
PrListId(lp)
List *lp;
{
     if (lp->id != NULL)
          printf(" %s ",lp->id);
     else
          printf(" nil ");

}
```

The AMRF part model report is currently in use in the AMRF. Several utility programs and one major system have been written which use the data structures described. In particular the Geometry Modeling System makes extensive use of the data structures both for reading the part model report,

adding information to the data structures and then producing a new part model report. The parser has been ported to a Sun Microsystems workstations, Silicon Graphics Iris and DEC Vax. The parser is available and is easily ported to any machine running YACC and Lex.

References

[TUa 86] Tu, J., Hopp, T., "Part Geometry Data in the AMRF", An overview of our approach toward modeling and management of part geometry data. NBSIR in preparation.

[TUb 86] TU, J. , Hopp, T., "Geometry Data Modeling in Smalltalk-80" Describes an implementation of the AMRF Part Model. NBSIR in preparation.

[BARKMEYER 86] Barkmeyer, E., Mitchell, M., Mikkilineni, K., Su, S., Lam, H., "An Architecture for Distributed Data Management in Computer Integrated Manufacturing", NBSIR 86-3312 Jan 86.

[CLARK 86] Clark, S., Ressler, S., "Geometry Modeling System User's Guide" Describes functionality and use of the GMS. NBSIR 87-3508 Jan 87.

[BROWN 86] Brown, P., McLean, C. "Interactive Process Planning in the AMRF" winter ASME, Anaheim Ca. Dec 1986.

[LESK] M.E. Lesk, E. Schmidt, "Bell Laboratories, Murray Hill New Jersey, part of UNIX documentation included with all UNIX systems.

[JOHN] Stephen C. Johnson, "Yacc - Yet Another Compiler-Compiler", Bell Laboratories, Murray Hill, New Jersey, part of UNIX documentation included with all UNIX systems.

[SCHR 85] A. Schreiner, H Friedman, Jr. "Introduction to Compiler Construction with UNIX" Prentice Hall, Englewood Cliffs, New Jersey.

[HOPP 86] Hopp, T., "AMRF Database Report Format: Part Model", Detailed specification of report format, syntax (BNF) and semantics. NBSIR in preparation.

Appendix A: Data Structures

/*
Data Structures for Geometry Display and Traversal

Sandy Ressler May 86

The data structures described below are created by the flat file parser which mallocs space for these structures and fills in the data as it parses.

The fundamental data structure is a list (struct list) which contains a union of pointers to most other data types. This in combination with a list type identifier enables the creation of generic list manipulation and id searching functions, which work with all the various data types.
*/

```
/***************** Naming Conventions ****************/
/* Names ending in D are structures which contain real data
 *as opposed to lists.
 * Names ending in Lp are List pointers
 * Names ending L are list entries
 */

/* all of the types of lists, used to enable construction of
 * generic list manipulation routines
 */
enum ListType {  id, tag, sense, topology, geometry, shell, face, loop,
        edge, vertex, surface, curve, point, unitvector,
        tolerance, drf, datum, datumfeature, feature, ftratr,
        ftratrval, func, modifier, header };


/* The #defines below refer to the union entries of a struct list
 * and are merely a convienient short hand notation.
 */

#define sense           list_union.LU_sense
#define topologyL       list_union.tp
#define geometryL       list_union.gp
#define shellL          list_union.sp
#define faceL           list_union.fp
#define loopL           list_union.lp
#define edgeL           list_union.ep
#define vertexL         list_union.vp
#define surfaceL        list_union.surp
#define curveL          list_union.cp
```

Appendix A: Data Structures

```
#define pointL              list_union.pp
#define unitvectorL         list_union.up
#define featureL            list_union.featp
#define ftratrL             list_union.fap
#define ftratrvalL          list_union.favp
#define funcL               list_union.funcp
#define drfL                list_union.drfp
#define datumL              list_union.dp
#define datumfeatureL       list_union.dfp
#define toleranceL          list_union.tolp
#define modifierL           list_union.modp
#define headerL             list_union.hp
#define UniKey              appl_data.key

struct list {
        char id[80];
        enum ListType ltype;
        struct list *nextLp;
        struct {
                        long            key;
        } appl_data;
        union {
                int LU_sense;
                struct header           *hp;
                struct topology         *tp;
                struct geometry         *gp;
                struct shell            *sp;
                struct face             *fp;
                struct loop             *lp;
                struct edge             *ep;
                struct vertex           *vp;
                struct surface          *surp;
                struct curve            *cp;
                struct point            *pp;
                struct unitvector       *up;
                struct feature          *featp;
                struct ftratr           *fap;
                struct ftratrval        *favp;
                struct func             *funcp;
                struct drf              *drfp;
                struct datum            *dp;
                struct datumfeature     *dfp;
                struct tolerance        *tolp;
                struct modifier         *modp;
          } list_union;
};
```

Appendix A: Data Structures

```
typedef struct list List;

struct part {
        struct list             *headerentryLp;
        struct list             *topologyLp;
        struct list             *geometryLp;
        struct list             *featureLp;
        struct list             *funcLp;
};
typedef struct part Part;

enum HeaderType { part_name, drawing_number, designer, gt};

struct string {
        char cs[80];
};

struct header {
        enum HeaderType         type;
        struct string           stringdata;
};
typedef struct header Header;

struct func {
        struct list             *toleranceLp;
        struct list             *drfLp;
        struct list             *datumLp;
};
typedef struct func Func;

struct topology {               /* Get you to the head of the parts data and lists */
        struct list             *shellLp;   /* points to the first shell */
        struct list             *faceLp;  /* point to first face */
        struct list             *loopLp;
        struct list             *edgeLp;
        struct list             *vertexLp;
};
typedef struct topology Topology;

struct geometry {
        struct list             *surfaceLp;
        struct list             *curveLp;
        struct list             *pointLp;
        struct list             *unitvectorLp;
};
typedef struct geometry Geometry;
```

Appendix A: Data Structures

```
struct shell {
        struct list             *faceIdLp;
};
typedef struct shell Shell;

struct face {
        struct list             *loopIdLp;
        struct list             surfTag;
};
typedef struct face Face;

struct loop {
        struct list             *edgeTagLp;
};
typedef struct loop Loop;

struct edge {
        struct list             *vertexIdLp;
        struct list             curveTagD;
};
typedef struct edge Edge;

struct pointId {
        char                    id[20];
};
typedef struct pointId PointId;

struct unitvectorId {
        char                    id[20];
};
typedef struct unitvectorId UnitVectorId;

struct vertex {
        struct pointId          pointIdD;
};
typedef struct vertex Vertex;

/* Used every place a tuple is needed not just for vectors, also
 * for point coordinates and in a variety of other structures.
 */
struct vector {
        float                   x, y, z;
};
typedef struct vector Vector;
```

Appendix A: Data Structures

```
struct SrfPlaneParms {
        struct unitvectorId     normal;
        float                   distance;
};

struct SrfCylinderParms {
        struct pointId          axis_point;
        struct unitvectorId     axis;
        float                   radius;
};

struct SrfConeParms {
        struct pointId          vertex;
        struct unitvectorId     axis;
        float                   vertex_angle_cosine;
};

struct SrfSphereParms {
        struct pointId          center;
        float                   radius;
};

enum SurfaceType { plane, cylinder, cone, sphere};

/* The following #defines refer to the union in struct surface */
#define planeD                  data.planeparms
#define cylinderD               data.cylparms
#define coneD                   data.coneparms
#define sphereD                 data.sphereparms

struct surface {
        enum SurfaceType type;
        union {
                struct SrfPlaneParms            planeparms;
                struct SrfCylinderParms         cylparms;
                struct SrfConeParms             coneparms;
                struct SrfSphereParms           sphereparms;
            } data;
};
typedef struct surface Surface;

enum CurveType { line, circle};

struct CrvLineParms {
        struct pointId          start;
        struct unitvectorId     direction;
```

Appendix A: Data Structures

```
};

struct CrvCircleParms {
        struct pointId          center;/* center of circle */
        struct unitvectorId     axis;  /* normal vector    */
        struct pointId          start; /* point on circle  */
};


/* The following #defines refer to the union in struct curve */
#define lineD                   data.lineparms
#define circleD                 data.circleparms

struct curve {
        enum CurveType type;
        union {
                struct CrvLineParms   lineparms;
                struct CrvCircleParms circleparms;
        } data;
};
typedef struct curve Curve;

struct point {
        struct vector           vector;
            };
typedef struct point Point;

struct unitvector {
        struct vector           vector;
            };
typedef struct unitvector UnitVector;


/* Defines for Features */
#define ftrSD                   data.ftrsimpleparms
#define ftrPD                   data.ftrpatternparms
#define ftrDD                   data.ftrdeburrparms

enum FeatureType { simple, pattern, deburr};
struct FtrSimpleParms {
        List                    *faceLp;
};

struct FtrPatternParms {
        List                    *featureIdLp;
};
```

Appendix A: Data Structures

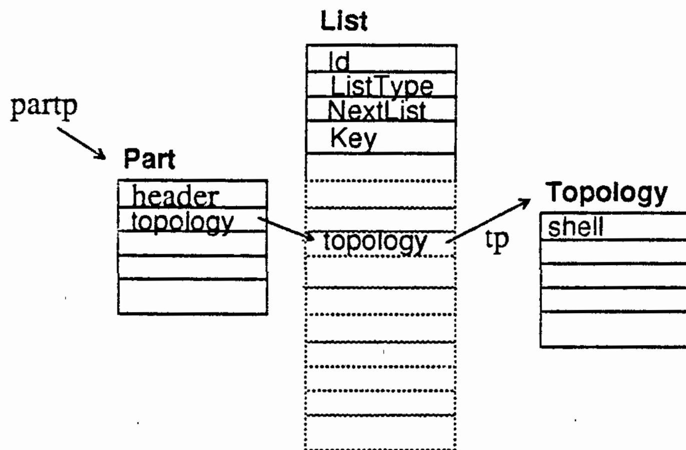
```
struct FtrDeburrParms {
        List                    *faceLp;
        List                    *edgeLp;
};

struct  ftratrval {
        List                    value;
};
typedef struct ftratrval FtrAtrVal;

enum FtrAtrType { faces, edges, features};
struct ftratr {
        enum FtrAtrType         type;
        List                    *ftratrvalLp;
};
typedef struct ftratr FtrAtr;

struct feature {
        enum FeatureType        type;
        struct list             *ftratrLp;
};
typedef struct feature Feature;


/* Datum reference frames */

enum MaterialCondition { lmc, mmc, rfs, nomatcond};

struct drf {
        struct list             *datumfeatureLp; /*the list is only valid up to 3 levels*/
};
typedef struct drf Drf;

struct datumfeature {
        enum MaterialCondition materialcond;
        struct list             *datumIdLp;
};
typedef struct datumfeature DatumFeature;

struct datum {
        struct list             *featureIdLp;
        char                    datumname[20];
};
typedef struct datum Datum;
```

Appendix A: Data Structures

```
/* Tolerance Structures*/

enum Characteristics { size, straightness, flatness, circularity,
                        cylindricity, limit, plus_minus,
                        intrinsic_line_profile, intrinsic_surface_profile,
                        extrinsic_line_profile, extrinsic_surface_profile,
                        angularity, perpendicularity, parallelism,
                        concentricity, circular_runout, total_runout, position};

enum DimensionalInd { r, sr, dia, sdia};

enum ModifierType {material_cond, proj_tol_zone, dimensionalind};

#define matcond             data.mc
#define projtolzone         data.ptz
#define dimind              data.di

struct modifier {
        enum ModifierType type;
        union {
                enum MaterialCondition      mc;
                float                       ptz;
                enum DimensionalInd         di;
        } data;
};
typedef struct modifier Modifier;

struct controlledfeature {
        struct list             *featureIdLp;
        struct list             *modifierLp;
};

struct tolerancedesc {
        struct list             *drfIdLp;
        enum Characteristics tchars;
        float                   limits[2];
};

struct tolerance {
        struct controlledfeature contfeat;
        struct tolerancedesc     toldesc;
};
typedef struct tolerance Tolerance;
```

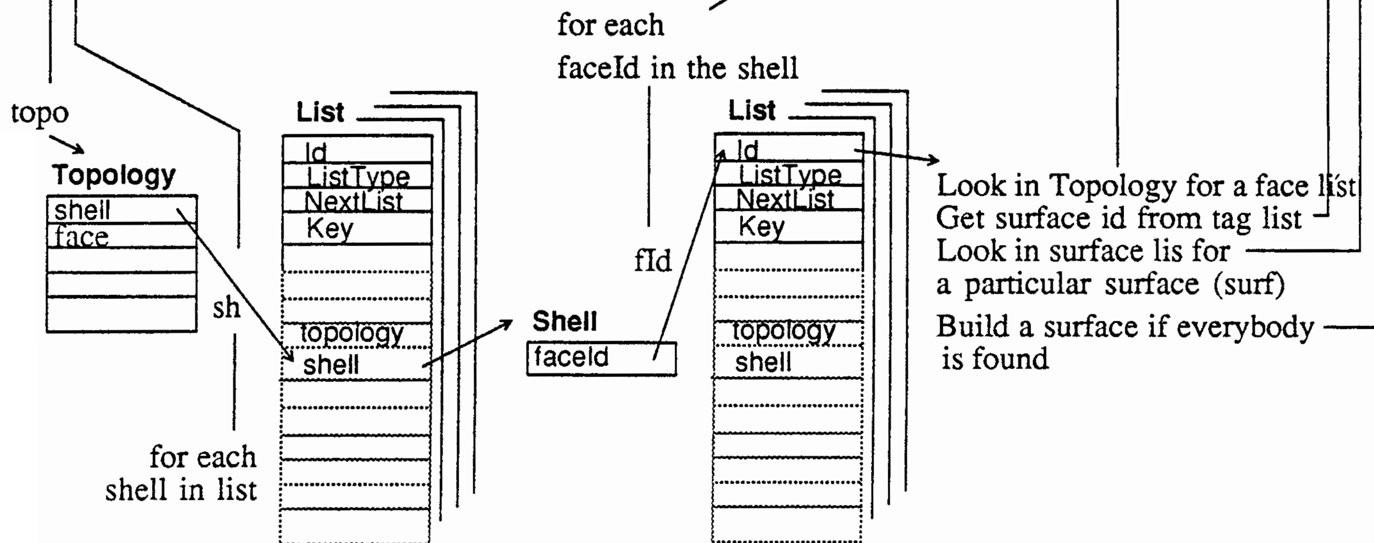
Appendix B: Example Data Structure Usage

```
/*
 * Amrf Topology and Geometry Flat File Format Display routines
 * Sandy Ressler 4/86
 * DT (Dictionary Traversal) Functions
 */

/* The following subroutines traverse the data structures which were created
 * by the part model parser. The purpose of the type of traversal done
 * here is to display a wire frame view of the part which was parsed.
 *
 * The routine BuildSurf does the actual drawing and is not listed
 * here. BuildSurf is called with a variety of arguments and builds up information
 * in static data until it has enough information to display the image.
 */

#include <usercore.h>
#include "math.h"
#include <stdio.h>
#include "parse.d"     <------  Contains structure definition required by parser.
#include "parse.x"     <------  Contains external definitions set in parser.



/* Traverse through the part, by traversing through the shells */

DTPart(partp)
Part *partp;
{
        Topology *tp;
        printf("DTing all shells for part: %s\n",
                partp->headerentryLp->id);

        tp = partp->topologyLp->topologyL;

        DTAllShells(tp);
}
```



Appendix B: Example Data Structure Usage

```
/* Traverse through the all the shells in the topology */
DTAllShells(topo)
Topology *topo;
{
        List *fa;
        List *fId, *sh;
        List surf;
        List *sp;
        sh = topo->shellLp;
        do { /* for each shell */
                fId = sh->shellL->faceIdLp; /* faceId list */
                do { /* for each faceid in the shell */
                        fa = FindId(fId,topo->faceLp);
                        if (fa != NULL) { /* found id */
                                surf = fa->faceL->surfTag;
                                eprintf(4,"surfaceid: %s\n",surf.id);
                                sp = FindId(&surf, SurfaceHLp);
                                if (sp != NULL) {
                                   eprintf(4,"surface found id: %s\n",sp->id);
                                   /* we now know what kind of surface */
                                   BuildSurf(sp);
                                }
                        /* fa is pointer to face list*/
                                DTFace(fa);
                        }
                        if (fId->nextLp == NULL) break;
                } while (fId = fId->nextLp);
                if (sh->nextLp == NULL) break;
        } while (sh = sh->nextLp);
        /*printf("\n");*/
}
```



Appendix B: Example Data Structure Usage

```
/* Traverse through the list of faces. */
/* Primarily to get to the loops, so we can traverse down these loops (DTLoop) */
DTFace(fl)
List *fl;
{
	List *loopId;
	List *loops;
	List *lp;
	/* get the loop list */
	loopId = fl->faceL->loopIdLp;
	if (loopId != NULL) { /* a null loop list is valid, but only
	                       * for spheres
	                       */
		do {  /* for each loopid */
			lp = FindId(loopId, LoopHLp);
			if (lp != NULL) { /* found a loopid in loop list */
				/*printf("found: %s\n",lp->id);*/
				/* look for edgetag in edge list */
				DTLoop(lp);
			}
			if (loopId->nextLp == NULL) break;
		} while (loopId = loopId->nextLp);
	}
}

/* Traverse through a loop list looking for the edges. */

DTLoop(ll)
List *ll;
{
	List *ep;
	List *et;
	ep = ll->loopL->edgeTagLp;
	do {
		et = FindId(ep,EdgeHLp);
		if (et != NULL){ /* found edgetag in edge list*/
			/*printf("found edge: %s\n",et->id);*/
			DTEdge(et);
		}
		/* could polygon here, if we assume that a loop
		 * is equivalent to a polygon
		 */

		if (ep->nextLp == NULL) break;
	} while (ep = ep->nextLp);
}
```

Appendix B: Example Data Structure Usage

```
/* Traverse through an edge list looking for vertices*/
DTEdge(el)
List *el;
{
        List *vid, *vp;
        List crv;
        List *cp;
        crv = el->edgeL->curveTagD;  /* get the curve id */
        cp = FindId(&crv, CurveHLp);  /* look for curve id in crv through the Curve list */
        if (cp != NULL) {
                /* we now know what kind of curve to draw*/
                BuildSurf(cp);
        }
        vid = el->edgeL->vertexIdLp;
        if ((vid == NULL) && (cp->curveL->type == circle)) {
                printf("EDGE (circle) with no vertex\n");
                BuildSurf(cp);
        }
        else {
                do {    /* look for vertexid in vertex list */
                   vp = FindId(vid,VertexHLp);
                   if (vp != NULL) { /* found a vertex*/
                        /* printf("found vertex: %s\n",vp->id);*/
                        DTVertex(vp);
                   }
                   if (vid->nextLp == NULL) break;
                } while (vid = vid->nextLp);
        }
}

/* Traverse through a vertex list looking for points */
DTVertex(vl)
List *vl;
{
        List *pp;
        List vertex;
        /* need the strcpy copy because pointIdD is not a list */
        strcpy(vertex.id,vl->vertexL->pointIdD.id);
        pp = FindId(&vertex,PointHLp);
        if (pp != NULL) {
                BuildSurf(pp);
        }
}
```