A New Approach to CAD/CAM Systems Data Exchange Using Plug-in Technology

Y. A. Chernopyatov[1], W. j. Chung[2,#] and C. M. Lee[2,#]

1 Researcher, Machine Tool Research Center (RRC), Changwon National University, Changwon, Kyongnam, South Korea
2 Professor, School of Mechatronics, Changwon National University, Changwon, Kyongnam, South Korea
Corresponding Author / E-mail: cmlee@sarim.changwon.ac.kr, TEL: +82-55-279-7572, FAX: +82-55-263-5221

KEYWORDS : CAD/CAM, Exchange Standards, Plug-in, Dynamic Link Library

Interoperability has been the problem of CAD/CAM systems. Starting from 1980's, national and international organizations have addressed the issue through development and release of standards for the exchange of geometric and nongeometric design data. To CAD/CAM vendors, the task of interpreting and implementing these standards falls into their products. This task is a balancing action between users' needs, available development resources, and the technical specifications of standards. This paper explores an area of CAD/CAM systems development, particularly the implementation of the effective exchange files translators'. A new approach is introduced, which proposes to enclose all the translation operations concerning each exchange format to a separate DLL, thus making a "plug-in." Then, this plug-in could be used together with the CAD/CAM system or with specialized translation software. This approach allows to create new translators rapidly and to gain the reliable, high-efficiency, and reusable program code. The second part of the paper concerns the possible problems of translators' development. These difficulties often come from the exchange standards' misunderstanding or ambiguity in standards. All examples come from the authors' practice experiences of dealing with CAD/CAM systems.

Manuscript received: January 16, 2004 / Accepted: January 4, 2005

1. Introduction

Nowadays a lot of CAD/CAM systems are widespread in the world market. To be sure, each of the CAD/CAM software developer tries to make its system reliable, convenient and effective. As manufacturing and construction industries become more global, there is growing demand to exchange the digital definition of products between different organizations as the product moves from design through manufacture to long-term support. One of the problems, which should be solved by every developer, is the format for the internal data representation. Obviously, this format should be compact and precise. The operational time of input and output is not so critical as before, but still important. When such an internal model or format is designed, developers often use some tactics or special solutions, and the internal formats usually are the subject of patents. For an example, we can consider the DWG binary format, which is introduced by AutoDesk Inc., for the AutoCAD®. Though AutoCAD® can easily use the open DXF text format for importing or exporting drawings, it uses the DWG format as the main format for storing drawings. Also we should note that when the new version releases, the internal format may change. The characteristic of such changes could be different. In turn, this renders the problem of compatibility and succession. Here is only small number of problems in format area with which developer might be confronted.

The next question is the compatibility between the newly designed and existing systems. In general, all systems could be divided into three main categories – "heavy," "medium" and "light" systems. The "heavy" systems are the most powerful ones, and do large assemblies, specialized design, engineering calculations, CNC program generations and so on. Since they are usually based on unit principle, user may purchase only necessary units. "Medium" systems usually possess the truncated abilities of "heavy" systems. In addition, the "light" systems are usually intended only for drawing and plotting so that they are the cheapest.

Usually an enterprise exploits only a few workstations with "heavy" systems to perform complicated and time-consuming tasks, and the basic works are submitted to average and low-class workstations, equipped with "medium" and "light" CAD/CAM systems. Such approach allows the considerable gain in the sense of costs of design and manufacturing. However, in turn, it requires CAD/CAM systems compatibility on the level of exchange formats. The usage of exchange files instead of the internal formats is caused by several reasons. First, internal formats are the private information of a developer company as a rule. It seems to be simple to develop interfaces between systems as required, but this rapidly produces a large number of translators, which are expensive for maintaining. Four systems require 12 interfaces - for 10 systems, this number grows to 90 and the problem rapidly spirals out of control, since change in one system may require changes to be made and tested to all the interfaces of the system.

The solution is to use the one common exchange format, but unfortunately this idea is unreal, simply because designers do not yet have any common standards. In these conditions, CAD/CAM

Copyright (c) 2005 by KSPE

engineers should be able to "understand" at least general industrial data exchange standards, which can be confirmed by the users' needs as shown in Fig.1 [1].

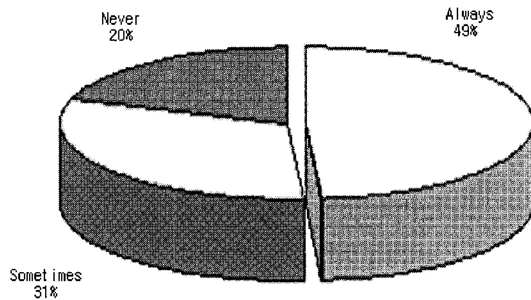

Fig. 1 Percentage of data exchange involving translators between different CAD systems

2. Background

At present there are several popular exchange formats, widely used for data sharing between CAD/CAM systems. The DXF format was developed by AutoDesk, Inc. for AutoCAD® software and initially was just a graphics format, with limited its capabilities. While the first versions of this format contained only plane elements, 3D elements and solids were added in the later versions [2].

The STEP standard [3] defines an integrated information model, which supports the multiple views of product data for different applications. All STEP information models are defined by using the EXPRESS data definition language (Part 11). The EXPRESS language has been developed in the mid-1980's to provide the necessary information modeling constructions to support the complex relationships of product information. The use of the EXPRESS language is supported by a range of software tools, which can assist the process of modeling and developing translators. Typical implementations use STEP to combine the information on the shape and other characteristics of individual parts with assembly structures to form a single integrated representation of a complex assembly or product. This information is gathered from a range of application systems, then consolidated into a STEP file, which can be transferred to other companies and unloaded into their corresponding systems. The advantage of combining this data is that it guarantees the consistency for information deliveries, and avoids the administrative cost of ensuring the consistency of data between multiple systems.

IGES (Initial Graphics Exchange Specification), neutral data format, describes product design and manufacturing information created and stored in CAD/CAM systems. Its purpose is to aid the exchange of geometry, annotation, and structure information between dissimilar systems. It has been initially developed and accepted as an USA national standard. IGES Version 1, published in 1980, included only basic capabilities for drawings created with wire frame geometry. The standard has been consistently updated. Since Version 5.1, the B-rep solids support has been added [4]. Though IGES is not expected to grow considerably after Version 6, it will be maintained as long as there is market demand for it, or it will be completely replaced with STEP standard.

VDAFS has been published as a German national standard in 1986. A number of automotive manufacturers and suppliers throughout Europe use the standard to exchange surface data used in the design of automotive tooling and components such as body parts, injection molded parts, seats, panels, and so on. One of key features of the standard is the simplicity of translator's implementation. Such well-known automobile companies, as BMW, VW, Opel, Porsche, Daimler-Benz as well as machine tools manufacturers Hella, Bosch have worked over this project.

Once again we can notice that such numbers of standards mean that CAD/CAM system should be able to read and write at least the most popular of them. We will examine the question of the data exchange implementation in the CAD/CAM systems. There are two ways to implement the translation capability on the system. The first way is to implement the reading/writing procedure directly on the program code. We call this way "internal" translator. For example, the import procedure could be represented as shown in Fig. 2.

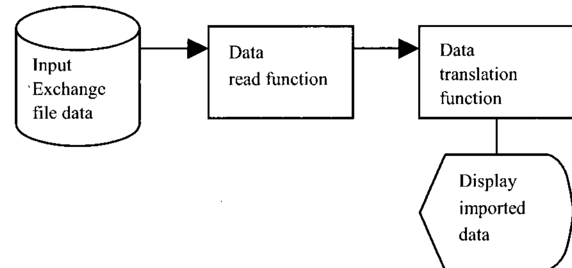

Fig. 2 Internal translator (import procedure)

What kinds of advantages and disadvantages does the internal translator have? Main advantages are – the simplicity of procedure implementation, the highest speed of data exchange, the minimum memory and time consumption. This approach is still used in several systems, such as ZCad or Camax, even though the disadvantages can be reduced to zero benefit. First of all, the system becomes too rigid to make any change or correction fast. For example, if you find an error in reading or writing procedure, you should recompile the whole system or translation unit. This, in turn, leads to numerous service packs release and highly degrades the responsibility. Besides, the system reliability strongly decreases. Some system could even crash down without saving any data only due to an error in a data exchange algorithm.

The solution for such kind of problem often lies in developing the "standalone" translator, which could be implemented as DLL (Dynamic Link Library) or even separate EXE (executable) unit. In this case translation procedure (import) is illustrated in Fig. 3.

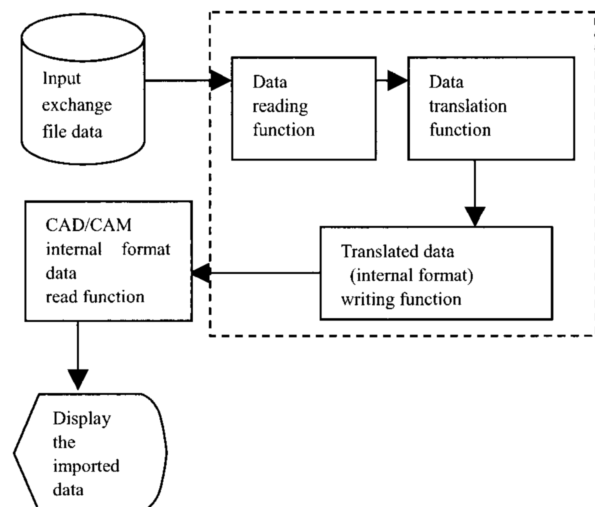

Fig. 3 Standalone translator (import procedure)

The dashed area is the standalone translator working stage. As a rule, the standalone translator is implemented as a separate executable application and could be used as either translator or exchange files generator. When this translator is used in a CAD/CAM system, the entire process can be passed "transparently" to a designer. Using this approach, we gain the ability of separate usage (the re-usage of program code) of the once written application and more system

stability. This way of translation is used, for example, for some Russian CAD/CAM's such as Gemma3D (GEMMA Ltd.) [5] and Kredo (NIC ASK) systems.

But this method may not be considered as the best one. The separate executable application requires writing the intermediate file in some format, that is, internal format, and this file then should be loaded by the CAD. This results in the disk space or time consumption. The T-FLEX CAD/CAM (Top Systems, Ltd.) system utilizes this system.

3. The Plug-in Technology in Data Translation

Contents In this paper we will consider the ability to develop translator in a fast and reliable way, and to provide good code reusability together with less software modifications. The way is to implement the translators as plug-ins. The plug-in for Windows platform is a DLL (Dynamic Link Library), explicitly linked with the application. The plug-in has some predefined number of functions that application may call. For the case of translator, we can illustrate the following minimum of predefined functions, as shown in Fig. 4.

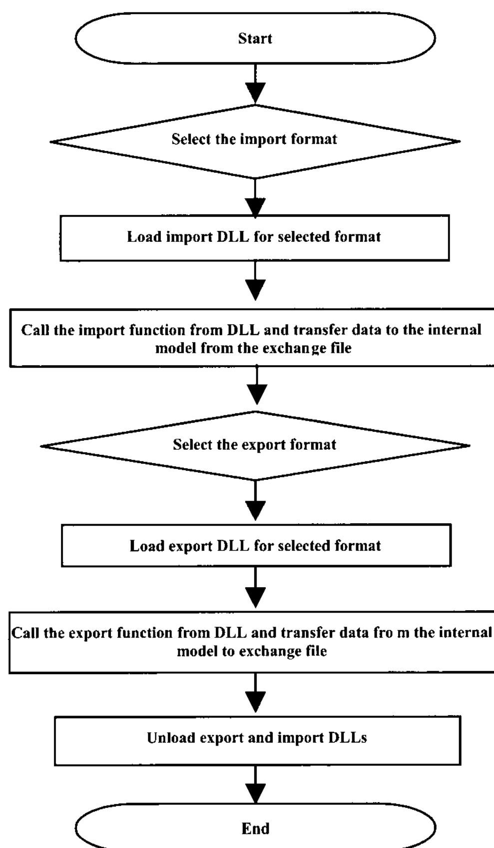

Fig. 4 Plug-in translator used as CAD/CAM system plug-in (complete cycle)

Import function should read the input file in exchange format and translate it to the internal representation (model). Export function should translate the necessary entities from the internal representation (model) to a selected exchange format and then write the output file. Setup function is optional and could be used to adjust some internal settings of DLL, such as output tolerance or the way of solid model exporting (explode them to bounded surfaces or export as solids, for example).

Since we use the same naming system both in caller application and in plug-in, the application can simply load the library and get the function entry point via calling to GetProcAddress function or its analog in languages, other than C++. Application then may call this function using its address to perform the required action as shown in Fig. 4.

What advantages does the plug-in approach have? First of all, the application has "loosely coupled" interface with the plug-in. It means that we may modify or change the plug-in DLL as we need without any modification in the program code of caller application. Update is as easy as file replacement. The only thing we need to do is to keep the predefined function names untouched both in application and in plug-in. We can have even more abilities, such as granting the user's right to access to the internal kernel (model) and thus write it's own plug-in.

One more important difference between standalone translator and the proposed method is the possibility of direct passing the pointer to the internal model between translator and calling application. In other words, translator immediately fills the model with entities, while standalone translator should create the temporary file, which later would be read by the system.

As far as plug-in can be explicitly linked to application, it is advantageous over implicit linking. For example, if the DLL is not found or failed, during runtime the application can display an error message and still continue. It could be easily seen that using traditional method the number of translators for all necessary formats interfacing will be $2n$, where n is the number of present formats (in the worst case of implementing the scheme of "one-to-one" translating). At the same time the number of plug-ins will be the same as that of standards.

The idea of this approach is not simply putting all methods into DLL and uses them instead of API calls. The most important benefit that we can gain from this approach is the remarkable code re-usability, which greatly improves the developing process. In other words, the plug-in, once made for the CAD/CAM system with modeling kernel "XYZ" could then be used in another system with the same "XYZ" kernel, or even as the plug-in for specialized executable translator (see Fig. 5).

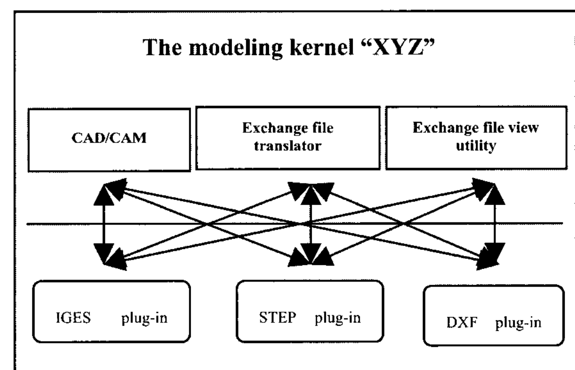

Fig. 5 The re-usability of plug-in's in different applications, based on the same modeler kernel

It can be noticed that there are two major issues in data exchange process. The first is the increasing efficiency of data exchange application, the second is the increasing the efficiency of data exchange itself (for example, in case of systems based on different mathematics kernels). We should note that the proposed approach deals with the first issue, while the second is the subject for further researches. Indeed, the translation accuracy strongly depends on the mathematics kernel used for the internal geometry structure representation. It can be easily seen if we consider the solid and surface geometry in the different exchange formats as an example. If a target format does not have the solid geometric items, but only surface, all solids should be reduced to the surfaces (using projection operation). Depending on the mathematics kernel, this operation may cause the accuracy lost. Thus the developer is completely responsible for the whole internal conversion implementation.

4. The Plug-in Technology in Data Translation

We will show how it is possible to program the necessary functions for plug-in implementation. Our plug-in will be able to read and write the files of an imaginary exchanging format. It is assumed that we are developing the plug-in's for the abstract solid modeler named the "XYZ." The kernel is built upon the object–oriented concept. The internal model structure is a list of entities (for simplicity) and a whole model enwrapped in the one class, named the "CEntityManager."

It is well known that programmers should keep the following rule while writing the DLL. That is, the memory allocated in the DLL's address space should be freed in the same DLL's address space. In other words, if we pass the pointer from the caller application to DLL function and DLL allocates the memory inside this function, this memory must be released in some other DLL function not inside the caller application.

Concerning our sample application, we see that the CEntityManager belongs to the caller application, but the import of entities from an input file to the model takes place in DLL's function "Import." Hence we can write the set of functions, which will fulfill our needs specified above:

```
int Import(LPCTSTR szFileName, CEntityManager *&pEM);
int Export(LPCTSTR szFileName, const CEntityManager *&pEM);
int MemFree(LPCTSTR szFileName, CEntityManager *&pEM);
```

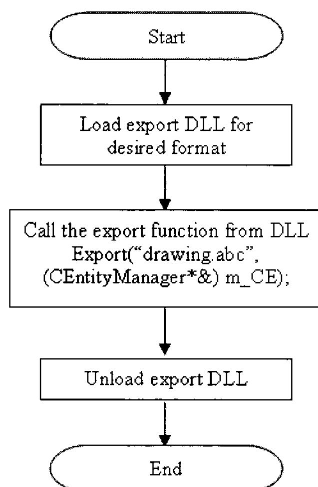

Fig. 6 Block-diagram of the export procedure using plug-in DLL

Now we will discuss the Import and Export functions more thoroughly. The Import directly affects the state of CEntityManager instance (allocates memory and adds new entities to the model), while Export function does not affect the one as shown in Fig. 6. In other words, the state of CEntityManager will change after calling Import function. It means that the better way is to create a new instance of CEntityManager class and pass it to the Import function, which will fill it with imported entities, and then append or copy its contents to the main caller model. After this, it is safe to call the MemFree function to clean the temporary model instance, as shown in Fig.7.

Below we would like to provide the detailed description of sample application implementation to illustrate the proposed approach. For simplicity we removed excessive portion of code, and left only essentials. All examples are given in Visual C++.

Let's assume that our internal model represents a simple list of pointers to geometric entities, which are inherited from one abstract class named "GeometricItem" (of course, in real industries applications it is more complex). This class consists of the basic set of properties and interface methods. Thus, the class for this model may contain: the items list, add, get, remove item functions and other service procedures and functions.

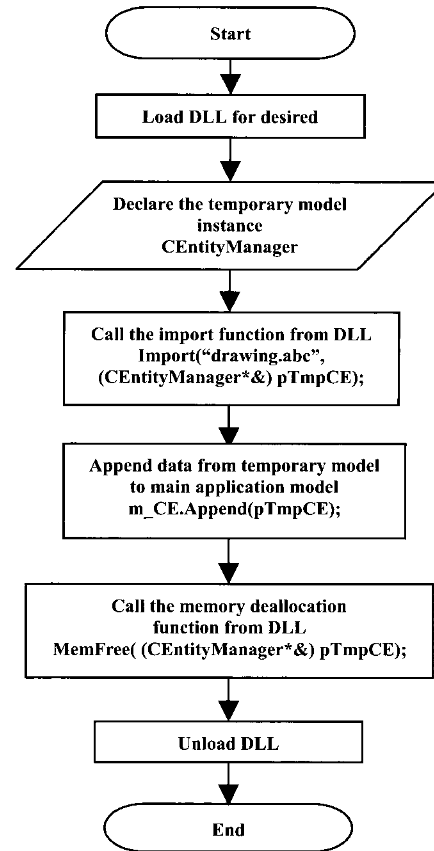

Fig. 7 Block-diagram of import procedure using plug-in DLL

The above described model is the core part of caller application. The pointer to this model should be passed to the Import or Export functions as described before. Besides the model itself, we will need some support structures. Using them caller application can exchange some service data with plug-in. Here is a simple class for this purpose:

```
struct CDLLInfoStruct
{
        CString DLLName; // name of plugin DLL,
        CString DLLExt;     // extension(s) of exchange files
proceed by this plug-in
        CString DLLExtra; // extra text information (for example,
plug-in description)
};
```

Now we see how caller application can process the data exchange. First of all, we should define the forward declarations of prototypes of pointers to functions, used in plug-ins. Since the plug-ins are explicitly linked to main application, it will use the pointers to have an address of function in plug-in DLL.

```
// obtain information about plug-in
typedef int
 (*LPFNDLLGETINFO)(CDLLInfoStruct&);
// import function
typedef int
 (*LPFNDLLIMPORT)(LPCSTR ,  CEntityManager *&);
// export function
typedef int
 (*LPFNDLLEXPORT)(LPCSTR , CEntityManager *&);
// cleanup function
typedef int
(*LPFNDLLFREEENTITYMANAGER)(CEntityManager *&);
```

The last one is very important. If the memory is allocated in DLL during operating, it should be free in the same DLL. This is the feature of DLL usage that we should obey.

Suppose that in our main application the name of model variable (of type "pointer to the CEntityManager") is m_pModel. Here goes the sample implementation of import procedure in a main application. In plug-in in the briefest way, we assume that all necessary checking and initializing (including model initializing) is already done:

```
BOOL CMainApplication::OnImport()
{
LPFNDLLIMPORT lpfnDllImport;
HANDLE hImportDll;
//load the import plug-in DLL
hImportDll=AfxLoadLibrary(sImportPluginName);
if (!hImportDll) return FALSE;
//get pointer to plug-in import function by its predefined name "Import"
lpfnDllImport=(LPFNDLLIMPORT)GetProcAddress(hImportDll,"Import");
if (!lpfnDllImport) return FALSE;
//call the plug-in import function
if(!lpfnDllImport(sFromFileName, *&m_pModel) ) return FALSE;

return TRUE;
}
```

The import function in plug-in should be able to read the exchange file, convert each entity to the internal representation and fill up the provided entities list. Let's assume that we are dealing with the plug-in, responsible for the STEP exchange format processing. In plug-in interface section we should declare the import and export functions as "exportable." It is convenient to hide (encapsulate) all methods of working with particular exchange standard in one class, and use its interface methods in exported functions. Detailed description of STEP processing is out of this paper topic, thus we are concentrating on implementation nuances.

The function of obtaining the information from DLL is straightforward:

```
__declspec (dllexport) int GetDLLInfo(CDLLInfoStruct& inf)
{
        AFX_MANAGE_STATE(AfxGetStaticModuleState());

        inf.DLLName="StepDLL";
        inf.DLLExt="stp;step";
        inf.DLLExtra="STEP ISO-10303 files";
        return (1);
}
```

The input file has the name of STEP file to be translated into internal representation.

```
__declspec (dllexport) int Import((LPCTSTR szFileName, CEntityManager *&pEM)
{
        AFX_MANAGE_STATE(AfxGetStaticModuleState());
        // all works for reading, compiling and translating STEP file are encapsulated in special CStepFile class
CStepFile sf;
        // call the internal procedures of STEP file processing
        int bRes = sf.InternalImport(szFileName, pEM);
// return the result of operation
return bRes;
}
```

By analyzing the method described above, we can easily see that it is more efficient to write one plug-in. It implements only import and export functions for single format, compared with any other approach.

As some kind of addendum we would like to explore a subject of subtle errors appearing due to misunderstanding of standard or inaccurate reading/writing procedure implementation. As an example, we will use IGES and STEP exchange formats.

Our first example is STEP exchange format. As the STEP format described using EXPRESS object-oriented language, all entities stay in "parent-descendant" relations. Thus ISO 10303 specification [3] allows two slightly different versions of exchange file generation. The first version is writing the descendant entity containing all parent entities, moreover each descendant entity in this series is written with its only set of parameters, distinguished from parent's parameters. See the example 1:

Example1.

```
#286=(BOUNDED_SURFACE()B_SPLINE_SURFACE(2,2,((#2
16,#223,#230,#237,#244,#251,#258,#265,#272),(#217,#224,#231,#2
38,#245,#252,#259,#266,#273),(#218,#225,#232,#239,#246,#253,#2
60,#267,#274),(#219,#226,#233,#240,#247,#254,#261,#268,#275),(#
220,#227,#234,#241,#248,#255,#262,#269,#276),(#221,#228,#235,#
242,#249,#256,#263,#270,#277),(#222,#229,#236,#243,#250,#257,#
264,#271,#278)),.UNSPECIFIED.,.F.,.T.,.U.)((3,2,2,3),(3,2,2,2,3),(-
2.030,-
0.676,0.676,2.030),(0.0,1.570,3.141,4.712,6.283185307179600), .UN
SPECIFIED.)GEOMETRIC_REPRESENTATION_ITEM()RATION
AL_B_SPLINE_SURFACE(((1.0,0.707,1.0,0.707,1.0,0.707,1.0,0.70
7,1.0),(0.779,0.551,0.779,0.551,0.779,0.551,0.779,0.551,0.779),(1.0,0
.707,1.0,0.707,1.0,0.707,1.0,0.707,1.0),(0.779,0.551,0.779,0.551,0.77
9,0.551,0.779,0.551,0.779),(1.0,0.707,1.0,0.707,1.0,0.707,1.0,0.707,1
.0),(0.779,0.551,0.779,0.551,0.779,0.551,0.779,0.551,0.779),(1.0,0.70
7,1.0,0.707,1.0,0.707,1.0,0.707,1.0)))REPRESENTATION_ITEM('')
SURFACE());
```

This example shows the entity "RATIONAL_B_SPLINE_SURFACE," which inherits the whole set of parent entities as shown in Fig. 8.

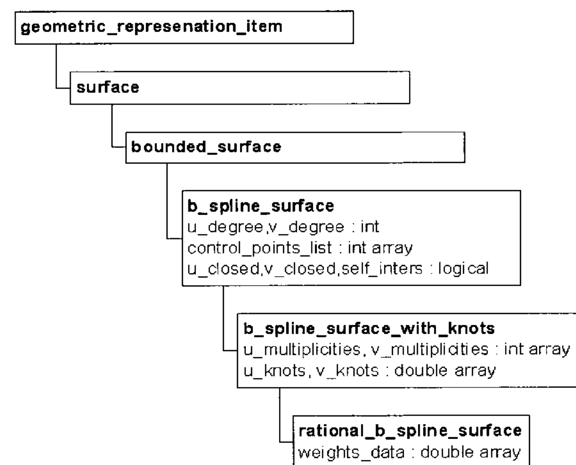

Fig. 8 Hierarchy of surfaces in STEP format (fragment)

Each entity adds new data members and inherits the parent's data. For "B_SPLINE_SURFACE" it is the u and v degree, control points array (as references to existing points entities, denoted by "#" sign), surface type, flags of closeness and self-intersection. In turn, "B_SPLINE_SURFACE_WITH_KNOTS" contains multipliers array, knot array, and knot types descriptor. The "RATIONAL_B_SPLINE_SURFACE" will add the weights array. Thus using this variant to read/write one descendant "RATIONAL_B_SPLINE_SURFACE" entity first, we should deal with the whole family of parent entities. The second variant implies that each descendant entity should contain the whole set of parameters. It inherits from its parents plus its own ones. See example 2:

Example 2.

#142=B_SPLINE_SURFACE_WITH_KNOTS('*SUR2151',5,1,((#143,#144),(#145,#146),(#147,#148),(#149,#150),(#151,#152),(#153,#154),(#155,#156),(#157,#158),(#159,#160),(#161,#162),(#163,#164),(#165,#166),(#167,#168),(#169,#170),(#171,#172),(#173,#174)),.UNSPECIFIED.,.U.,.U.,.U.,(6,5,5,6),(2,2),(0.000,1.000,2.000,3.000),(0.000,1.000),.UNSPECIFIED.);

Unlike example 1, we see that only one entity description is present - B_SPLINE_SURFACE_WITH_KNOTS, and whole parameters are given either. Considering the above samples, it is obvious that a developer should implement different procedures to read/write the same entities.

The next example is the IGES exchange format. Sometimes developers do not test their translators thoroughly, so that the files exported by one system could not be read by another system. Rational B-spline curve (entity number 126) and rational B-spline surface (entity number 128) [4] are good illustrations for above-mentioned problems. These entities have their last parameters to be set to the minimum and maximum parameters of entity function (t for curve and u, v for surface correspondingly). Indeed this specifies the entity trimming - $\lambda(u)$ for curve and $\sigma(u, v)$ for surface. While proceeding the export, some systems perform "reparameterization" of splines (especially when spline is trimmed with 3-dimensional vertices) in such a manner that the knots vectors remain the same, but the minimum and maximum parameters exceed the bounds set by knot vectors. The result is a failure of consequent import process of another system.

Next unobvious question is the trailing zeroes in floating-point numbers. Some exporting translators write the zero after the floating-point dot, while some omit them. But in turn import systems could not "understand" the numbers with dot but without trailing zero.

Another hidden catch is the terminating section of IGES file. The terminating section contains service information about starting section line number, directory entry line number, etc. Some systems such as ZCad just omit some parameters of this section. Though importing system could calculate such kind of information by scanning a file itself, a number of CAD/CAM's just do not perform the import at all if they would find an error in a terminating section.

5. Conclusion

There is a lot of problem with which developers might face during the CAD/CAM implementation. One of these problems is the problem of data interface with other systems. Unless one universal standard appears, each system should support the modern industrial exchange standards. This paper considers the ways of implementing such translators and introduces the new approach based on the plug-in technology. First of all, this approach helps the efficiency of translators increase. Then, we improve the quality of the translators implemented on its base, because they are more stable and reliable comparing to other methods. Also, we gain high reusability, which is very important in the process of developing.

In the second part of the article, the some practical questions of data exchange were considered. Problems of STEP and IGES data transfer were examined, and practical recommendations were given. These recommendations are based upon the analysis of wide variety of modern CAD/CAM systems. Following these conclusions may help to plenty of possible errors during the development of translators.

ACKNOWLEDGEMENT

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Machine Tool Research Center at Changwon National University.

REFERENCES

1. "Data Exchange - Is Sharing Really A Pain," CAD SPAGHETTI, October 2001 Issue, 2001.
2. Jump, D. N., "AutoCAD Programming. 2nd edition," McGraw-Hill Companies, 1991.
3. ISO 10303-203 Industrial automation systems and integration - Product data representation and exchange - Part 203 (IS): Application protocol: Configuration controlled design
4. The Initial Graphics Exchange Specification (IGES) Version 5.1 – IGES/PDES Organization, USA, 1991.
5. Vermel, V., Nikolaev, P., "GeMMa-3D for Windows: increasing productivity," SAPR I GRAPHICA, Vol. 4, 2000.
6. Course, D. L., "Developer roundtable: STEP vs. IGES," CADALYST Magazine, October Issue, 2001.
7. Gubich, L., Ivanets, G., Hamets, N., "Design and manufacturing of molds in the integrated system SolidEdge - GeMMa-3D," SAPR I GRAPHICA, Vol. 11, 2000.
8. Mason, H., "ISO 10303-STEP. A key standard for the global market," ISO Bulletin, January, pp. 8-13, 2002.
9. Stevens, T., "Viewpoint -- Now You're Speaking My Language," Industry Week.com, Columns - Publication Date 7.3.2001, 2001.