ENGINEERING DATABASE AS A MEDIUM FOR TRANSLATION

Hisham Assal and Charles Eastman*

ABSTRACT:

In this paper we describe the translation facilities as a component of EDM-2 database. We introduce a new approach to translation that is different from the traditional translators in databases and the standard neutral file approach. First, we define design views, which are different from database views in that they allow manipulation of data, and they represent the same object or information in different formats. Second, we define object structures that capture the different representations of objects and define the relationships among them. The two main relationships here are the specialization lattice and the composition lattice. Third, we describe the basic steps of the translation process and generalize the common ones. We provide facilities for developing translators that take advantage of this generalization. We present an example of the most common representations in engineering design (IGES and DXF) to illustrate the various steps and structures in our model.

INTRODUCTION:

In the process of designing an engineering product, a wide variety of information types are generated, modified or deleted until a final product is reached. This information involves both data and relations. Computer applications perform design tasks that generate product information, access a subset of existing product information and possibly modify it. Each application has its internal representation of the information it requires and assumptions about its relationships. For example, CAD systems deal with the geometry of the product and its spatial relationships, while analysis programs such as daylighting or thermal performance deal with properties of materials of that product. There is a growing number of applications that perform the same task in different manners or using different models of analysis. Such programs are being developed independently of each other and most applications use data structures optimized for their operations. As a result, each application program has its own view of the product characterized by its own representation. During design, there is a need to integrate many tools to assist the designer or provide useful information about the current state of the design. In an integrated design environment, these different representations need a way of transferring information back and forth among each

*Dept. of Architecture and Center for Design and Computation, University of California, Los Angeles, 90024-1476. E-mail: {hassal, chuck}@gsaup.ucla.edu. WWW: http://www.gsaup.ucla.edu

other. There are two approaches to deal with this issue. One approach is to develop a universal representation that can accommodate all types of applications and require that all components of the design system conform to that form. The second approach is to accept the variety of representations of different applications and provide means for transferring information from one view to the other. The universal representation approach requires that all entities be predefined and given a specific reference to be used by all modules that may need it later. The issues of modularity and extensibility of CAD systems make this solution not attractive for several reasons. As technology develops, new applications are introduced in many areas affecting design. There is no guarantee that new applications will not require different formats for their internal operations. New methods emerge that may require a different structure of information or the representation of relationships that are not supported by the universal representation. We believe that the anticipation of every possible need for future applications is not feasible. In addition, using a representation that has more structure and information than needed can be unnecessary burden on simpler applications. During design, too much structure of information can hinder the effective use of the system. We conclude that the ability to move among different representations within the same environment is an important factor in integrating design environments.

In the second approach, the process of transferring information among views is translation. There have been efforts to develop models for translation that can be categorized in three approaches: direct translation (also known as the pairwise approach), intermediate file format, and database view generation.

EXISTING APPROACHES TO TRANSLATION:

The pairwise translation approach develops a new pair of translators for each new application. Each translator manipulates objects in a specific way based on assumptions about the use of those objects in the target application. Any change in the assumptions requires re-writing of some parts of the code. Adding a new application requires N translators, where N is the number of communicating applications.

The intermediate (neutral) file format approach develops a general representation of all the anticipated entity types. This representation may include some dependencies among different types. The checking of these dependencies is the responsibility of each application in the modules reading or writing from or to the neutral file format. The neutral file format also makes assumptions about its entities both for their representation and the methods of handling them. Examples of neutral file formats and standards include IGES, DXF, and STEP. The advantage of using a neutral file format over pairwise translation is that each application has to communicate with a central format instead of all other applications, thus reducing the number of required translators considerably, especially when the number of applications grows.

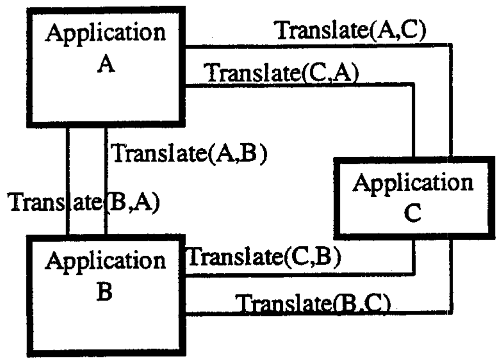

Figure 1. Pairwise Translation Model

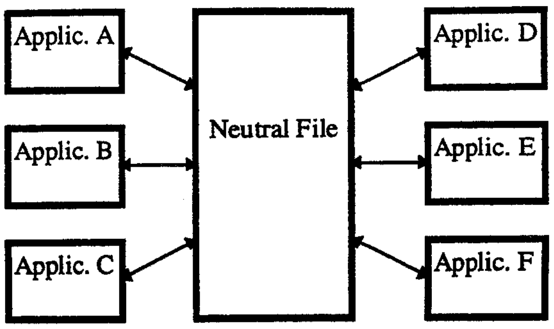

Figure 2. Neutral File Translation Model

The third approach to translation is the database view generation. Traditional databases can provide some help for the translation process by defining a view for each added application, which groups the subset of information it uses. The DBMS can provide tools to generate one view from another based on predefined queries. However, this solution does not solve all the problem. The expected format and the internal assumptions about the data and its relationships still have to be provided. Also, there is a difference between database views in its traditional sense and design views in design databases, as discussed below.

DATABASE VIEWS AND DESIGN VIEWS

In relational database systems, a view is a subset of the conceptual database or an abstraction of part of the conceptual database [Ullman, 88]. Dealing with views allows access to its data, either stored or derived. It does not, however, allow changing this data directly in the view. Changes have to be made to the canonical database, representing all integrity relations in the canonical model, and then have the view regenerated. This is because of the ambiguity of the effect of changing the view on the attributes that are not represented in that view. Also, all integrity relations involving the changed variables and incorporated in the core update program must be respected. In object-oriented systems a view of a class type can be considered a subclass of this type and methods can be used to define the values in the subclass in relation to their inherited attributes of the super class [Monk 94]. (This approach transforms the instance from one view to the other; to use a previous view it has to be transformed back!) Efforts to define mechanisms for deriving views in object-oriented systems include [Abiteboul & Bonner 91].

A design view is a representation of (a subset of) design objects that satisfy the data needs of a specific design task. Design views are used by members of a design team or cooperating design applications that can produce new data or change existing values. Therefore, design views must have equal stature and allow modification of data within the view. The change in one view must be reflected into all the views of the same object. For example, if the glazing area of a window is changed by a daylighting application, this change should be visible to the acoustics application and vice versa. Thus, current implementations of database views are incompatible with the concept of design views.

Problems with Existing Approaches

There are several problems in all of the above approaches to translation. First, the representation of input and output formats has certain assumptions about the well-formedness of its entities. These assumptions are hidden, but any application has to be aware of it. For example, the DXF representation of an arc assumes that if the two end points are coincident, then the entity is reduced to a point; while in IGES this assumption is that the entity, under the same condition, becomes a full circle. Second problem is that the mapping of entities from one representation to another is predetermined. It is possible that one entity type can map to more than one type in another representation. Object types can be arranged in a hierarchy that defines the relationships across them and conditions for type subsumption. The subsumption relationship among two types (A and B), denoted by A > B, states that A subsumes B if all the characteristics of A can be represented by B. This arrangement is useful in type conversion among applications as well as within the same application. A more detailed discussion of the subsumption relationship among representations is presented in [Stouffs et al., 1995]. For example, it may be desirable to represent a simple arc in DXF as a spline in IGES. This is acceptable in principle, since the spline representation is subsumed by the arc representation. However, traditional translators predetermine the type conversion of all types. Third problem is that the type conversion process does not make use of the structural relationships among types. These relationships include the well-formedness conditions of entity types. Type conversion of entities within the same representation is one of the translation tasks and structural relations among types can determine the possibility of converting between two types. In addition, there is the problem of isomorphism between types in different representations. Isomorphism of types can be defined in terms of the subsumption relationship in the following manner. Two types, A and B, are isomorphic if and only if (A > B) and (B > A). This means that all characteristics of type A can be defined in terms of attributes of B and all characteristics of B can be defined in terms of attributes of A. If types in two different representations are not isomorphic, then exact translation from the subsuming representation is allowed but only in specific conditions. If no subsumption relationship holds, then approximation has to be applied. There is usually more than one method of approximation for each type conversion process. For example, a spline can be approximated to a number of straight line segements (AutoCAD igesout translator). This may lead to accumulated errors, if the approximated type is used in a subsequent conversion. It may also lead to loss of information in translating in the opposite direction. For example, the approximated spline in the above example loses all information about curvature and control points and cannot be translated back to its original type.

In this paper we propose an approach to translation that deals with the above problems. In this approach different representations can be stored in a unified database that provides means for translating from one representation to the other. Representations are not fixed and new ones can be added as design proceeds.

EDM APPROACH TO TRANSLATION

The Engineering Data Model (EDM) is a data model and a database implementation (EDM-2) developed to provide a platform for both representing design information and supporting translation between different application views. [Eastman et. al. 91]. The design database is the repository of all information pertaining to the product being designed, for re-use by all applications that need it. In order to make translation a task of the database, EDM defines some structures that capture the relationships of the object types and provides mechanisms for managing the integrity of the views when they are updated, possibly in an arbitrary order. It also provides a mechanism for deriving dependent data and generating and maintaining equivalent views.

Primitives:

The primitives of EDM-2 language include domains, constraints and maps. Domain is a type or a set of allowable values. EDM-2 is a strongly-typed language which requires all variables to be of declared types. Constraints define relations among object attributes or among objects and are defined globally with type reference only. To invoke a constraint for a specific instance, a constraint call is used to identify the required instance reference and the actual input parameters. A state value is associated with each constraint to manage the integrity of the model. Maps are specialization of constraints that have an output set and its implementation allows the changing of the database variables and schema. As with constraints, map calls are used to identify the object reference and the actual input/output sets. The inherited state value is used to indicate the validity of the current values that were generated by the map call. If a variable in the input set of a map is changed, the state value is set to blank, marking the output set variables as invalid. Constraints and maps can be implemented in any available programming language and then linked to the database as object methods. EDM-2 provides a management system for checking constraints and invoking maps, which maintains the state value of both and checks the parameter types.

Structures:

The main structure in EDM is the Design Entity (DE). There is a class definition for DE which carries all its attributes and constraints. DEs can be arranged in a specialization lattice, to allow inheritance of higher level DE attributes. Composite objects are defined using a 'composition' structure which identifies a set of parts that make up the composite object and a set of constraints that define how the parts are put together. The checking and maintenance of the integrity constraints may follow a precedence order, i.e. certain constraints have to be checked before others. To allow this precedence order EDM provides an 'accumulation' structure which has two sets of constraints: preconditions and postconditions and is associated with either a DE or a composition. Maps are defined in reference to a DE and can be applied to any instance of that DE.

TRANSLATION WITHIN THE EDM DATABASE:

Translation is mapping the set of objects in one representation into the set of objects in a different representation. This mapping takes place on various levels of the representation. As described above, the 'map' construct in EDM-2 provides a method for mapping between two attribute type sets: input set and output set. In translation, the main construct for defining identity is the DE. A high level DE carries all views of the identical object. In some cases equivalence is only supported if certain rules or conditions hold. These conditions are defined by constraints. If a map has precondition constraints that must be satisfied before execution, then an accumulation is used to specify those preconditions and control the execution of the map. A DE may be defined as a composite object and a map translates the DE and all the parts defined in the associated composition. In this case the scope of the map is all the DEs in the composition.

Here, We foucus on three types of translation based on the object definition above.

1. The first type is *view translation*. This type generates one view of the object from another view of the same object. Different applications have different views of the same object and these views are used in the presenting the object and manipulating it. For example, the circular arc is represented in one view by three points, center, start point and end point; in another view it is represented by a center point, radius and two angles. It is obvious that one view can easily be generated from the other.
2. The second type of translation is *object type conversion*. Object types can be defined in terms of other object types by applying some restrictions to a more general type. For example, the rational B-spline curve can represent a variety of curve types including the straight line and conic sections. The general representation of the B-spline is restricted by some constraints to represent the conic section curves and by other constraints to represent segments of straight

lines. Now, given a straight line that is represented by its two end points, it may be desirable to convert its representation into a rational B-spline representation, e.g. to modify it as such so that it describes a more sophisticated shape. Type relationships can be defined in general terms in a lattice for the common geometric entities.

3. The third type of translation is *composite object translation*. A composite object is an object that is made up of a set of simpler objects. For example, a polygon is made up of a set of lines. A composite object can be represented as a set of enumerated components and a set of relations (constraints) describe how they are put together. In this case relations specify that the lines are two connected and non intersecting. Alternatively, the polygon can be described as a single entity defined by a sequence of vertices. The translation of composite objects can take place in one of three ways: Single object to composition, Composition to single object and Composition to composition. In the third case a decomposition operation is required in order to generate a set of components that can be used to create the new composition.

Processes of Translation.

We identify the steps of the translation operation and their language commands in EDM as follows:

1. Definition of input format: data structures, integrity rules, and type hierarchies. These definitions match the relations that already exist in the given format. A model is defined in EDM constructs to represent this format isomorphicaly as shown in the example below.
2. Definition of output format: data structures, integrity rules, and type hierarchies. The required output may have a set of relations that map onto relations in the database or it may need to be incorporated within it. A formal way of defining relationships for both input and output is provided by EDM.constructs.
3. Definition of the generic objects that accommodate both representations as subclasses. This is defined in the generic entities section of the example.
4. Definition of maps as methods of converting between two representations. A map is defined for each entity to be translated. It should be noted that more than one map can be defined for the same entity to convert it in a number of ways. A mapcall is defined to specify the actual arguments of the entity class to translate.
5. Definition of mapping paths between input and output format. A variety of maps can be defined for the same type conversion requirement based on the type lattices. One-to-one mapping produces equal types on both ends. Other types of mapping result in different conditions in each direction. The latter may suffer some loss of data in going from a complex type to a simpler one. The recovery requires defining more conditions under which the simple type can convert to the more complex one.

6. The structure that manages the mapping operation is accumulation. An accumulation can have a set of precondition constraints that define the data state requirements for the mapping to take place. If all the constraints in the precondition set are satisfied, the map can execute and produce instances of the output set type entities.
7. Application of mapping of a path of choice. Objects of the same type need not translate through the same path. Each instance (or set of instances) of a type may choose a different path according to the intended use in the target type (format). The application of these choices can be made through the design process. The mapcall specifies the entity paths and the map to be applied to them. The 'EVAL' command operates on a map call and is passed a set of DEs to be translated. It checks the construct that holds the mapcall. If it is a DE, it executes directly. If it is an accumulation, it checks the state of the precondition constraints and execute the map only if they are satisfied.

THE EXAMPLE

Geometry is a basic element in engineering design. There are several formats for geometry in different systems. We choose IGES and DXF among those formats for our example to illustrate the process of translation since these are the two most common formats in CAD applications. We apply the processes of translation as described above to simple objects in both representations. More complex objects can be translated in the same manner, but the map implementation will be more sophisticated. Note that input set and output set are interchangeable and are determined by the map construct. First, we define the classes of entities that exist in IGES in EDM-2 model as shown in figure 3.

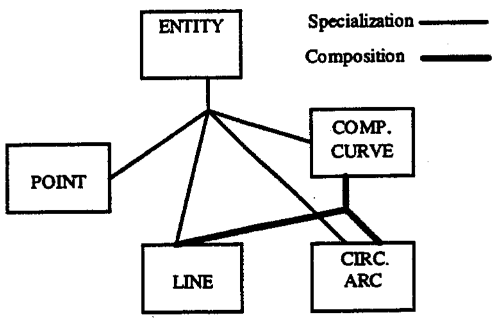

Figure 3. Example IGES Entities

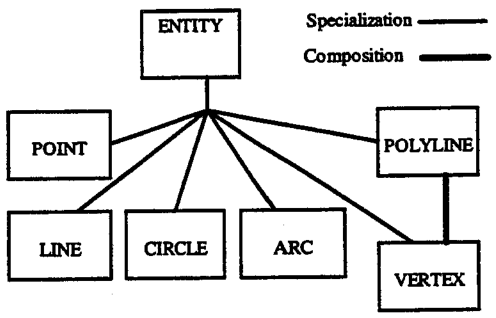

Figure 4. Example DXF Entities

Then we define DXF entity classes as shown in figure 4.. Unlike IGES, there is no general entity class. The common characteristics, such as layer, thickness, elevation, etc are defined for each entity. Here we define a general entity class for DXF entities to make all the attributes explicit. The EDM-2 language commands for all the steps are shown in appendix A.

The next step is defining generic classes as generalizations of both IGES and DXF classes for each entity (Figure 5). These generic classes can be expanded later to accomodate more formats as the need arises. Instances of the defined entities are created during design operations or by some applications.

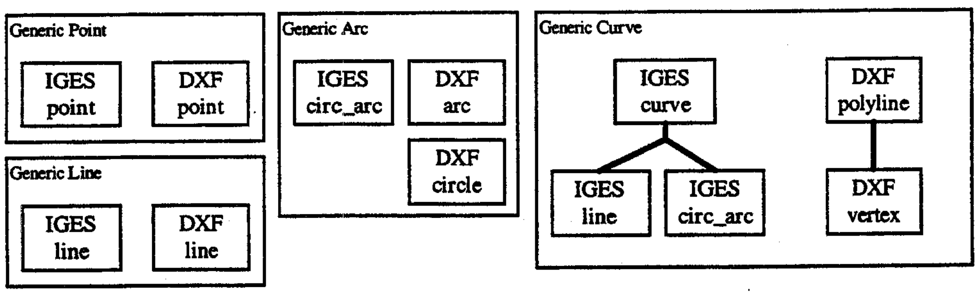

Figure 5. Generalization of IGES and DXF Classes

At this point the newly created generic line instance has an iges line entity but has no dxf line entity. The maps can now be defined to translate from IGES to DXF.

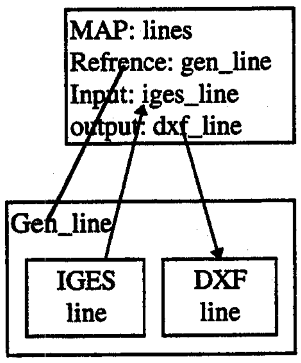

Figure 6. Map Definition

Map calls operate on instances of DEs. There is an EDM-2 command to execute a map call with reference to a specified set of DE instances of its reference object. For example, to translate a set of iges_line instances to dxf_line instances within the same gen_line instance we invoke the following command:

```
EVAL d_lines ON DE (KEYNAME=(gl1, gl2, gl3)) OF gen_line;
```

where gl1, gl2, gl3 are keynames of the chosen instances. To select all instances, we can use the * character as a wild card. This command takes each instance of gen_line in the specified set and applies the map call d_lines to it. The map implementation handles the input checking and dereferencing through the 'Constraint and Map Programming Interface CMPI' of EDM-2, which provides access to the internal structure of the EDM-2 database and allows manipulation of its values and schema within dynamically linked C functions. The result of this command is the creation of an instance of dxf_line in each of the specified gen_line instances as translation of the iges_line instance. If a dxf_line instance already exists in any of the specified gen_line instances, the values of its attributes are modified to reflect the current translation.

CONCLUSION

We have presented the EDM database management system as a medium for performing data translation operations in a design database. The structure of EDM and its integrity management facilities provide a means for establishing relationships among different representations and methods for mapping from one representation to the other. The example shows simple maps for translating between simple entities in IGES and DXF formats, which are two of the most common formats for exchanging graphic information. The same approach can be used for complex objects found in high level applications. We believe these capabilities can be scaled to a production level system.

Note

This work has been supported by the National Science Foundation, grant No. IRI 9319982.

REFERENCES

Abeitboul, Serge and Bonner, Anthony. *Objects and Views.* Proceedings of the ACM SIGMOD International Conference on Management of Data. Denver, Colorado, May, 1991.

Adam, Nabil R. & Bhargava, Bharat K. *Advanced Database Systems.* Lecture Notes in Computer Science Series, Springer Verlag, Berlin, 1993.

Eastman, Charles M., Bond, Alan H. and Chase, Scott C. *A Formal Approach for Product Model Information.* Research in Engineering Design. 2:4, pp. 65-80. 1991.

Monk, S.R. *View Definition in an Object-Oriented Database.* Information and Software Technology, Vol. 36, No. 9, pp. 549-554, September, 1994.

Stemple, David and Sheard, Tim. *Construction and Calculus of Types for Database Systems.* In Bancilhon, F. and Buneman, P. (Ed.) Advances in Database Programming Languages. ACM press, New York, 1990.

Stone, Harold. *Discrete Mathematical Structures and Their Applications.* Science Research Associates, Inc. Palo Alto, California, 1973.

Stouffs, R; Krishnamurti, R., Eastman, C. & Assal, H. *Non-Standard Representation of Solid Models.* 1995 (To be presented at CADD Future 1995, Singapore.).

Tremblay, J.P. and Manohar, R. *Discrete Mathematical Structures with Applications to Computer Science*. McGraw Hill Computer Science Series, McGraw Hill Inc., 1975.

Ullman, Jeffrey D. *Principles of Database and Knowledge-Base Systems.* Vol. 1. Computer Science Press Inc. Rockville, Maryland, 1988.

APPENDIX A

EDM-2 LANGUAGE COMMANDS FOR THE EXAMPLE

Create IGES class entities

```
CREATE DE iges_entity KEYNAME
        ATTR(e_type:num, para:num,
            struct:num, pattern:num,
            e_level:num, e_view:num,
            transform:name, label:name,
            stat:num, seq:num,
            weight:num, color:num,
            para_count:num, form:num)
        DESC "A generic class for all
            entities in IGES";

CREATE DE iges_point KEYNAME
        ATTR(e:iges_entity, x:coord,
            y:coord, z:coord)
        DESC "A simple point";

CREATE DE iges_line KEYNAME
        ATTR(e:iges_entity,
            start_p:iges_point,
            end_p:iges_point)
        DESC "A straight line segment";

CREATE DE iges_circ_arc KEYNAME
        ATTR(e:iges_entity,
            center_p:iges_point,
            start_p:iges_point,
            end_p:iges_point)
        DESC "Circular arc-includes
            circle";

CREATE DE iges_comp_curve KEYNAME
        ATTR(e:iges_entity,
            n_segments:num);
        DESC "Composite Curve, lines &
            arcs";

CREATE COMP to_curve
        TARGET iges_comp_curve
        PART(iges_line,iges_circ_arc);
```

Create DXF class entities

```
CREATE DE dxf_entity KEYNAME
        ATTR(e_type:num, layer:num,
            elevation:coord,
            thickness:coord,
            linetype:num, color:name)
        DESC "Generic class for all
            entities in DXF, carries
            common properties to all
            entities ";

CREATE DE dxf_point KEYNAME
        ATTR(e:dxf_entity, x:coord,
            y:coord, z:coord)
        DESC "A simple point";

CREATE DE dxf_line KEYNAME
        ATTR(e:dxf_entity,
            start_p:dxf_point,
            end_p:dxf_point)
        DESC "A straight line segment";

CREATE DE dxf_arc KEYNAME
        ATTR(e:dxf_entity,
            center_p:dxf_point,
            raduis:coord, start_p:angle,
            end_p:angle)
        DESC "Circular arc - no
            circle";

CREATE DE dxf_circle KEYNAME
        ATTR(e:dxf_entity,
            center_p:dxf_point,
            radius:coord)
        DESC "Full circle";

CREATE DE dxf_vertex KEYNAME
        ATTR(e:dxf_entity,
            location:dxf_point,
            start_width:coord,
            end_width:coord,
            bulge:angle, flag:num,
            tangent_dir:coord)
        DESC "A polyline vertex";

CREATE DE dxf_polyline KEYNAME
        ATTR(e:dxf_entity,
            elevation:dxf_point,
            flag:num, start_width:coord,
            end_width:coord, m_vert:num,
            n_vert:num,m_dnst:num,
            n_dnst:num, pl_type:num);

CREATE COMP to_polyline
        TARGET dxf_polyline
        PART(dxf_vertex);
```

Create the generic class entities to generalize both IGES and DXF

```
CREATE DE gen_point KEYNAME
        ATTR(i_point:iges_point,
            d_point:dxf_point)
        DESC "Generic point";

CREATE DE gen_line KEYNAME
        ATTR(i_line:iges_line,
            d_line:dxf_line)
        DESC "Generic straight line
            segment";

CREATE DE gen_arc KEYNAME
        ATTR(i_arc:iges_circ_arc,
            d_arc:dxf_arc,
            d_circle:dxf_circle)
        DESC "Generic arc segment.";

CREATE DE gen_curve KEYNAME
        ATTR(i_curve:iges_comp_curve,
            d_curve:dxf_polyline)
        DESC "A generic composite
            curve-polyline.";

CREATE CONSTRAINT not_coincident
    (point, point)
        IMPL $CONSTRAINTS/coincd.so
        DESC "Checks that two points
            are not coincident";

CREATE CONSTRAINT coincident (point,
    point)
        IMPL $CONSTRAINTS/coincd.so
        DESC "Checks that two points
            are coincident";

CREATE CCALL open_arc
        CONSTRAINT not_coincident
            (start_p, end_p):V
        DESC "Checks if the arc is
            open";

CREATE CCALL closed_arc
        CONSTRAINT coincident (start_p,
            end_p):V
        DESC "Checks if the arc is
            closed";
```

Example of create iges_line instance and assign it to a gen_line instance

```
:v_11 = INSERT INTO DE iges_line
        KEYNAME=11 (start_p.x=12.3,
            start_p.y=14.5,
            start_p.z=0.0, end_p.x=28.9,
            end_p.y=37.6, end_p.z=45.3);

INSERT INTO DE gen_line KEYNAME=g11
        (i_line=:v_11);
```

Create Maps to translate lines, arcs and curves.

```
CREATE MAP lines
        (iges_line)
        RETURN (dxf_line)
        IMPL $MAP_METHODS/lines.so
        DESC "Make a dxf line out of an
            iges line. (trivial)";

CREATE MAPCALL d_lines
        MAP lines
        (i_line)
        RETURN (d_line)
        REF gen_line
        DESC "Requires the actual
            objects as parameters.";

CREATE MAP arcs
        (iges_circ_arc)
        RETURN (dxf_arc, dxf_circle)
        IMPL $MAP_METHODS/arcs.so
        DESC "Make dxf arc or circle
            out of iges ciruclar arc.";

CREATE MAPCALL circ_arcs
        MAP arcs
        (i_arc)
        RETURN (d_arc, d_circle)
        REF gen_arc
        DESC "Requires the actual
            objects as parameters.";

CREATE MAPCALL arc_circs
        MAP arcs
        (i_arc)
        RETURN (d_arc, d_circle)
        DESC "Requires the actual
            objects as parameters.";

CREATE MAP curves
        (iges_comp_curve)
        RETURN (dxf_polyline)
        IMPL $MAP_METHODS/curves.so
        DESC "Make dxf polyline out of
            iges composite curve.";

CREATE MAPCALL curve_to_polyline
        MAP curves
        (i_curve)
        RETURN (d_curve)
        REF gen_curve
        DESC "Requires the actual
            objects as parameters.";
```