Extracting STEP Geometry and Topology from a Solid Modeler: Parasolid-to-STEP

Thomas R. Kramer

May 2, 1991

NISTIR 4577

Funding for the work described in this paper was provided to Catholic University by the National Institute of Standards and Technology under cooperative agreement Number 70NANB9H0923.

Disclaimer

Trademarks

Acknowledgments

Funding for the work described in this paper was provided to Catholic University by the National Institute of Standards and Technology under cooperative agreement Number 70NANB9H0923.

The National PDES Testbed is funded by the Computer-aided Acquisition and Logistic Support (CALS) Office of the Department of Defense.

Table of Contents

List of Figures

1 Introduction

1.1 What is Parasolid-to-STEP

Parasolid-to-STEP is a software system for converting a Parasolid model of a single solid object stored in a Parasolid transmit file to a STEP physical file containing topology and geometry for a boundary representation of the object. Parasolid is a commercial solid modeling software system made available by Shape Data, Limited. STEP is the emerging international Standard for the Exchange of Product Model Data.

In addition to a Parasolid transmit file, the Parasolid-to-STEP system takes as input an Express schema file defining STEP geometry and topology entities. As will be discussed in section 8.4, however, the system is not schema independent. It requires that the input schema file define a specific set of STEP geometry and topology entities.

Parasolid-to-STEP will be frequently be abbreviated as “PTS” in this paper.

1.2 Express

Data descriptions are handled in STEP by defining an information model in the Express data modeling language [Schenck90] for each type of data that is needed. Once an information model is in place, data representing a specific product may be put into a computer or on paper according to the rules for mapping Express to a STEP physical file [Altemueller88a, Altemueller88b]. The data section of a STEP physical file normally consists wholly of instances of data entities of the sort made available by the Express model. Express information models can also be used for automatically prescribing the format of a database in a computerized database system or the format of information about a product in the active memory of a computer.

1.3 NIST Tools for STEP

Tools for dealing with STEP data have been developed at NIST. An overview of many of these tools is given in “An Introduction to the NIST PDES Toolkit” [Clark90a]. This section discusses only those tools that are used in the Parasolid-to-STEP system.

A system called Fed-X [Clark90b, Clark90c] takes as input an information model written as an Express schema file and builds an Express Working Form for the schema (in the C programming language) in the internal memory of a computer. This Working Form can be used by other C programs. The Fed-X system also includes a library of C functions for manipulating Express Working Forms.

A system called STEPparse [Clark90d, Clark90e] takes as input a STEP physical file representing a product and builds a STEP Working Form for the product (in the C programming language) in the internal memory of a computer. STEPparse uses the Fed-X system to parse and represent an information model; an Express Working Form is required in order to define what sorts of data may be in a STEP Working Form. STEPparse also includes a library of functions for manipulating STEP Working Forms.

1.4 Parasolid

Parasolid can be used either as a library of C functions which may be included in a C program, or interactively. In Parasolid-to-STEP it is used as a library. Parasolid saves representations of the shapes of objects in files called “transmit” files. When the Parasolid system is being used, it can read a transmit file to bring a part model into active computer memory, and it can write out a transmit file for a part that is already in active memory.

The conceptual view of the representation of a solid object in Parasolid is that of a hierarchical arrangement of entities. There is a top-level entity which is the body. It is composed of entities called shells, which are composed of entities, and so forth. Each entity has an identifier called a “tag” in Parasolid. Parasolid keeps track of the connections in the hierarchy, so that given the tag of one entity, the user can easily retrieve the tags of all other entities which have a specific type of connection to it. Below the lowest level in the hierarchy of entities is a lot of information represented as real numbers or arrays of real numbers that are not entities and have no tags. The axis of a circle, for example is represented as an anonymous array, not as a tagged entity which could have been called a unit_vector.

Additional details of Parasolid are presented in the next chapter.

1.5 Entities

What Express calls an “entity,” we are calling an “entity type,” and we are using “entity” to mean what would be called an “entity instance” in STEP terms. The usage in this paper is intended to be more in keeping with the common meanings of these terms. Parasolid uses “entity” in the same way as this paper. The term Entity (starting with a capital E), however, refers to the construct of that name used in Fed-X.

1.6 Audience

This paper is aimed at persons who want to use the Parasolid-to-STEP system for converting Parasolid models to STEP boundary representation models. It is also aimed at persons who want to understand how the system works; a reader who does not care how it works can stop reading at the end of chapter 4, “Using Parasolid-to-STEP.” This paper does not, by itself, provide sufficient information for programmers who want to rewrite the software or use parts of it in other systems. However, the software was written with the intent of making it easy to read and is heavily commented, so that a C programmer familiar with boundary representations could probably deal with it after reading this paper.

1.7 Acknowledgment

The Parasolid-to-STEP system could not have been built in its current form without the software written at NIST by Stephen N. Clark for handling Express information models and STEP product data.

2 Shape Models

2.1 Introduction

The shape of a solid object may be expressed in many ways amenable to automated data processing. These include: exact boundary representation, facetted boundary representation, wire frame, constructive solid geometry, octree, and form features. Information models for several of these have been readied for inclusion in STEP. PTS uses only boundary representations. STEP entity types used in PTS (listed in Section 3.3) are drawn from the STEP topology and geometry schemas only.

Within boundary representations, a distinction may be made between systems which constrain the user to build data representations only of objects which could actually exist in the real world and systems which allow the user to build data representations of impossible objects or data components (such as vertices) that have no physical manifestation except as part of an actual object. The former are generally called manifold modelers, and the latter non-manifold. The term “manifold” has an exact mathematical definition, which we will not give here, but in three dimensions a mathematical manifold corresponds well to a physically realizable object.

2.2 Parasolid Boundary Representation

The only scheme used by the Parasolid system for representing solid objects is an exact boundary representation scheme [Shape90]. The Parasolid system is a manifold modeler; any three dimensional body defined in Parasolid is physically realizable. Conversely, any physically realizable body can be described in Parasolid. The functions provided by Parasolid for manipulating bodies do not allow the user to model an impossible object.

As noted earlier, Parasolid stores information in files called “transmit” files, which may be written or read using functions provided in the Parasolid system. Transmit files may be written either as binary data or as ASCII text data. The Parasolid-to-STEP system uses only the text version. Individual characters and groups of characters in these files are easy enough to read and interpret. A transmit file consist of lots of numbers (integers and numbers with decimal points in them) separated by spaces. The meaning of the file as whole, however, is completely obscure to a reader who does not know the format, and Shape Data keeps the format as proprietary information.

Shape Data also does not disclose the format of the data structures in active computer memory for shape representation. This means that a user cannot meaningfully circumvent the system’s desire to deal with only manifold models, either by modifying structures in active memory or by using a text editor on a transmit file. It also means that it is not possible to reformat a Parasolid transmit file directly into a STEP physical file or to construct a STEP Working Form by parsing a Parasolid transmit file. It is always necessary to use the Parasolid system as an intermediary.

2.3 STEP Boundary Representation

The STEP geometry and topology schemas [Smith 88] provide the entity types required for building exact boundary representations. The schemas were clearly developed with this as the foremost intended use, and this is how Parasolid-to-STEP uses them.

Unlike Parasolid, STEP does not include modeling functions. The STEP geometry and topology schemas provide entity types that can be used for shape representation, and they provide many constraint rules regarding entities - things like requiring that the number of vertices referenced in an edge_loop be twice the number of edges. However, there are no rules provided for how entities of these types may be included in a file to represent specific objects. For example, can a face_logical_structure be used as a component of more than one closed shell, or, is it legal to include a cartesian point which is not used for defining any other entity? The official documentation [Smith88] of the geometry and topology schemas does not deal with questions of this sort. A STEP application protocol specifying how to use STEP geometry and topology to build a file giving a boundary representation is in draft stage [Weick90]. The format adopted for the physical file prepared by Parasolid-to-STEP is discussed in the next chapter of this paper.

A STEP boundary representation is more general than a Parasolid boundary representation because STEP allows 0-dimensional, 1-dimensional, and 2-dimensional manifolds, as well as 3-dimensional manifolds.

2.4 Comparison of STEP and Parasolid Representations

2.4.1 Topology

Because Parasolid builds representations in which all topological entities are required to be components of a physically realizable solid object, some constraints on topological entities are automatic. A Parasolid face, for example, is always a face of a solid object, so one side of the face can be distinguished from the other. A STEP face, on the other hand, is not necessarily a component of any solid object, so there is no natural differentiation between the two sides. To deal with the wider range of cases it must handle, STEP provides “logical structures” which attach a binary flag to a base topology type.

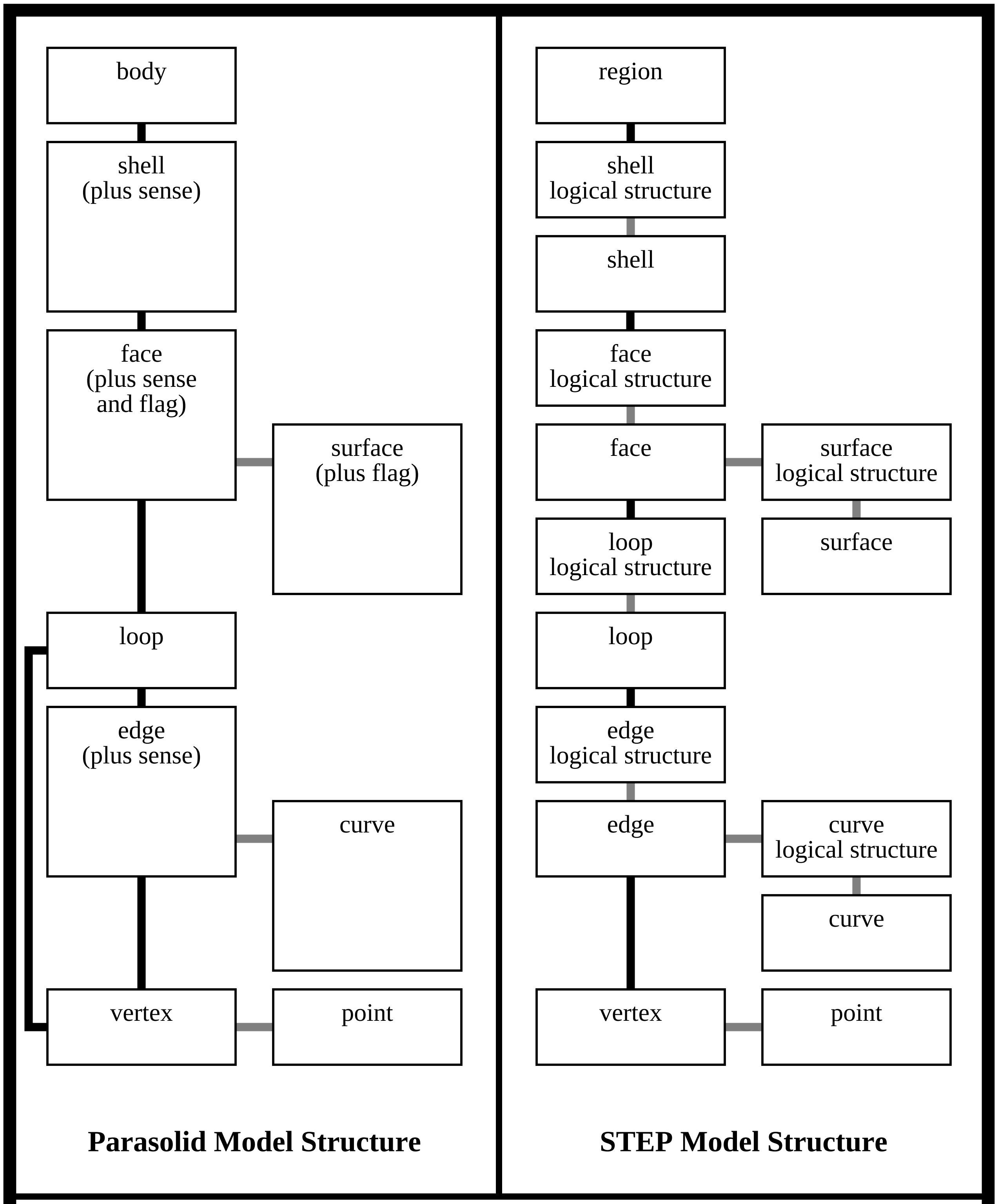
Parasolid does not have entity types equivalent to logical structures. Parasolid does, however, have explicit sense flags and implicit senses for many types of entities, which serves the same function as logical structures in STEP.

Figure 1 shows a comparison of topology in the STEP and Parasolid models, depicting the entity types and their relationships. Parasolid is shown on the left, STEP on the right. The two models are shown with corresponding types in the same relative location. For example, the second box from the top is “shell (plus sense)” on the left and “shell_logical_structure” on the right. This means that a Parasolid shell, plus knowing its sense (which can be extracted from Parasolid), is equivalent to a STEP shell_logical_structure and its shell. Where the word “flag” is used in a Parasolid entity type, that means a binary flag whose value is provided by a call to a Parasolid function. Where the word “sense” is used, that means sense information that may be deduced by examining information provided by calls to Parasolid functions. For example, the flag for a Parasolid face tells whether the face normal, which always points away from material, is in the same direction as the normal to the surface of the face, or in the opposite direction. The sense of a shell, however, must be deduced from its position in the list of shells belonging to a body, the outer shell being first in the list.

The most striking feature of the two models is that they are almost exactly the same, allowing for a Parasolid entity type being equivalent to a STEP entity type plus its associated logical structure. The names of the entity types are all identical, with the exception that a Parasolid "body" is equivalent to a STEP "region."

One significant difference between Parasolid and STEP that is not shown on Figure 1 is that it must be possible to map each STEP face to a plane. Parasolid does not have this limitation. This becomes significant in PTS only in the case of an entire sphere or an entire torus. Parasolid does not require any loops on such surfaces. These two surfaces cannot be mapped to a plane without at least a vertex_loop for a sphere and an edge_loop for a torus. PTS has to make these loops from scratch, using information about the geometry of the surface to build the underlying geometry of the loop. The PTS functions which handle this problem are named make_extras_for_face and make_extras_for_torus.

The rule about mapping a STEP face to a plane is not contained in the Express schema for topology. Rather it is in the text accompanying the schema, which explains what the schema means. A corollary of the rule is that every STEP face must have at least one loop_logical_structure, and this corollary is included in the Express schema.



KEY: Entity above or left may have more than one of entity below or right ━
Entity above or left has exactly one of entity below or right ▬

Figure 1. Comparison of Parasolid and STEP Models

2.4.2 Logical Structures

A detailed examination follows of how logical flags required for STEP logical structures are determined by PTS for each Parasolid entity type for which STEP has a corresponding logical structure. By providing a “logical structure” partner for almost all base topology entity types, STEP has gone farther than necessary and left room for alternate methods of defining entities for the same thing. In particular, curve_logical_structures are completely unneeded since an edge can always be defined to have the same sense as its underlying curve. Parasolid does this, so PTS adopts the convention that only “true” curve_logical_structures will be used. Loop_logical_structures are not needed if the convention is adopted that traversals in opposite directions around the same edge or edges will be defined as two different loops. Parasolid uses this convention, and PTS has adopted it. Thus, PTS uses only “true” loop_logical_structures for converting Parasolid loops. In the case of an entire torus, discussed in section 2.4.1, where a STEP edge_loop must be built for which there is no Parasolid loop, it is more efficient to build only one loop and have loop_logical_structures with true and false logical values, so that has been implemented.

Curve

STEP has a positive sense for each curve. The definition of each type of curve tells what the positive sense is for that type. The logical value in a STEP curve_logical_structure used in an edge is defined to be true if the curve is traversed in its positive direction while going along the curve from the first vertex of the edge to the second.

Similarly, Parasolid has a positive sense for each curve. The definition of each type of curve tells what the positive sense is for that type. For all of the curve types handled by PTS, the positive direction is the same in both models.

As noted above, PTS has adopted the convention that the logical value in a curve_logical_structure made by PTS is always true. Parasolid makes it easy to observe this convention, since the vertices of any open edge on a curve are returned by the Parasolid IDCOEN function ordered going in the positive sense of the curve from the first vertex to the second, and this vertex order is used by PTS for defining each edge.

PTS makes one STEP curve and one STEP curve_logical_structure (the true one) for each curve in the Parasolid model.

Edge

STEP has a positive sense for each edge. If an edge is open, the positive sense is from the first vertex to the second. If the edge is closed, the positive sense may be in either direction as long as the logical flags of (i) the curve_logical_structure underlying the edge and (ii) any edge_logical_structure defined on the edge, are set correctly. The logical value in a STEP edge_logical_structure used in a loop is defined to be true if the positive sense of the edge is the same as the positive sense of the loop.

Parasolid does not have a sense for an edge.

As noted above, PTS makes the positive sense of each edge it creates be the same as the direction of the underlying curve.

PTS makes one STEP edge and two STEP edge_logical_structures (one true and one false) for each edge in the Parasolid model.

Open Edge

If an edge is found in a Parasolid loop of two or more edges, the edge must be open. In this case, it may be determined by examining the list of vertices of the loop, the list of edges of the loop, and the vertices of the edge (all of which are returned by appropriate Parasolid function calls), whether the underlying curve of the Parasolid edge (and hence the corresponding edge created by PTS) is being traversed in the positive or negative direction while traversing the loop in the positive direction. PTS assigns a logical value accordingly when making the STEP edge_logical_structure to be included in a loop.

Closed Edge

If a Parasolid loop has only one edge, it must be closed, which can happen in PTS only if the underlying curve of the edge is a circle or an ellipse. In Parasolid, such an edge may have no vertices, one vertex, or two occurrences of one vertex. STEP requires that every edge have a start vertex and end vertex. If Parasolid does not have any vertex for an edge, PTS finds an arbitrary point on the curve, uses it to make a vertex, and uses the vertex for the start and end of the edge.

The PTS convention that only true logical structures will be made for curves and loops forces decisions on setting logical values into edges. If a loop has only one (necessarily closed) edge, it is necessary to retrieve a list of the two loops on the edge in order to determine if the direction of the original loop is the same as the direction of the edge (which always has the same direction as its underlying curve). Parasolid provides this list with the first loop in the list going in the loop positive direction around the edge in the curve positive direction. Thus, if the given loop is the first one in the list, the logical value is true. If the given loop is second in the list, the logical value is false.

Shell

In STEP, a shell normal is defined to be positive when it is pointing away from whatever the shell encloses (whether space or material). The logical value for a STEP shell_logical_structure used for a region is defined to be true if the shell normal points away from the interior of the region and false otherwise.

Parasolid does not have a flag indicating the sense of a shell, but when the IDCOEN function is asked for the shells of a region, the outer one is always first in the list, and all the others enclose voids in the interior of the region.

Thus, PTS assigns the logical value true to the shell_logical_structure whose shell is the outer shell of a region and false to the shell_logical_structure for all other shells (since they are inner shells).

PTS makes one STEP shell and one STEP shell_logical_structure (true or false, as appropriate) for each shell in the Parasolid model.

Face

In STEP, the positive sense of the normal to a face is arbitrary. The logical value for a STEP face_logical_structure used for a shell is defined to be true if the face normal points in the same direction as the shell normal, and false otherwise.

In Parasolid, the normal of a face always points away from material.

The design decision in building PTS was to have the positive sense of each face generated by PTS agree with the positive sense of the corresponding Parasolid face. Thus, PTS assigns a true logical value to every face_logical_structure used in a STEP shell it builds that is the outer shell of a region (since the shell normal points out and the face normals point out) and false to every face_logical_structure in a shell that is an interior shell of the region (since the shell normal points out and the face normals point in).

PTS makes one STEP face and one STEP face_logical_structure (true or false, as appropriate) for each face in the Parasolid model.

Surface

STEP has a positive sense for each surface. The definition of each type of surface tells what the positive sense is for that type. The logical value for a STEP surface_logical_structure used for a face is defined to be true if the surface normal points in the same direction as the face normal.

Parasolid has a positive sense for each surface. The definition of each type of surface tells what the positive sense is for that type. Surfaces may, however, become reversed. The Parasolid OUTSUR function returns a flag which tells whether this has happened to any given surface.

The positive senses of STEP and Parasolid surfaces are the same for all surfaces handled by PTS.

The Parasolid IDSOFF function returns a flag which tells whether the Parasolid surface normal has the same sense as the Parasolid face normal. PTS makes the STEP surface with the same sense as unreversed Parasolid surface. The logical value for a surface_logical_structure in a face is assigned by PTS as true if either (i) the Parasolid surface is unreversed and the Parasolid surface normal is in the same direction as the Parasolid face normal, or (ii) the Parasolid surface is reversed and the Parasolid surface normal is in the opposite direction from the Parasolid face normal. Otherwise, the logical value is assigned as false.

PTS makes one STEP surface and one or two STEP surface_logical_structures (one true, one false, or one of each, as appropriate) for each surface in the Parasolid model.

Loop

In both Parasolid and STEP, a loop may consist of a single vertex (such a loop is called a vertex_loop), or it may be an edge_loop. In Parasolid, a loop is associated with only one face, and the positive direction of an edge_loop has the face on its left, looking at the face from outside the material while traversing the loop.

A vertex_loop is a vertex which serves as a boundary of a surface. In the surfaces handled by PTS, the only instance of this is the vertex at the tip of a cone. The tip of a cone is identified by Parasolid as a vertex_loop on the face of the cone, and the corresponding STEP entity is created as a vertex_loop by PTS.

As mentioned above, PTS adopts the convention that the logical value of a STEP loop_logical_structure used for building a face is always true. If the loop is a vertex_loop the logical value is irrelevant, so true is fine. If the loop is an edge_loop, as noted above, the loop always is defined in the positive direction on the face.

PTS makes one STEP edge_loop or one vertex_loop for each loop in the Parasolid model. PTS also makes one STEP loop_logical_structure (the true one) for each loop in the Parasolid model.

2.4.3 Geometry

In general, for simple geometry, STEP and Parasolid have the same geometry entity types, use the same names, and have identical or very similar methods of definition. For complex geometry, the differences are more substantial, but PTS does not handle complex geometry.

Points

STEP provides three kinds of points. Only one, cartesian_point, is used by PTS. Parasolid provides one kind of point, cartesian point.

Curves

STEP provides line, four conics (circle, ellipse, parabola, and hyperbola), four types of bounded_curve, four types of curve_on_surface, and two types of offset_curve. Parasolid provides line, two conics (circle and ellipse), intersection curve, and parametric curve (including b_spline). PTS deals only with three of the types common to the two systems: line, circle, and ellipse.

Surfaces

STEP provides five “elementary surfaces” (plane, cylinder, cone, sphere, and torus), plus swept surface, bounded surface, and offset surface. Parasolid provides the same five elementary surfaces, plus swept surface, spun surface, blending surface and parametric surface. PTS deals with only the five elementary surfaces: plane, cylinder, cone, sphere, and torus.

Placing Geometry Entities in 3-Space

STEP and Parasolid differ in how curves and surfaces are placed in 3-space. STEP uses entities while Parasolid uses parameters.

STEP and Parasolid both locate an ellipse in space by having a convention for attaching a coordinate system to the ellipse and then specifying the location and orientation in space of the attached coordinate system. STEP locates the coordinate system using an axis_placement, which is a STEP entity. Parasolid locates the coordinate system using parameters which are real numbers or arrays of real numbers, and are not Parasolid entities. Location of surfaces and circles is similar. Thus, for each Parasolid surface, ellipse, or circle, the STEP model built by PTS has four or five entities: the surface itself, an axis2_placement, and a cartesian_point plus one or two directions required to define the axis2_placement.

The situation for lines is similar but simpler. PTS builds a line, point, and direction in STEP for each Parasolid line.

3 Output Format and Scope of Parasolid-to-STEP

3.1 Output File

The output of Parasolid-to-STEP is a STEP physical file prepared in accordance with the "Tokyo" version of STEP. In particular, geometry and topology entities are as defined in [Smith88] and the file is arranged according to the rules for STEP file structure and the mapping from Express to a STEP physical file given in [Altemueller88a] and [Altemueller88b]. The overall structure of the file is, as required, a header section followed by a data section.

In addition, the file is formatted to be easily read by humans: in the data section of the file, each entity definition starts on a new line, the entities are numbered in sequential order starting with 1, and a modest amount of white space is left to improve readability. Parasolid-to-STEP has two printing options: the number of decimal places to be kept and whether superfluous trailing zeros should be suppressed. Numbers are rounded, rather than truncated, to the selected number of places. An example of a PTS output file is shown in Appendix B. Entity id numbers in this section refer to that example. The part described in that file is shown in Figure 2. The part consists of a cylinder with a conical tip at the bottom and a flat top. The top is bevelled by planes on both sides and has a spherical dimple surrounded by a toroidal groove.

The entity numbers in the file which correspond to the numbered faces in Figure 2 are:

entities 1-17	face 1 - sphere
entities 18-68	face 2 - plane
entities 69-86	face 3 - plane
entities 87-104	face 4 - plane
entities 105-124	face 5 - cone
entities 125-139	face 6 - cylinder
entities 140-158	face 7 - plane
entities 159-170	face 8 - torus

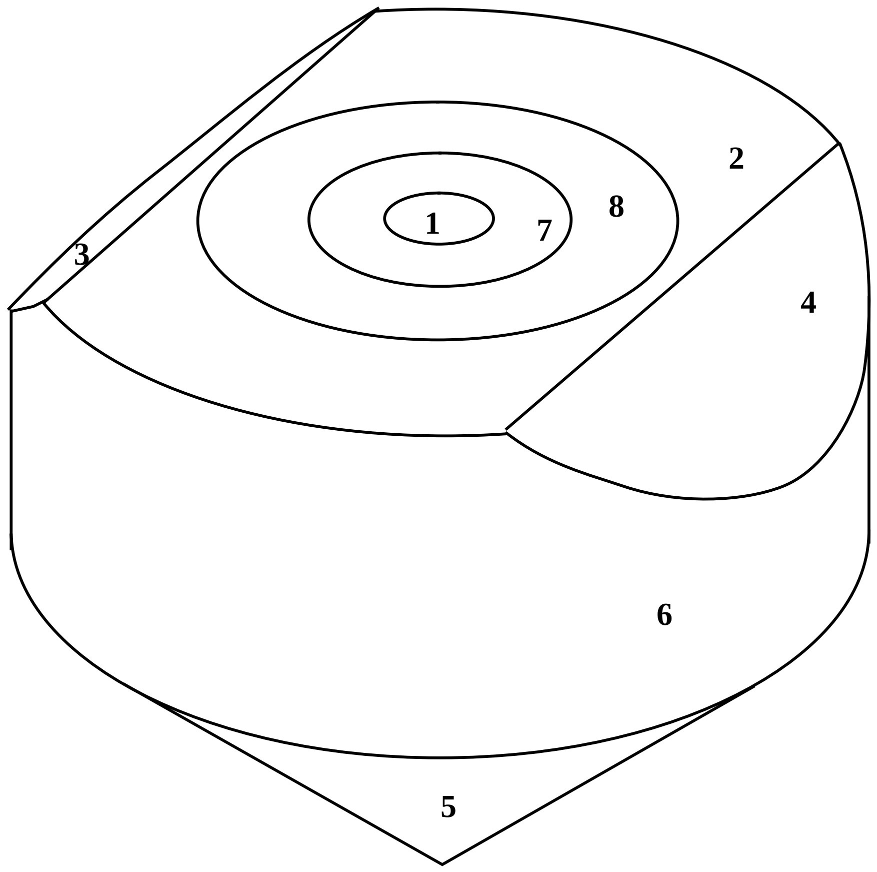


Figure 2. Example Part

3.1.1 Hierarchical Structure

At first glance, the entities in the data section of the output file appear to be in random order, since entities of every type are scattered throughout the file. Actually, the entities are arranged in hierarchical blocks by topological entity type. The hierarchy is the STEP model structure as shown on the right in Figure 1.

The blocks described here are simply sets of adjacent lines. They are not marked in the file in any way. This block structure is a side effect of the way PTS works, not an intentional desirable feature of PTS. For human readability, it is more desirable to group entities of the same type together and to have them ordered within entity type, as shown in Appendix C.

Because the Tokyo version rules for constructing a STEP physical file require that an entity be defined before it is referenced, the entity at the top of each block in the hierarchy is defined in the last line of the block in the file. Thus, in PTS, a region is always defined in the last line of the data section (entity 173 in the example).

Preceding the region definition is a set of blocks, each of which defines a shell_logical_structure. There is only one shell in the example, so it takes up entities 1 through 172. Each block defining a shell_logical_structure has a closed shell (171) as the next to last entity and a shell_logical_structure (172) as the last entity. Preceding that is a set of blocks, each of which defines a face_logical_structure. There are eight faces in the example, so there are eight such blocks in the file.

Each face_logical_structure block (look at 18-68 in the example) ends with the same four entity types: some type of surface (65), a surface_logical_structure (66) on that surface, a face (67) on the surface_logical_structure, and a face_logical_structure (68) on the face. Immediately preceding the surface definition is a block of entities (63-64) required for defining the surface. Before that is a set of blocks (19-27, 28-62) each of which defines a loop_logical_structure used in defining the face.

Each loop_logical_structure block (look at 28-62 in the example) ends with an edge_loop (61) or a vertex_loop followed by a loop_logical_structure (62). If the loop is an edge_loop, this is preceded by a set of blocks, each of which defines an edge_logical_structure (28-37, 38-46, 47-54, 55-60). If the loop is a vertex_loop (117), it is preceded by two lines defining a point (115) and a vertex (116) at the point.

Each edge_logical_structure block (look at 47-54 in the example) ends with a line defining an edge_logical_structure (54). If the edge used to build the edge_logical_structure has not been defined previously, the beginning of the block will define the curve (49), curve_logical_structure (50), point or points (51), and vertex or vertices (52) needed to build the edge. If a vertex has been defined previously, however, it is not defined again (such as 44, which is used in line 53). If the edge of an edge_logical_structure block has been defined previously (69, for example), the block consists of a single line.

3.2 Scope

Parasolid-to-STEP takes as input a Parasolid transmit file which defines a single Parasolid three-dimensional body; this corresponds to a region in STEP. It has an outer closed shell and zero or more other closed shells, each of which represents a completely enclosed void in the body. PTS also takes as input a file defining a STEP schema which includes geometry and topology entity types.

3.2.1 Parasolid Input

The Parasolid transmit file used as input to PTS may include Parasolid entities of the following Parasolid types. None of the entities in the file should have a transformation attached to it. These entity types include only geometry and topology. PTS does not deal with Parasolid features as input.

GEOMETRY

TYPTCA	cartesian point
TYCUCI	circle
TYSUCO	cone
TYSUCY	cylinder
TYCUEL	ellipse
TYCUST	line
TYSUPL	plane
TYSUSP	sphere
TYSUTO	torus

TOPOLOGY

TYTOED	edge - if closed, must be normal (EDEDCN) or ring (ENEDRN)
TYTOFA	face
TYTOLO	loop - hole (ENLOHO), peripheral (ENLOPE) or other (ENLONA)
TYTOBY	body - must be solid (ENBYSO)
TYTOSH	shell
TYTOVX	vertex - must be normal (ENVENO) or isolated (ENVEIS)

3.3 STEP Input Schema and Output Entity Types

The names of STEP entity types PTS will make follow. They include selected geometry and topology entity types only. PTS will not make entities of any other geometry or topology entity type or entities from any other STEP schema (such as features or tolerances). PTS expects to find all these entity types defined in the STEP schema which is input. If any of these is missing from the schema, PTS will print an error message and stop. The schema may include other definitions, rules, etc.; they will be read and checked for having correct syntax, but they will not be used.

Each entity type is expected to have the attributes given in the “Tokyo” version of the STEP geometry and topology. The attributes must be in the same order and have the same meaning as given in the Tokyo version. The names of the attributes, however, are irrelevant to PTS. If the name of an attribute in the schema is “axis” and is changed to “dir_axis,” without changing what it means, for example, this would have no effect on PTS.

GEOMETRY

axis2_placement
cartesian_point
circle
conical_surface
cylindrical_surface
direction
ellipse
line
plane
spherical_surface
toroidal_surface

TOPOLOGY

closed_shell
curve_logical_structure
edge
edge_logical_structure
edge_loop
face
face_logical_structure
loop_logical_structure
region
shell_logical_structure
surface_logical_structure
vertex
vertex_loop

4 Using Parasolid-to-STEP

4.1 Environment

The executable file for Parasolid-to-STEP is “parasolid_to_step.” It is slightly less than 5 megabytes in size. When PTS starts up and finishes parsing in the standard STEP schema of geometry and topology (before the Parasolid file is loaded), the size of the process resident in active memory is about 2 megabytes. Loading the Parasolid file causes the process to grow by about a tenth of a megabyte per 1000 equivalent STEP entities. Building the STEP model causes the process to grow by about one megabyte per 1000 STEP entities. Converting a 22028 byte Parasolid file for which the STEP output file contained 939 entities (35896 bytes) caused the process to grow to a maximum of just over 3 megabytes. Converting a 216302 byte Parasolid file for which the STEP output file contained 7907 entities (334240 bytes) - the largest one tested - caused the process to grow to a maximum of 9.4 megabytes. The machine used to run PTS to convert a specific Parasolid file must be capable of handling a process of the size expected for that file. A crude estimate would be: process size in megabytes equals 2 plus 40 times Parasolid file size in megabytes. PTS is known to run well on a Sun3/160 computer with 16 megabytes of RAM or a Sun3/60 with 24 megabytes of RAM. Testing has not been done with other machines.

In order to use Parasolid-to-STEP, it is necessary to have the Parasolid system, which is available for licensing from Shape Data, Limited.

The output file produced by Parasolid-to-STEP is as described in chapter 3. An example output file is shown in Appendix B for the part shown in Figure 2. A separate system, “compress-step,” built in Common LISP by the author, may be use to reformat the PTS output file to be more concise and easier to read. Appendix C contains the file produced by running the PTS output shown in Appendix B through compress-step. Compress-step performs several operations, the most important of which are (i) putting the file into total order based on entity type and attribute values, and (ii) deleting duplicate entities. Development of a C version of compress-step has begun.

4.2 Getting an Executable

If a user has access to the /usr/local/pdes/src directory at NIST, and access to a Sun3 computer, the conversion may be done conveniently by using the executable which is in the parasolid_to_step subdirectory of /usr/local/pdes/src.

For other machines running a UNIX or SunOS operating system, if a user has a copy of the NIST PDES Toolkit software, has access to the “gcc” C compiler, has the PTS source code (parasolid_to_step.c) in the parasolid_to_step subdirectory of the Toolkit “src” directory, and is the owner of the parasolid_to_step subdirectory, then an executable may be created by changing directories to the parasolid_to_step subdirectory and giving the command “make.”

4.3 Timing

The four time consuming stages of running Parasolid-to-STEP are parsing the STEP schema, loading the Parasolid transmit file, converting the Parasolid model to a STEP model, and printing the STEP file. Total running time and time for each stage have been measured when PTS is running on a Sun3/60 with 24 megabytes of RAM. For a model with 939 STEP entities, total time was about 1 minute. For a model with 7907 STEP entities, total time was about 13 minutes. Total time increases roughly in proportion to the increase in size of the input file, as does the output file size (which is usually about 1.5 times input file size).

Schema parsing requires about 20 seconds when the standard STEP geometry and topology schema is used.

Reading the Parasolid input file took two seconds for the smaller file and 20 seconds for the large one.

Converting the model took 34 seconds for the small model and 12 minutes for the large.

Printing the output file took 6 seconds for the small file and 40 seconds for the large.

4.4 Command Call

The executable file for Parasolid-to-STEP is “parasolid_to_step.” When it is used as a command, there is one required argument: the name of a Parasolid transmit file. The argument should not have the suffix “.xmt_txt” on it. That is added by the system. The file itself must have the suffix “.xmt_txt”. The “.xmt_txt” file must be in the directory from which the call to “parasolid_to_step” is made.

In addition, the function has two options. The “-s” option is followed by the STEP output file name the user wants. The “-e” option is followed by the name of the Express schema input file.

The STEP output file name defaults to step.out, and the Express schema file name defaults to geom-topo.exp . The schema must include the geometry and topology entity type definitions listed in section 3.3.

Example 1 - The simplest possible form of a call to parasolid_to_step:

```
parasolid_to_step  cover
```

For this call to work, the directory from which the call is made must contain the files: parasolid_to_step, cover.xmt_txt, and geom-topo.exp . The output file will be called step.out.

Example 2 - A call to parasolid_to_step using all options:

```
../parasolid_to_step -e ../geom-topo.exp -s block.step clevis2_block
```

For this call to work, the directory from which the call is made must contain the file clevis2_block.xmt_txt, and the parent directory must contain the files geom-topo.exp and parasolid_to_step . The output file will be called block.step and will be found in the directory from which the call was made.

After model conversion is complete and before output file printing starts, the user is prompted to enter the number of decimal places to be used for printing numbers and is prompted to say whether superfluous trailing zeros should be suppressed when printing numbers. With zero suppression on, PTS would print 3.123 rather than 3.12300000, for example. The minimum number of decimal places is 1. The maximum is 16.

4.5 Messages and Errors

While Parasolid-to-STEP is running it prints messages from time to time to let the user know it is alive and well, or that it isn't.

4.5.1 Healthy Messages

As long as PTS is healthy, it prints messages as follows, shown in italics. Items that may change depending upon the command are shown in boldface italics. User input is shown in non-italics. In item 5 below, which gives the number of STEP entities created so far, another line is printed each time another Parasolid face is converted.

1. *STAMOD: Start Modeler*
   Modeler version number 220
2. *Compiling Express from* ***/usr/local/pdes/src/parasolid_to_step/geom-topo.exp***
3. *Loading Parasolid file* ***shuttle_block.xmt_txt****.*
4. *Converting Parasolid model to STEP working form model.*
5. ***42*** *STEP entities*
   70 *STEP entities*
   92 *STEP entities*
   •
   •
   •
   939 *STEP entities*
6. *Enter number of decimal places*
   for printing real numbers, minimum 1 maximum 16 : 6
7. *Suppress trailing zeros for real numbers ? (y/n) :* y
8. *STOMOD: Stop Modeler*

4.5.2 Error Messages

Error handling is covered in detail in chapter 6. The basic approach of PTS is to print one or more helpful messages and quit if something is wrong. Errors encountered during model conversion are discussed in chapter 6. The main function of PTS will detect problems with the command call or the files and print messages (shown in italics) as follows.

Incorrect Command Call

If the command call is incorrectly constructed, for example by using an illegal -x option:

parasolid_to_step: illegal option -- x
usage: parasolid_to_step [-e express][-s step] input_name

Schema File Won’t Open

If the STEP schema file cannot be opened (probably because the wrong file name or path name was used):

Error opening schema file [file name].

Parasolid Modeler Will Not Start

Modeler did not start

Fed-X Will Not Start

Express did not initialize

Problem With STEP Schema

If Fed-X cannot parse the schema, it will probably print one or more messages itself. This will be followed by:

Errors in Express input

If the schema does not include definitions of all 24 required STEP entity types:

Schema differs from expected

If the required entity types are defined in the schema, but they are defined differently from what is expected by PTS, no error will occur during schema parsing. An error will occur later during model conversion.

Parasolid File Will Not Load

If the Parasolid file cannot be read in (probably because wrong file name or path name was used, or the file is not a transmit file):

Loading Parasolid file shuttle_block.xmt_txt.xmt_txt.
GETMOD called in load_object returned non-zero ifail 58.
Parasolid would not load shuttle_block.xmt_txt.

The easiest way to get the file name wrong is to include the .xmt_txt suffix in the function call. This error will be apparent in the error message, since the suffix will be included twice, as shown in the first line above.

Any Error in Model Conversion

Error in model conversion

Output File Cannot be Opened

Error opening output file [file name]

5 How Parasolid-to-STEP Works

5.1 Approach

The approach implemented in Parasolid-to-STEP is to parse in the Express schema by using Fed-X (the NIST express parser), create a Parasolid model in active memory by using the Parasolid file loading function to read in a Parasolid transmit file, and then traverse the tree of connected entities in the Parasolid model, building a STEP Working Form product model from geometry and topology entities which correspond to the Parasolid entities as the Parasolid model is traversed. The similarity between the Parasolid and STEP models shown in Figure 1 makes it feasible to build the STEP tree in parallel with traversing the Parasolid tree. Once the STEP Working Form product model is built, a set of file printing functions prints the STEP physical file from it.

This approach isolates the effect of changes in the physical file format to the printing functions. It also provides a little help in dealing with changes in the geometry and topology schemas, but not much. In general, the boundary representation modeling approaches of both STEP and Parasolid are hard-coded in this system.

5.2 Details

5.2.1 What the Main Function Does

The main function goes through the following steps in the order shown here. After each step it checks to be sure it succeeded before proceeding to the next step. The steps that take a noticeable amount of time are 5, 8, 10, and 12.

1. get the command line arguments from the call to Parasolid-to-STEP
2. open the STEP schema file for reading
3. start the Parasolid system
4. initialize the Express schema parser
5. parse the STEP schema (two passes)
6. close the schema file
7. check that parsing the schema defined all required entity types
8. load the Parasolid transmit file to create the Parasolid model
9. create an empty STEP Working Form Product
10. traverse the Parasolid model, filling the STEP Working Form Product
11. open the STEP output file for writing
12. print the STEP file from the list of entities in the STEP Working Form Product
13. close the STEP output file
14. stop the Parasolid system

5.2.2 Keeping Track of What Has Already Been Converted

Parasolid provides that when it is started, the user may direct that some number of user fields (each of which holds an integer) will be attached to each Parasolid entity. Parasolid-to-STEP keeps track of the status of converting a Parasolid entity by employing these user fields. PTS uses three such fields in each entity. The first user field of each Parasolid entity is used to hold the entity number of the STEP entity created as a counterpart, the second user field holds the entity number of the "true" logical structure built on that STEP entity (if any), and the third user field holds the entity number of the "false" logical structure built on that STEP entity (if any). All STEP entities which have a Parasolid counterpart are created by PTS by a call to the add_entity_mark function, which fills in the first user field as well as creating the entity. All "logical structure" entities whose base type parent has a Parasolid counterpart are created by PTS by a call to the add_logical_entity_mark function, which fills in the second or third user field as well as creating the logical structure entity.

The PTS functions which convert Parasolid entities to STEP entities almost always check one, two, or all three user fields of the Parasolid entity. The checks are made for several purposes. If the Parasolid entity is an edge, for example, the first field will tell if the STEP counterpart exists; if so, exactly one of the two logical structures on the edge should also already have been created, and it should have the opposite sense from the one which is now required. The user fields provide this information. If the Parasolid entity is a face, as another example, no STEP equivalent should have been created, since each Parasolid face should be encountered only once in traversing the Parasolid model. Again, the user fields provide this information.

If STEP entities must be created which have no counterpart entity in the Parasolid model, obviously the user fields are not useful in keeping track of them. This happens when STEP points, directions, and axis placements must be created to define the STEP equivalent of a Parasolid curve or surface. When that happens, duplicate entities may well be created, such as two points at the same place or two unit vectors in the same direction. PTS does not try to forestall creating such duplicates except for the six principal unit vectors, each of which will be created at most once.

5.2.3 Two Important Variables

STEP entities are given id numbers in sequential order starting with 1 by PTS. In order to keep track of which number comes next, the "line_number" global variable is used. It is referenced and reset by the add_entity_mark and add_logical_entity_mark functions and it is referenced but not set by the convert_shell and make_direction functions.

Another key variable used throughout PTS is the STEP Working Form Product being built. It is passed around as an argument (always named "product") rather than being a global variable. Most of the functions which take it as an argument add entities to it.

6 Error Handling in Parasolid-to-STEP

6.1 General Approach

Parasolid-to-STEP does a great deal of checking when it runs. Both the data being used and the operation of function calls being made are checked. During operation, PTS calls functions written by the author, functions for manipulating the STEP Working Form, and Parasolid functions.

Errors in the command call to PTS by the user are discussed in Chapter 4, “Using Parasolid-to-STEP.”

The first stage of PTS operation, parsing the STEP schema file, is handled entirely by Fed-X. The documentation of Fed-X [Clark90b, Clark90c] should be consulted if there are problems. The rest of PTS’s work (reading in the Parasolid transmit file, converting the resulting Parasolid body to a STEP Working Form Product whose top-level entity is a region, and printing a STEP file for the product) uses Parasolid functions, Express and STEP Working Form functions, and PTS functions written by the author.

6.2 Error Sources

6.2.1 Parasolid Function Errors

Every Parasolid function has an argument which is a pointer to a failure indicator. This failure indicator is set by Parasolid when a call to the function is made. PTS checks the failure indicator after every call to any Parasolid function. If there is an error, the PTS function which prints Parasolid error messages is called right away. Most of the error handling for Parasolid functions is embedded in the compiler macros named MODELn and is invisible in the text of the PTS functions which call Parasolid functions.

6.2.2 STEP Working Form Function Errors

Most of the STEP Working Form functions also have an argument which is a pointer to a failure indicator. This failure indicator is checked by PTS for only a few of the more critical Working Form functions. Many of the Working Form functions themselves also print messages to “stderr” in case of error.

6.2.3 PTS Function Errors

The PTS functions whose names start with “convert” or “make” all return something. All of these functions have “error_return” values of the correct type, which indicate that an error has occurred. Most of the error handling for these functions is embedded in the compiler macros named CALLn and is invisible in the text of the PTS functions which call these functions.

Any time a user flag set by PTS in a Parasolid entity is examined, and it indicates that an entity should have been made in the STEP Working Form Product being built, a check is made that the STEP entity exists.

6.2.4 Parasolid Model Errors

The input data for PTS comes from a Parasolid transmit file. When this file is read in by Parasolid, a model is automatically set up in active memory by Parasolid. Many explicit checks are made of the validity of this model. The names of the functions which make such checks with the error messages they print and an explanation (if needed) are listed in Table 1. The functions are in alphabetical order. The checks made on the Parasolid model are spot checks only, not a comprehensive validation of the model. In most cases, a spot check is made if the PTS function making the check will not work if an unexpected situation exists.

6.3 If an Error is Found

If an error is detected during operation of PTS, usually the function in which the failure occurred will print a message and return an error indicator to the function which called it. Then the calling function will check the error indicator, print its own message, and return an error indicator to its calling function, and so on up the hierarchy of function calls to the top of the stack. When the main function of PTS is notified of an error, it exits PTS after printing its message, and does not print an output file.

This method of error handling leaves the user with a clear description of the error, and a trace of the stack at the time the error occurred. It does not provide the user with any method of error recovery. If the problem is in the Parasolid transmit file, it is expected that either the file will be changed and another attempt made to convert it, or it will not be converted. If the problem lies in any of the three software packages, the error will have to be fixed and PTS rebuilt - not something the user is expected to be able to do.

It would be nice to be able to deal with valid entity types in the input file (transformations, b-splines, and intersection curves, for a start) which cannot be handled by PTS, in such a way that a blank space would be left in the output file for the user to fill in later, without aborting the entire conversion process. This has not been attempted.

Error messages are printed to “stderr.”

Table 1. Checks on Parasolid Model

Function	Message	Error Condition
convert_curve	Cannot handle intersection curve Cannot handle parametric curve Unknown curve type	A curve being converted is an intersection curve. A curve being converted is a parametric curve. A curve being converted is not a circle, line, ellipse, intersection curve, or parametric curve.
convert_edge	Vertices of edge do not match up Edge used more than twice Edge used twice in same direction	The vertices at the nth edge of a loop are not the same as the nth pair of vertices of the loop. An edge is part of more than two loops. An edge is traversed twice in the same direction.
convert_face	Face used twice Found two outside loops	A face has been used twice. A face has two outside loops.
convert_loop	Loop used twice Numbers of edges and vertices differ	A loop is being converted for the second time. A loop with more than one edge does not have the same number of edges and vertices.
convert_loop_ edge	Edge used more than twice Edge used twice in same direction Curve not circle or ellipse Ring edge has vertices Zero or more than 2 vertices two different vertices Bad edge type	An edge is part of more than two loops. An edge is traversed twice in the same direction. The curve of a closed loop is not a circle or ellipse. A ring edge has vertices. A closed normal edge has been found with zero or more than two vertices. An edge that forms a complete loop has two different vertices. An edge is neither a ring edge not a closed edge.
convert_point	Unknown point type	A point is not a cartesian point.
convert_shell	Shell used twice	A shell has been used twice.
convert_surface	Unknown surface type	A surface is not a plane, cylinder, sphere, cone, or torus.
convert_vertex_ loop	Number of vertices is not 1 Vertex not isolated	A vertex loop has more than one vertex. The vertex of a vertex loop is not isolated type.
find_ring_sense	Ring edge not connected to 2 loops Parent loop not connected to edge	A ring edge is not used for making two loops. A loop is not connected to an edge which is an edge of the loop.
main	Parasolid entity is not a solid body	A Parasolid entity is not a solid body.
make_extras_ for_face	Surface is not a sphere or torus	The surface of a face with no loops is not a sphere or torus.
make_vertex_ on_curve	Bad axis unit vector for circle Curve not circle or ellipse	A purported unit vector does not have length 1. The curve of a loop with no vertices is not a circle or ellipse.

7 Software of Parasolid-to-STEP

7.1 Overview

The C software written by the author for Parasolid-to-STEP is all in the file parasolid_to_step.c in the parasolid_to_step directory of /usr/local/pdes/src. In order to build an executable file, of course, the compiler needs to draw on the Parasolid system, the Fed-X system, and the STEP Working Form functions. These are all referenced appropriately by the Makefile in the parasolid_to_step directory and by the “include” directives to the compiler at the beginning of parasolid_to_step.c.

The PTS software is written in ANSI C [ANSI89]. It has been written to be readable. Compiler macros replace several of the uglier C special character combinations with English words. Almost no abbreviations are used for function or variable names, and the code is heavily commented. In general, function and variable names are words joined together with underscores, as in “parasolid_to_step.”

7.2 Hierarchy of Function Calls

The hierarchy of function calls used by Parasolid-to-STEP is shown in Figure 3. The figure does not include:

1. calls to Parasolid functions
2. calls to Fed-X or STEP Working Form functions
3. calls to report_ifail (which follows every call to a Parasolid function)
4. calls to add_entity_mark, which is called by:
   convert_edge, convert_edge_loop, convert_face, convert_loop_edge, convert_region, convert_shell, convert_vertex, convert_vertex_loop, make_axis2_no_x, make_axis2_with_x, make_circle, make_cone, make_cylinder, make_direction, make_ellipse, make_extras_for_face, make_extras_for_torus, make_line, make_plane, make_point, make_sphere, make_torus, make_vertex_on_curve.
5. calls to add_logical_entity_mark, which is called by:
   convert_curve, convert_edge, convert_face, convert_loop, convert_loop_edge, convert_shell, convert_surface, make_extras_for_face, make_extras_for_torus.

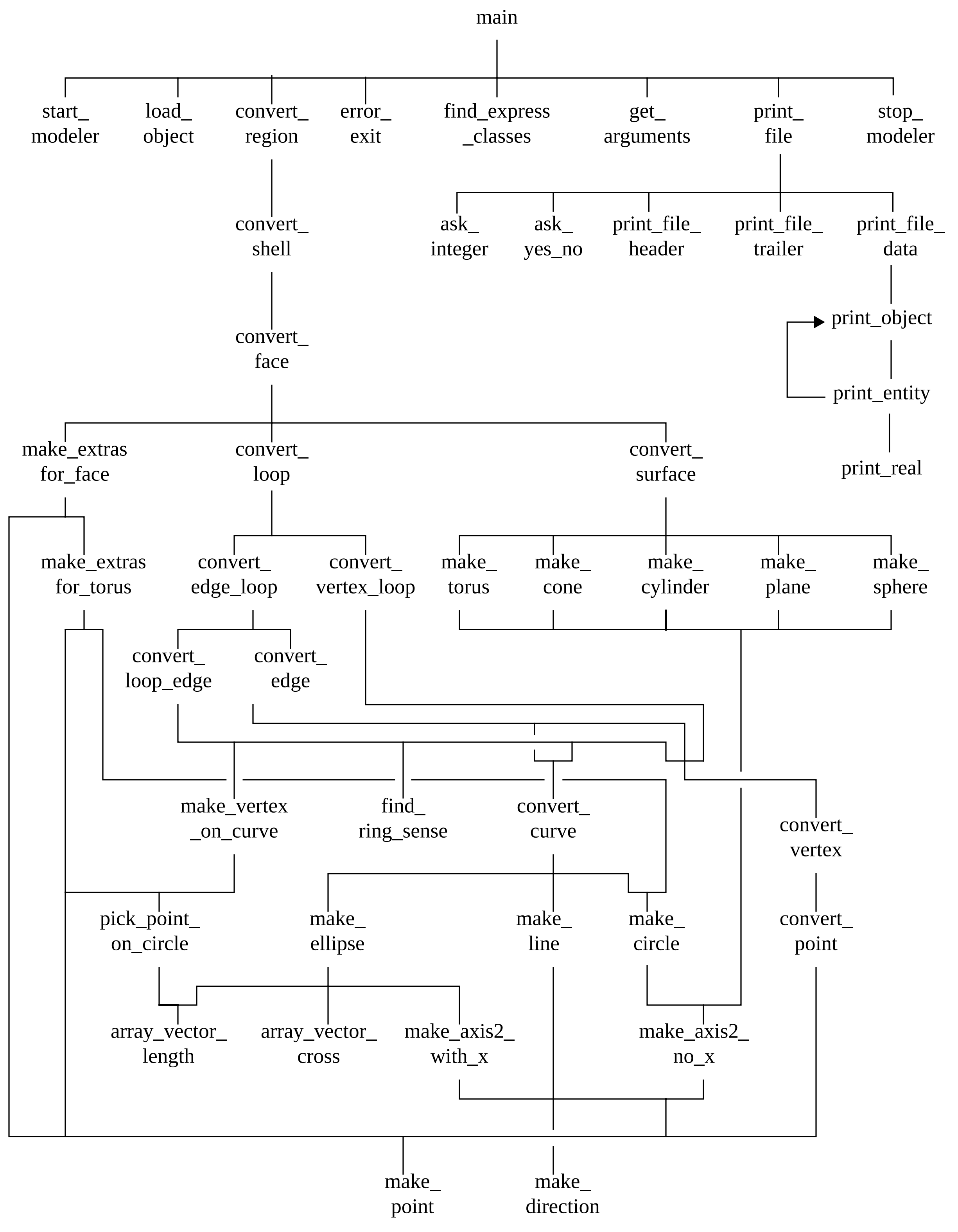


Figure 3. Hierarchy of Function Calls

Function calling flows downwards and horizontally along solid lines, except that print_entity calls, its caller (print_object) as indicated by an arrow going upwards.

7.3 Use of Fed-X and STEP Working Form Functions

The Fed-X system is accessed in Parasolid-to-STEP by function calls to EXPRESSinitialize, EXPRESSpass_1, and EXPRESSpass_2.

In addition, the following Express Working Form and STEP Working Form functions are called by PTS. They are described in [Clark90c] and [Clark90e].

CSTget_name
ERRORhas_error_occurred
ERRORreport
ENTITYget_all_attributes
ENTITYget_name
LISTadd_first
LISTadd_last
LISTcreate
LISTdo
LISTempty
LISTfree
OBJcreate
OBJcreate_entity
OBJaggr_at
OBJaggr_at_put
OBJaggr_lower_bound
OBJaggr_upper_bound
OBJfast_get_attribute
OBJget_name
OBJget_type
OBJinitialize
OBJis_external
OBJput_name
OBJput_value
PRODadd_object
PRODcreate
PRODget_contents
PRODget_named_object
SCHEMAget_scope
SCOPElookup_entity
TYPEget_class
TYPEget_entity

7.4 Use of Parasolid Functions

Parasolid-to-STEP calls the following Parasolid functions. They are described in [Shape90].

ENBYTY	ENquire BodY TYpe
ENEDTY	ENquire EDge TYpe
ENLOTY	ENquire LOop TYpe
ENVETY	ENquire VErtex TYpe
GETMOD	GET archived MODel
GTTGLI	GeT values from a TaG LIst
IDCOEN	IDentify COnnected ENtities
IDCOFE	IDentify Curve OF Edge
IDPOFV	IDentify Point OF Vertex
IDSOFF	IDentify Surface OF Face
OUTCUR	OUTput CURve
OUTPOI	OUTput POInt
OUTSUR	OUTput SURface
OUUFEN	OUtput User Field of ENtity
SEUFEN	SEt User Field of ENtity
STAMOD	STArt MODeller
STOMOD	STOp MODeller

7.5 Descriptions of Parasolid-to-STEP Functions

Brief descriptions of the PTS functions follow in alphabetical order.

The functions whose names start with “convert” are similar to one another. Each such function converts a Parasolid entity into an equivalent STEP entity and/or one of the logical structure counterparts to the STEP entity. All of these functions call on add_entity_mark and/or add_logical_entity_mark to do the actual STEP entity creation.

The functions whose names start with “make” are also similar. In most cases, such functions take a few arguments which are numbers or arrays of numbers and create a STEP entity of the given type from the numbers. All of these functions call on add_entity_mark and/or add_logical_entity_mark to do the actual STEP entity creation. With the exception of make_point, none of these functions has a corresponding Parasolid entity for the STEP entity being made.

In these descriptions, whenever a function is said to “make” a STEP entity, it is to be understood that the entity is added to the STEP product being built, unless a description explicitly states otherwise. Since all STEP entities are put into the product by either add_entity_mark or add_logical_entity_mark, calls to these functions are not mentioned.

add_entity_mark

This function takes a Product being constructed (product), an entity class name (class), a Linked list of attributes (attributes), and a Parasolid entity tag (tag), which may be null.

The function makes a STEP entity from a list of its attributes and adds the STEP entity to the STEP Working Form Product being constructed. The external variable line_number is used as the line number of the STEP entity and also (converted to a string) as the name of the entity.

If the Parasolid entity tag is not null, the line_number is inserted as the value of the first user field of the Parasolid entity. This makes it possible to determine if any Parasolid entity has already been converted to STEP form.

If the function runs without error it returns the new STEP entity. Otherwise, it returns a null Object.

add_logical_entity_mark

This function takes an Object (object) which is an instance of a STEP entity, a STEP Working Form Product (product), an Entity (class), a Parasolid tag (tag), which is null or is the tag of the Parasolid entity corresponding to the object, and a boolean enumeration constant (logical_flag). The class should be the _logical_structure equivalent of the entity class of the object. For example, if object is a loop, class should be loop_logical_structure. If tag is null, that indicates there is no Parasolid entity corresponding to the STEP entity being built.

The function checks the Parasolid entity (if there is one) to see if the corresponding STEP logical entity already exists. If so, it finds it in the product and returns it.

If not, it creates a STEP entity of type class and inserts it in the product. If tag is not null, the function also puts the name of the corresponding STEP logical entity in the second (for true) or third (for false) user field of the Parasolid entity. Then it returns the STEP entity.

array_vector_cross

This function computes the cross product of two 3d vectors (first and second) expressed as arrays of three doubles. It returns an array of 3 doubles.

array_vector_length

This function computes the length of a 3D vector (vector) expressed as an array of 3 doubles. It returns a double.

ask_integer

This function takes a string (prompt), and two integers (minimum and maximum). It prints the prompt on the screen and reads the user’s response as an integer. If minimum and maximum are not equal, it checks that the response falls between the minimum and the maximum and prompts again if not. When the user provides a response in bounds, it is returned.

ask_yes_no

This function takes a string (prompt). It prints the prompt on the screen followed by "? (y/n) :" and reads the user's response as a character. If the character is y, 1 is returned. If the character is n, 0 is returned. Otherwise, the user is prompted again. Only the first character of the user's response is used. The rest is discarded. Thus, "yes" and "no" will also be recognized as valid responses (but Y and N will not).

CALLn

This "function" is used to make error-handling invisible in the text of PTS functions. It is actually a series of compiler macros defined for values of n from 2 through 5. It sets a variable to the value returned by some other PTS function (call it "foo") which returns a STEP Working Form Object. The value of n is the number of arguments to foo. The first argument to CALLn is the variable, the second is foo, and the rest of the arguments are the arguments to foo. CALLn calls foo and sets the variable to the value returned. Then it tests to see if the value is OBJECT_NULL. If so, it prints an error message of the form "bar called foo, which failed," where bar is the name of the function from which the CALL is being made. Then it returns an error return value specified by bar.

In order for CALLn to work, every function which contains a CALLn must have a local variable "function_name," the value of which is a string that should be the function name, and a local variable "error_return," the value of which has the same type as the type returned by the function. The names of these two variables are built into CALLn.

CALLn has a counterpart named MODELn for calling Parasolid functions.

convert_curve

This function takes the tag (parasolid_curve) of a Parasolid curve and a STEP Working Form Product (product). It calls the appropriate curve converter (one of make_line, make_circle, make_ellipse) to make the STEP curve. It then makes the companion curve_logical_structure.

The function returns the STEP curve_logical_structure.

convert_edge

This function takes the tag (parasolid_edge) of a Parasolid edge, two Parasolid vertices (start_vertex and end_vertex), and a STEP Working Form Product (product).

The start_vertex and the end_vertex are the upstream and downstream vertices of the edge with respect to the sense of the edge_loop. parasolid_vertex1 and parasolid_vertex2 are the upstream and downstream vertices of the edge with respect to the sense of the curve of the edge. If start_vertex = parasolid_vertex1 and end_vertex = parasolid_vertex2, then the logical flag of the edge_logical_structure is true. If they are reversed, it is false.

The function returns the STEP edge_logical_structure.

convert_edge_loop

This function takes the tag (parasolid_loop) of a Parasolid loop, the tag (parasolid_edges) of a list of Parasolid edges, the tag (parasolid_vertices) of a list of Parasolid vertices, an integer (number_edges) representing the number of edges in the list, and a STEP Working Form Product (product). It calls convert_loop_edge (if there is only one edge) or makes several calls to convert_edge (if there is more than one edge) to make the STEP entities for the edge(s) of the loop. Then it makes the edge_loop.

The function returns the STEP edge_loop.

convert_face

This function takes the tag (parasolid_face) of a Parasolid face, an integer (shell_counter), and a STEP Working Form Product (product). It calls convert_surface to make the STEP entities for the surface of the face. It calls either convert_loop (if there is at least one Parasolid loop) or make_extras_for_face (if there are no Parasolid loops) to make the STEP entities for the loop(s) of the face. It then makes the face and its companion face_logical_structure.

The sense of the face_logical_structure is set according to the value of the shell counter, being true if the counter is 1 and false otherwise.

The “reverse” flag returned by IDSOFF says whether the normal to the face (which always points away from material), is in the same direction as the normal to the surface. It is passed to convert_surface.

The function returns the STEP face_logical_structure.

convert_loop

This function takes the tag (parasolid_loop) of a Parasolid loop and a STEP Working Form Product (product). It calls convert_edge_loop or convert_vertex_loop to make the STEP loop corresponding to the Parasolid loop. Then it makes the companion loop_logical_structure. The sense of the loop_logical_structure is always true.

The function returns the STEP loop_logical_structure.

convert_loop_edge

This function takes the tag (parasolid_edge) of a Parasolid edge, the tag (parasolid_loop) of a loop on the edge, and a STEP Working Form Product (product). The edge is expected to be a full circle or ellipse. Any other loop type will cause an error.

Finding the sense of the edge is accomplished by calling find_ring_sense.

The loop edge may be a ring with no vertices or a closed edge with one or two vertices. If the closed edge has two vertices, they must be the same vertex. Parasolid will not allow two different vertices in the same place. A STEP edge is retrieved from the product or converted from the Parasolid edge. The edge is used to make an edge_logical_structure with the appropriate logical value.

If there is no vertex, the function will generate a point on the edge to use as a vertex. The same vertex will do for the start and end of the edge.

The function returns the STEP edge_logical_structure.

convert_point

This function takes the tag (parasolid_point) of a Parasolid point and a STEP Working Form Product (product). If the point has already been converted to STEP form, the STEP point is returned. If not, make_point is called to make the STEP point. The STEP point is returned.

convert_region

This function takes the tag of a Parasolid solid body (body) and an empty STEP Working Form Product (product) as arguments. The function fills up the STEP Working Form Product so it represents a solid of the same shape as the Parasolid body, using STEP topology and geometry entities. The function returns 1 if there are no errors, 0 otherwise.

convert_shell

This function takes the tag (parasolid_shell) of a Parasolid shell, an integer (shell_counter) which will be 1 for the first shell in a region, and something larger for the other shells, and a STEP Working Form Product (product). It calls convert_face to make the STEP entities for the faces of the shell. It makes the shell and its companion shell_logical_structure. The sense of the shell_logical_structure is true if shell_counter is 1, and false otherwise.

The function returns the shell_logical_structure.

convert_surface

This function takes the tag (parasolid_surface) of a Parasolid surface, an integer flag (reversed), and a STEP Working Form Product (product). If the surface has already been converted, it retrieves the STEP surface from the product. If not, it calls the appropriate surface converter (one of make_plane, make_cylinder, make_torus, make_sphere, make_cone) to make the STEP surface. It then makes the companion surface_logical_structure. The sense of the surface_logical_structure is obtained by considering the “reversed” argument and using the Parasolid surface enquiry function, OUTSUR, to see if the surface is reversed from its normal sense.

The Parasolid surface normal vectors for the surface types handled all point in the same direction as the STEP normals for the corresponding surfaces. See the documentation of OUTSUR in Chapter 21 of the Parasolid manual, and the entity definitions in the STEP geometry schema.

The function returns the STEP surface_logical_structure.

convert_vertex

This function takes the tag (parasolid_vertex) of a Parasolid vertex and a STEP Working Form Product (product). If the vertex is already converted, the function does nothing. If not, it calls convert_point to make the STEP entity for the point of the vertex. Then it makes the vertex on the point.

The function returns the STEP vertex.

convert_vertex_loop

This function takes the tag (parasolid_loop) of a Parasolid loop, the tag (parasolid_vertices) of a list of Parasolid vertices, an integer (number_vertices) representing the number of vertices in the list, and a STEP Working Form Product (product). It expects the number of vertices to be 1, and checks for that. This vertex is a Parasolid vertex_loop, such as at the tip of a cone. The function makes the corresponding STEP vertex_loop.

The function returns the STEP vertex_loop.

error_exit

This function prints a message and calls the exit function.

find_express_classes

This function takes a Schema (schema) and sets a number of external variables, each of which is supposed to be a particular entity class, to the appropriate class from the schema. If any class cannot be found in the schema, the function prints an error message. If all classes are found, the function returns 1. Otherwise it returns 0.

The function works by building an array of pointers to the external variables and a corresponding array of the strings which are the names of the entity classes that should be in the schema. Then it calls SCOPElookup_entity repeatedly to find and set the value of each external variable. The values (which are Entities) are extracted from the scope of the schema.

find_ring_sense

This function takes a Parasolid edge (parasolid_edge) and a Parasolid loop (parasolid_loop) which has parasolid_edge as its sole underlying edge. The function returns an integer which is 1 if the curve of the edge has the same sense as the loop and 0 if not. If there is an error, it returns 2.

It works by calling IDCOEN to return a list of the two loops of the edge. IDCOEN puts the loop which is on the surface to the left of the edge (as it is traversed in the direction of the curve of the edge) first in the list. Thus, if the parasolid_loop is first in the list, the parasolid_loop has the same sense as the curve, and if the parasolid_loop is second in the list, the parasolid_loop has the opposite sense from the curve.

get_arguments

This function takes five arguments. The first two are the standard command line arguments of a C main function: an integer (argc) giving the number of entries on the command line, and an array (argv) of strings which are the command line entries. The next three arguments (schema_name, input_name, output_name) are empty strings into which the correct information will be placed. The function copies the appropriate names into schema_name (defaults to geom-topo.exp), output_name (defaults to step.out), and input_name.

The function uses the C library function getopt to deal with argc and argv.

The function returns 1 if it executes without error, and 0 otherwise.

load_object

This function takes one argument: a string (input_file) which is the name of a Parasolid transmit file describing a single solid object. It loads the file into Parasolid, which automatically creates a model of the object. If the load is successful, it returns the tag of the object. If not, it returns a null tag.

main

This is the main function for Parasolid-to-STEP. It proceeds as described in subsection 5.2.1.

make_axis2_no_x

This function takes a STEP Working Form Product (product) and two arrays of three doubles representing (i) a cartesian point (point) which is to be the value of the location attribute of the axis2_placement, and (ii) a direction (axis) which is to be the value of the axis attribute. It calls make_point to make the point and make_direction to make the direction and uses them to make an axis2_placement. The ref_direction (the x-axis direction) of the axis2_placement is null.

The function returns the STEP axis2_placement.

make_axis2_with_x

This function takes a STEP Working Form Product (product) and three arrays of three doubles representing (i) a cartesian point (point) which is to be the value of the location attribute of the_axis2 placement, (ii) a direction (axis) which is to be the value of the axis attribute, and (iii) a direction (reference) which is to be the value of the ref_direction attribute, i.e. the x-axis direction. It calls make_point to make the point and make_direction (twice) to make the two directions and uses them to make an axis2_placement.

The function returns the STEP axis2_placement.

make_circle

This function takes a STEP Working Form Product (product), two arrays of three doubles representing a cartesian point (point) which is the center of the circle and a vector (unit_vector) which gives the direction of the normal to the plane of the circle, and a double (radius) representing the radius of the circle. It makes a STEP axis2_placement and a real number object, and uses them to make a circle.

The function returns the STEP circle.

make_cone

This function takes a STEP Working Form Product (product), two arrays of three doubles representing a point on the axis (point) and a unit vector in the direction of the axis (unit_vector), a double (radius) representing the radius of the cone, and a double (half_angle) representing half the angle of the cone. It makes an axis2_placement with no reference direction for the x-axis, and uses it as the axis. Then it makes the conical_surface, which is returned.

make_cylinder

This function takes a STEP Working Form Product (product), two arrays of three doubles representing a point on the axis (point) and a unit vector in the direction of the axis (unit_vector), and a double (radius) representing the radius of the cylinder. It makes an axis2_placement with no reference direction for the x-axis, and uses it as the axis. Then it makes the cylindrical_surface, which is returned.

make_direction

This function takes a STEP Working Form Product (product) and an array of three doubles representing a direction (unit_vector). It returns a STEP direction.

In order to avoid making a lot of copies of the same direction vector, a static array of the STEP names of six common direction vectors ((1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), and (0,0,-1)) is maintained. Every time make_direction is called, a check is made to see if the direction to be made is one of those, and, if so, whether it is already named. If the direction is one of those and is already named, the existing one is retrieved by name from the product and returned. If the direction of one of those but is not already named, it is made, its name is stored, and it is returned. If the direction is not one of those, it is made and returned.

make_ellipse

This function takes (i) a STEP Working Form Product (product), (ii) an array of three doubles representing a cartesian point (point) at the center of the ellipse, (iii) two arrays of three doubles representing two unit vectors (axis1 and axis2) which give the directions of the major and minor radii of the ellipse, and (iv) two doubles (major and minor) representing the length of the major and minor radii of the ellipse. It makes a STEP axis2_placement (with an x-direction) and two real number objects and uses them to make the ellipse.

The STEP ellipse needs the normal to the plane of the ellipse, so that is generated as the cross product of axis1 and axis2. A check is made to be sure the length of this is very near one.

The STEP ellipse is returned.

make_extras_for_face

This function takes the tag (parasolid_surface) of a Parasolid surface and a STEP Working Form Product (product). The face on which this function works is known to have no loops in Parasolid, which is possible only if the surface is an entire sphere or torus. This function checks the surface type and returns a null object if the surface is not a sphere or torus.

If the surface is a sphere, this function makes (i) a point at the north pole, (ii) a vertex on the point, (iii) a vertex_loop on the vertex, (iv) a true loop_logical_structure on the vertex_loop, and (v) a list of the one loop_logical_structure, which is returned but not added to the product.

If the surface is a torus, it calls make_extras_for_torus, which returns a list of two loop_logical_structures. The list is returned.

make_extras_for_torus

This function takes the tag (parasolid_surface) of a Parasolid surface which is known to be a torus, and a STEP Product (product). The face on the surface is known to have no loops in Parasolid.

The function puts a loop around the large equator of the torus, so that the torus can be mapped to a plane. It makes (1) a point on the circle (by calling pick_point_on_circle and make_point) (2) a vertex on the point, (3) a circle around the large equator (by calling make_circle), (4) a curve_logical_structure on the circle, (5) an edge on the curve_logical_structure with its ends at the vertex, (6) an edge_logical_structure on the edge, (7) a list object of the edge_logical_structure (8) an edge_loop on the list object, (9) a true loop_logical_structure on the edge_loop, (10) a false loop_logical_structure on the edge_loop, and (11) a list object of the two loop_logical_structures, which is returned but is not added to the product.

make_line

This function takes a STEP Working Form Product (product) and two arrays of three doubles representing a cartesian point (point) which is the location of a point on the line and a vector (axis) which gives the direction of the line. It makes a STEP point and a STEP direction and uses them to make a line.

The STEP line is returned.

make_plane

This function takes a STEP Working Form Product (product) and two arrays of three doubles representing a point on the plane (point) and a normal to the plane (normal). It makes an axis2_placement with no reference direction for the x-axis, and uses it to make the plane, which is returned.

make_point

This function takes a STEP Working Form Product (product), an array of three doubles representing a point on the plane (point), and the tag (parasolid_point) of a Parasolid point. The tag may be null. The function makes a STEP point and returns it.

make_sphere

This function takes a STEP Working Form Product (product), an array of three doubles representing the point at the center of the sphere (point), and a double (radius) representing the radius of the sphere. It makes an axis2_placement with a vertical axis and no reference direction for the x-axis, and uses it as the position. Then the spherical_surface is made and returned.

make_torus

This function takes a STEP Working Form Product (product), two arrays of three doubles representing a point on the axis (point) and a unit vector in the direction of the axis (unit_vector), a double (major_radius) representing the major radius of the torus, and a double (minor_radius) representing the minor radius of the torus. It makes an axis2_placement with no reference direction for the x-axis, and uses it as the axis. Then the toroidal_surface is made and returned.

make_vertex_on_curve

This function takes a STEP Working Form Product (product), and the tag (parasolid_curve) of a curve which is a circle or an ellipse. It makes a vertex on the curve. For an ellipse, the point is the one at the end of the minor radius. For a circle, the point is found by calling pick_point_on_circle.

The function makes a STEP point at the proper location, makes a STEP vertex on the point, and returns the STEP vertex.

MODELn

This “function” is used to make error-handling invisible in the text of PTS functions. It is actually a series of compiler macros defined for values of n from 0 through 7. MODELn calls a Parasolid function (call it “foo”) which always has a void returned value. The value of n is one less than the number of arguments to foo (the last argument to foo is always “ifail,” which is not included in the arguments to MODELn). The first argument to MODELn is foo, and the rest of the arguments are the arguments to foo. MODELn calls foo. Then it calls “report_ifail,” which prints an error message and returns a non-zero value if foo executed improperly. If report_ifail returns a non-zero value, MODELn returns an error return value specified by the function in which the MODELn occurs.

In order for MODELn to work, every function which contains a MODELn must have (i) a local variable “function_name,” the value of which is a string that should be the function name, (ii) a local variable “ifail,” the value of which is a Parasolid IFAIL, and (iii) a local variable “error_return,” the value of which has the same type as the type returned by the function. The names of these three variables are built into MODELn.

MODELn has a counterpart named CALLn for calling PTS functions.

pick_point_on_circle

This function takes three arrays of three doubles (center, axis, and point) and a double (radius). The center, axis and radius are those of a circle. The function finds a point on the circle and copies its coordinates into point. The axis is supposed to be a unit vector. If it is, the function returns 1. If not, it returns zero.

A unit vector (radial) is found perpendicular to axis. Then the point on the circle is found by moving one radius from the center of the circle in the direction of radial.

The algorithm for finding the components of radial is to pick a coordinate axis for which the component of axis has a length of 0.5 or more (there must be one or the length of axis could not be 1). The order of preference for the coordinate axis is z, x, y. Then one of the components of radial (not the one on the selected coordinate axis) may be arbitrarily set to zero, and the other two components may be found by writing two simultaneous equations which say that the dot product of radial with the axis must be zero and the length of radial must be 1.

This method deals with the case of axis lying on one of the principal axes, and protects against numerical instability in case axis is very close to one of the principal axes.

print_entity

This function takes a non-null STEP Object (value), a file pointer (file), an integer (places) representing the number of decimal places to which real numbers should be rounded, and an integer (is_attribute) which is a flag indicating whether an entity should be printed by using its reference number (if it has one). The function prints a STEP representation of the Object in the file.

The function does not return anything.

print_file

This function takes a STEP Working Form Product (product) and a file pointer (file) as arguments. It asks the user to specify the number of decimal places for printing real numbers and whether trailing zeros should be suppressed. Then it calls the functions print_file_header, print_file_data, and print_file_trailer to do what their names indicate. The options are passed to print_file_data. The file is printed in STEP physical file format.

The function does not return anything.

print_file_data

This function takes a STEP Working Form Product (product), a file pointer (file), an integer (places) specifying the number of decimal places for printing real numbers, and an integer (suppress) which indicates whether trailing zeros should be suppressed. The function prints the data section of a STEP physical file representing the product. The objects in the model are printed in the same order in which they were created.

The function does not return anything.

The STEP Working Form function STEPprint was used as the point of departure.

print_file_header

This function takes a STEP Working Form Product (product), a file pointer (file), and a string (description) describing the nature of the STEP file being printed, and prints a perfunctory file header.

The function does not return anything.

print_file_trailer

This function takes a STEP Working Form Product (product) and a file pointer (file) for a STEP file which is almost complete, and prints the last few lines of the file.

The function does not return anything.

print_object

This function takes a STEP Object (value), a file pointer (file), an integer (places) representing the number of decimal places to which real numbers should be rounded, and an integer (is_attribute) which is a flag indicating whether an entity should be printed by using its reference number (if it has one). The function prints a STEP representation of the Object in the file.

The function does not return anything.

print_real

This function takes a real number (value), a file pointer (file), an integer (places) representing the number of decimal places to which real numbers should be rounded, and a integer flag (suppress) indicating whether trailing zeros are to be suppressed. Places must be a positive integer not greater than 16. The function prints the number in the file, unless places is out of bounds, in which case it does nothing.

If the suppress flag is non-zero, trailing zeros are suppressed, except for one immediately after the decimal point, if there is one there. Otherwise, zeros are used as necessary to fill up the number to the indicated number of places.

The function does not return anything.

report_ifail

This function takes the name (name) of a Parasolid function, the name (caller) of the function that called the Parasolid function, and a failure code (ifail). The function prints an error message and returns true (1) if ifail is not zero. Otherwise, it returns false (0).

start_modeler

This function takes a pointer (world) to a tag and starts the Parasolid modeler by calling STAMOD with appropriate arguments. It fills in the value of world with the tag of the world which is created. It prints the modeler version number and returns true (1) if the STAMOD works. Otherwise it prints an error message and returns false (0).

It tells the modeler to use the journal file named “para_to_step.”

This function was copied from the Parasolid manual with minor changes.

The user field argument of Parasolid entities is set to 3 because three user fields are used, as described in section 5.2.2.

stop_modeler

This function takes no arguments. It stops the Parasolid modeler by calling STOMOD with appropriate arguments. It returns true (1) if STOMOD works. Otherwise it prints an error message and returns false (0).

This function was copied from the Parasolid manual with minor changes.

8 Other Comments

8.1 Observations on Parasolid

Parasolid appears to be a robust system. It never broke down while PTS was being built and did not make any detectable errors.

The Parasolid extraction routines (identifying connected entities, ascertaining the type of an entity, outputting data regarding a curve, etc.) are well-suited to the use being made of them by PTS.

8.2 Testing Parasolid-to-STEP

The Parasolid-to-STEP system has been tested by using it to convert a few dozen Parasolid transmit files (including five sent to NIST by a private firm) to STEP physical files. PTS worked as intended on all these files and has no known bugs. It is believed to be robust.

8.3 Building Systems Similar to Parasolid-to-STEP

Building a STEP to Parasolid system (the reverse of PTS) that runs from a STEP boundary representation appears to be possible but much more difficult because the Parasolid input routines do not allow building bits and pieces of topology. Parasolid provides input routines that make it easy to define primitive solids and swept solids and to do boolean operations. Thus, converting a STEP constructive solid geometry (CSG) model to a Parasolid model would probably be straightforward. Similarly, since many features in the STEP Form Features Information Model are much like simple solids, building a Parasolid model from STEP form features is also a more tractable problem. The Off-Line Programming system developed by the author [Kramer91] does this for a limited range of feature types.

It might be feasible to print a Parasolid transmit file directly from a STEP Working Form Product (or possibly even directly from a STEP boundary representation file without using the Working Form), without using the Parasolid system if the format of Parasolid transmit files were known.

The heart of the conversion process in PTS is the “convert_region” function, which converts a Parasolid body in active memory to a STEP Working Form product in active memory. It would be easy to include this function in other systems, providing the capability, for example, of building a STEP Working Form by reading in a Parasolid transmit file and then processing the Working Form further rather than just printing it out. As another example, the function could be used to print out a STEP file from a Parasolid model in active memory, rather than having to read in a transmit file.

As may be seen in Figure 3, all the functions having to do with printing a STEP physical file hang from a single branch of the PTS function call hierarchy. The entire branch could be replaced simply by substituting some other version of “print_file,” if some other form of output were desired.

The current PTS system could be strengthened by adding more curve and surface types, parametric curves, intersection curves, and swept surfaces, for example. This could be done by adding capabilities to the system without having to make structural changes. Most of the existing functions would not have to be changed at all.

The STEP standard is still evolving. The geometry and topology schemas and the rules for mapping a schema to a physical file in the latest draft are somewhat different from the Tokyo version (on which PTS is based). Updating PTS to conform to the current standard should be straightforward and require only a few days.

It would probably be feasible to build a system that prints a STEP file from a Parasolid model in active memory without calling on the Fed-X or STEP Working Form functions at all. The latter would be replaced wherever they appear in PTS with printing functions. This would have the advantage of being independent of the other systems and the disadvantage of being harder to update in case of changes to STEP.

8.4 Schema Dependence

Parasolid-to-STEP requires a STEP schema as input only because Fed-X, and therefore STEPparse, requires a schema. Fed-X and the STEP Working Form functions are schema independent. They will run with any valid schema and any valid physical file based on the schema. This is possible because they do not process semantic knowledge of the subject matter of the schema or physical file.

PTS, on the other hand, is not schema independent. Although it requires a schema as input, it requires entity types to be defined in the schema as described in section 3.3. PTS cannot be made schema independent because it is based on specific semantic knowledge: how geometry and topology can be used in two slightly different ways to give the boundary representation of a solid object. This knowledge is implicit in the functions that comprise PTS. It would be possible to eliminate the requirement that the schema exist before the call to PTS is made by having PTS print out the schema it wants when it starts up and feed it to Fed-X, but this has not been done.

One design decision in building PTS involves schema dependence. The STEP Working Form functions provide alternate methods of building entities. It is possible (i) to create an empty entity of a given type and then assign values to the attributes of the entity, or (ii) to create an entity of a given type (with the values of all attributes specified) from an ordered list of values of the entity’s attributes. In the first method, the order of the list of attributes does not need to be known. In the second method, the names of the attributes do not need to be known. Thus, the two methods provide different kinds of protection from changes in the schema. The first protects against changes in attribute order, the second against changes in attribute name. Neither protects against changes that add attributes or that change the meaning of an attribute. The second method is used in PTS.

8.5 Miscellany

Parasolid-to-STEP undertakes almost no memory management. It allocates itself a lot of memory to build structures (mainly lists) which it could give back to the operating system by freeing them when they are no longer needed. It does not do this. This has not been a problem for any of the test parts. Because the amount of reusable memory wasted by PTS is not large compared with the amount needed through the end of a call to PTS, memory management in PTS would be useful only in the case of a conversion that was slightly too big for the system without it - a small percentage of likely inputs. If the functions of PTS were to be incorporated in some other system that had a longer lifetime and included memory management itself, however, adding memory management to the PTS functions would be very desirable.

The Fed-X and STEP Working Form software built by Stephen N. Clark work almost flawlessly and comprise a generally impressive system.

PTS was built using release 2.2 of Parasolid. It was rebuilt using release 3.1. The timing data given in section 4.3 of this paper were taken using release 3.1. No changes in the PTS source code were required (or made) to accommodate the use of release 3.1.

A References

[Altemueller88a] Altemueller, J., The STEP File Structure, ISO TC184/SC4/WG1 Document N279, September, 1988

[Altemueller88b] Altemueller, J., Mapping from Express to Physical File Structure, ISO TC184/SC4/WG1 Document N280, September, 1988

[ANSI89] American National Standards Institute, Programming Language C, Document ANSI X3.159-1989

[Clark90a] Clark, S. N., An Introduction to the NIST PDES Toolkit, National Institute of Standards and Technology, Interagency Report 4336, May 1990, 7 p.

[Clark90b] Clark, S.N., Fed-X: The NIST Express Translator, National Institute of Standards and Technology, Interagency Report 4371, July 1990, 11 p.

[Clark90c] Clark, S.N., NIST Express Working Form Programmer's Reference, National Institute of Standards and Technology, Interagency Report 4407, September 1990, 51 p.

[Clark90d] Clark, S.N., The NIST Working Form for STEP, National Institute of Standards and Technology, Interagency Report 4351, June 1990, 6 p.

[Clark90e] Clark, S.N., NIST STEP Working Form Programmer's Reference, National Institute of Standards and Technology, Interagency Report 4353, June 1990, 23 p.

[Clark90f] Clark, S.N., The NIST PDES Toolkit: Technical Fundamentals, National Institute of Standards and Technology, Interagency Report 4335, May 1990, 24 p.

[Clark90g] Clark, S.N., QDES Administrative Guide, National Institute of Standards and Technology, Interagency Report 4334, May 1990, 12 p.

[Clark90h] Clark, S.N., QDES User's Guide, National Institute of Standards and Technology, Interagency Report 4361, June 1990, 35 p.

[Kramer91] Kramer, T.R., The Off-Line Programming System (OLPS): A Prototype STEP-Based NC-Program Generator, proceedings of a seminar *Product Data Exchange for the 1990's*, New Orleans, Louisiana, NCGA, February 1991, Vol. 2

[McLay90] McLay, M. J., and Morris, K.C., The NIST STEP Class Library, National Institute of Standards and Technology, Interagency Report 4411, August 1990, 20 p.

[Morris90] Morris, K.C., Translating Express to SQL: A User’s Guide, National Institute of Standards and Technology, Interagency Report 4341, May 1990, 17 p.

[Nickerson90] Nickerson, D., The NIST Database Loader: STEP Working Form to SQL, National Institute of Standards and Technology, Interagency Report 4337, May 1990, 7 p.

[Schenck90] Schenck, D., ed., Exchange of Product Model Data - Part 11: The Express Language, ISO TC184/SC4 Document N64, July 1990

[Shape90] Shape Data Limited, Parasolid v3.0 Reference Manual, Shape Data Limited, Cambridge, England, June 1990

[Smith88] Smith, B., and G. Rinaudot, eds., Product Data Exchange Specification First Working Draft, National Institute of Standards and Technology, Interagency Report 88-4004, December 1988

[Weick90] Weick, W., Application Protocol for the Data-Transfer of STEP B_Rep Models Via a Physical File, Version 2.0, PROCAD GmbH, Karlsruhe, Germany, September 1990

B STEP File for Test Part

This file is a STEP boundary representation of the part shown in Figure 2. It was produced by Parasolid-to-STEP.

```
STEP;

HEADER;
 FILE_IDENTIFICATION(
 'PARASOLID TO STEP',
 'some date',
 ('Noman'),
 ('NIST'),
 '1.0',
 '1.0',
 'originating system');
 FILE_DESCRIPTION('This file was produced by the
NIST parasolid_to_step tool.');
 IMP_LEVEL('1.0');
ENDSEC;

DATA;
@1 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@2 = DIRECTION(, 0.0, 0.0, 1.0);
@3 = AXIS2_PLACEMENT(, #1, #2, );
@4 = CIRCLE(, 1.0, #3);
@5 = CURVE_LOGICAL_STRUCTURE(#4, .T.);
@6 = CARTESIAN_POINT(, 1.0, 0.0, 7.0);
@7 = VERTEX(#6);
@8 = EDGE(#7, #7, #5);
@9 = EDGE_LOGICAL_STRUCTURE(#8, .T.);
@10 = EDGE_LOOP((#9));
@11 = LOOP_LOGICAL_STRUCTURE(#10, .T.);
@12 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@13 = AXIS2_PLACEMENT(, #12, #2, );
@14 = SPHERICAL_SURFACE(, 1.0, #13);
@15 = SURFACE_LOGICAL_STRUCTURE
        (#14, .F.);
@16 = FACE(, (#11), #15);
@17 = FACE_LOGICAL_STRUCTURE(#16, .T.);
@18 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@19 = AXIS2_PLACEMENT(, #18, #2, );
@20 = CIRCLE(, 4.5, #19);
@21 = CURVE_LOGICAL_STRUCTURE(#20, .T.);
@22 = CARTESIAN_POINT(, 4.5, 0.0, 7.0);
@23 = VERTEX(#22);
@24 = EDGE(#23, #23, #21);
@25 = EDGE_LOGICAL_STRUCTURE(#24, .F.);
@26 = EDGE_LOOP((#25));
@27 = LOOP_LOGICAL_STRUCTURE(#26, .T.);
@28 = CARTESIAN_POINT(, -5.0, 0.0, 7.0);
@29 = DIRECTION(, 0.0, 1.0, 0.0);
@30 = LINE(, #28, #29);
@31 = CURVE_LOGICAL_STRUCTURE(#30, .T.);
@32 = CARTESIAN_POINT(, -5.0, -6.244998, 7.0);
@33 = VERTEX(#32);
@34 = CARTESIAN_POINT(, -5.0, 6.244998, 7.0);
@35 = VERTEX(#34);
@36 = EDGE(#33, #35, #31);
@37 = EDGE_LOGICAL_STRUCTURE(#36, .F.);
@38 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@39 = DIRECTION(, 0.0, 0.0, -1.0);
@40 = AXIS2_PLACEMENT(, #38, #39, );
@41 = CIRCLE(, 8.0, #40);
@42 = CURVE_LOGICAL_STRUCTURE(#41, .T.);
@43 = CARTESIAN_POINT(, 5.0, -6.244998, 7.0);
@44 = VERTEX(#43);
@45 = EDGE(#44, #33, #42);
@46 = EDGE_LOGICAL_STRUCTURE(#45, .F.);
@47 = CARTESIAN_POINT(, 5.0, 0.0, 7.0);
@48 = DIRECTION(, 0.0, -1.0, 0.0);
@49 = LINE(, #47, #48);
@50 = CURVE_LOGICAL_STRUCTURE(#49, .T.);
@51 = CARTESIAN_POINT(, 5.0, 6.244998, 7.0);
@52 = VERTEX(#51);
@53 = EDGE(#52, #44, #50);
@54 = EDGE_LOGICAL_STRUCTURE(#53, .F.);
@55 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@56 = AXIS2_PLACEMENT(, #55, #39, );
@57 = CIRCLE(, 8.0, #56);
@58 = CURVE_LOGICAL_STRUCTURE(#57, .T.);
@59 = EDGE(#35, #52, #58);
@60 = EDGE_LOGICAL_STRUCTURE(#59, .F.);
@61 = EDGE_LOOP((#37, #46, #54, #60));
@62 = LOOP_LOGICAL_STRUCTURE(#61, .T.);
@63 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@64 = AXIS2_PLACEMENT(, #63, #2, );
@65 = PLANE(, #64);
@66 = SURFACE_LOGICAL_STRUCTURE
        (#65, .T.);
@67 = FACE(#62, (#27, #62), #66);
@68 = FACE_LOGICAL_STRUCTURE(#67, .T.);
@69 = EDGE_LOGICAL_STRUCTURE(#36, .T.);
@70 = CARTESIAN_POINT(, 0.0, 0.0, 12.0);
@71 = DIRECTION(, -0.707107, 0.0, 0.707107);
@72 = DIRECTION(, 0.707107, 0.0, 0.707107);
@73 = AXIS2_PLACEMENT(, #70, #71, #72);
@74 = ELLIPSE(, 11.313708, 8.0, #73);
@75 = CURVE_LOGICAL_STRUCTURE(#74, .T.);
@76 = EDGE(#35, #33, #75);
@77 = EDGE_LOGICAL_STRUCTURE(#76, .T.);
@78 = EDGE_LOOP((#69, #77));
@79 = LOOP_LOGICAL_STRUCTURE(#78, .T.);
@80 = CARTESIAN_POINT(, -5.0, 0.0, 7.0);
@81 = DIRECTION(, 0.707107, 0.0, -0.707107);
```

```
@82 = AXIS2_PLACEMENT(, #80, #81, );
@83 = PLANE(, #82);
@84 = SURFACE_LOGICAL_STRUCTURE
        (#83, .F.);
@85 = FACE(#79, (#79), #84);
@86 = FACE_LOGICAL_STRUCTURE(#85, .T.);
@87 = EDGE_LOGICAL_STRUCTURE(#53, .T.);
@88 = CARTESIAN_POINT(, 0.0, 0.0, 12.0);
@89 = DIRECTION(, 0.707107, -0.0, 0.707107);
@90 = DIRECTION(, -0.707107, 0.0, 0.707107);
@91 = AXIS2_PLACEMENT(, #88, #89, #90);
@92 = ELLIPSE(, 11.313708, 8.0, #91);
@93 = CURVE_LOGICAL_STRUCTURE(#92, .T.);
@94 = EDGE(#44, #52, #93);
@95 = EDGE_LOGICAL_STRUCTURE(#94, .T.);
@96 = EDGE_LOOP((#87, #95));
@97 = LOOP_LOGICAL_STRUCTURE(#96, .T.);
@98 = CARTESIAN_POINT(, 5.0, 0.0, 7.0);
@99 = DIRECTION(, -0.707107, 0.0, -0.707107);
@100 = AXIS2_PLACEMENT(, #98, #99, );
@101 = PLANE(, #100);
@102 = SURFACE_LOGICAL_STRUCTURE
        (#101, .F.);
@103 = FACE(#97, (#97), #102);
@104 = FACE_LOGICAL_STRUCTURE(#103, .T.);
@105 = CARTESIAN_POINT(, 0.0, 0.0, 0.0);
@106 = AXIS2_PLACEMENT(, #105, #2, );
@107 = CIRCLE(, 8.0, #106);
@108 = CURVE_LOGICAL_STRUCTURE(#107, .T.);
@109 = CARTESIAN_POINT(, 8.0, 0.0, 0.0);
@110 = VERTEX(#109);
@111 = EDGE(#110, #110, #108);
@112 = EDGE_LOGICAL_STRUCTURE(#111, .F.);
@113 = EDGE_LOOP((#112));
@114 = LOOP_LOGICAL_STRUCTURE(#113, .T.);
@115 = CARTESIAN_POINT(, 0.0, 0.0, -7.0);
@116 = VERTEX(#115);
@117 = VERTEX_LOOP(#116);
@118 = LOOP_LOGICAL_STRUCTURE(#117, .T.);
@119 = CARTESIAN_POINT(, 0.0, 0.0, -7.0);
@120 = AXIS2_PLACEMENT(, #119, #39, );
@121 = CONICAL_SURFACE(, 0.851966, 0.0, #120);
@122 = SURFACE_LOGICAL_STRUCTURE
        (#121, .T.);
@123 = FACE(#114, (#114, #118), #122);
@124 = FACE_LOGICAL_STRUCTURE(#123, .T.);
@125 = EDGE_LOGICAL_STRUCTURE(#76, .F.);
@126 = EDGE_LOGICAL_STRUCTURE(#59, .T.);
@127 = EDGE_LOGICAL_STRUCTURE(#94, .F.);
@128 = EDGE_LOGICAL_STRUCTURE(#45, .T.);
@129 = EDGE_LOOP((#125, #126, #127, #128));
@130 = LOOP_LOGICAL_STRUCTURE(#129, .T.);
@131 = EDGE_LOGICAL_STRUCTURE(#111, .T.);
@132 = EDGE_LOOP((#131));
@133 = LOOP_LOGICAL_STRUCTURE(#132, .T.);
@134 = CARTESIAN_POINT(, 0.0, 0.0, 0.0);
@135 = AXIS2_PLACEMENT(, #134, #2, );
@136 = CYLINDRICAL_SURFACE(, 8.0, #135);
@137 = SURFACE_LOGICAL_STRUCTURE
        (#136, .T.);
@138 = FACE(, (#130, #133), #137);
@139 = FACE_LOGICAL_STRUCTURE(#138, .T.);
@140 = EDGE_LOGICAL_STRUCTURE(#8, .F.);
@141 = EDGE_LOOP((#140));
@142 = LOOP_LOGICAL_STRUCTURE(#141, .T.);
@143 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@144 = AXIS2_PLACEMENT(, #143, #39, );
@145 = CIRCLE(, 2.5, #144);
@146 = CURVE_LOGICAL_STRUCTURE(#145, .T.);
@147 = CARTESIAN_POINT(, -2.5, 0.0, 7.0);
@148 = VERTEX(#147);
@149 = EDGE(#148, #148, #146);
@150 = EDGE_LOGICAL_STRUCTURE(#149, .F.);
@151 = EDGE_LOOP((#150));
@152 = LOOP_LOGICAL_STRUCTURE(#151, .T.);
@153 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@154 = AXIS2_PLACEMENT(, #153, #2, );
@155 = PLANE(, #154);
@156 = SURFACE_LOGICAL_STRUCTURE
        (#155, .T.);
@157 = FACE(#152, (#142, #152), #156);
@158 = FACE_LOGICAL_STRUCTURE(#157, .T.);
@159 = EDGE_LOGICAL_STRUCTURE(#24, .T.);
@160 = EDGE_LOOP((#159));
@161 = LOOP_LOGICAL_STRUCTURE(#160, .T.);
@162 = EDGE_LOGICAL_STRUCTURE(#149, .T.);
@163 = EDGE_LOOP((#162));
@164 = LOOP_LOGICAL_STRUCTURE(#163, .T.);
@165 = CARTESIAN_POINT(, 0.0, 0.0, 7.0);
@166 = AXIS2_PLACEMENT(, #165, #2, );
@167 = TOROIDAL_SURFACE(, 3.5, 1.0, #166);
@168 = SURFACE_LOGICAL_STRUCTURE
        (#167, .F.);
@169 = FACE(, (#161, #164), #168);
@170 = FACE_LOGICAL_STRUCTURE(#169, .T.);
@171 = CLOSED_SHELL
        ((#17, #68, #86, #104, #124, #139, #158, #170));
@172 = SHELL_LOGICAL_STRUCTURE(#171, .T.);
@173 = REGION(#172, (#172));
ENDSEC;

ENDSTEP;
```

C STEP File for Test Part (COMPRESSED)

This file is a STEP boundary representation of the part shown in Figure 2. It was produced by running the file shown in Appendix B through the “compress-step” facility.

```
STEP;

HEADER;
 FILE_IDENTIFICATION
  (‘PARASOLID TO STEP’,
   ‘some date’,
   (‘Noman’),
   (‘NIST’),
   ‘1.0’,
   ‘1.0’,
   ‘originating system’);
 FILE_DESCRIPTION
  (‘This file was produced by the NIST
                    parasolid_to_step tool.’);
 IMP_LEVEL
  (‘1.0’);
ENDSEC;

DATA;
@1 = CARTESIAN_POINT (, -5.0, -6.245, 7.0);
@2 = CARTESIAN_POINT (, -5.0, 0.0, 7.0);
@3 = CARTESIAN_POINT (, -5.0, 6.245, 7.0);
@4 = CARTESIAN_POINT (, -2.5, 0.0, 7.0);
@5 = CARTESIAN_POINT (, 0.0, 0.0, -7.0);
@6 = CARTESIAN_POINT (, 0.0, 0.0, 0.0);
@7 = CARTESIAN_POINT (, 0.0, 0.0, 7.0);
@8 = CARTESIAN_POINT (, 0.0, 0.0, 12.0);
@9 = CARTESIAN_POINT (, 1.0, 0.0, 7.0);
@10 = CARTESIAN_POINT (, 4.5, 0.0, 7.0);
@11 = CARTESIAN_POINT (, 5.0, -6.245, 7.0);
@12 = CARTESIAN_POINT (, 5.0, 0.0, 7.0);
@13 = CARTESIAN_POINT (, 5.0, 6.245, 7.0);
@14 = CARTESIAN_POINT (, 8.0, 0.0, 0.0);
@15 = DIRECTION (, -0.70711, 0.0, -0.70711);
@16 = DIRECTION (, -0.70711, 0.0, 0.70711);
@17 = DIRECTION (, 0.0, -1.0, 0.0);
@18 = DIRECTION (, 0.0, 0.0, -1.0);
@19 = DIRECTION (, 0.0, 0.0, 1.0);
@20 = DIRECTION (, 0.0, 1.0, 0.0);
@21 = DIRECTION (, 0.70711, 0.0, -0.70711);
@22 = DIRECTION (, 0.70711, 0.0, 0.70711);
@23 = AXIS2_PLACEMENT (, #2, #21, );
@24 = AXIS2_PLACEMENT (, #5, #18, );
@25 = AXIS2_PLACEMENT (, #6, #19, );
@26 = AXIS2_PLACEMENT (, #7, #18, );
@27 = AXIS2_PLACEMENT (, #7, #19, );
@28 = AXIS2_PLACEMENT (, #8, #16, #22);
@29 = AXIS2_PLACEMENT (, #8, #22, #16);
@30 = AXIS2_PLACEMENT (, #12, #15, );
@31 = LINE (, #2, #20);
@32 = LINE (, #12, #17);
@33 = CIRCLE (, 1.0, #27);
@34 = CIRCLE (, 2.5, #26);
@35 = CIRCLE (, 4.5, #27);
@36 = CIRCLE (, 8.0, #25);
@37 = CIRCLE (, 8.0, #26);
@38 = ELLIPSE (, 11.31371, 8.0, #28);
@39 = ELLIPSE (, 11.31371, 8.0, #29);
@40 = CONICAL_SURFACE (, 0.85197, 0.0, #24);
@41 = CYLINDRICAL_SURFACE (, 8.0, #25);
@42 = PLANE (, #23);
@43 = PLANE (, #27);
@44 = PLANE (, #30);
@45 = SPHERICAL_SURFACE (, 1.0, #27);
@46 = TOROIDAL_SURFACE (, 3.5, 1.0, #27);
@47 = VERTEX (#1);
@48 = VERTEX (#3);
@49 = VERTEX (#4);
@50 = VERTEX (#5);
@51 = VERTEX (#9);
@52 = VERTEX (#10);
@53 = VERTEX (#11);
@54 = VERTEX (#13);
@55 = VERTEX (#14);
@56 = CURVE_LOGICAL_STRUCTURE (#31, .T.);
@57 = CURVE_LOGICAL_STRUCTURE (#32, .T.);
@58 = CURVE_LOGICAL_STRUCTURE (#33, .T.);
@59 = CURVE_LOGICAL_STRUCTURE (#34, .T.);
@60 = CURVE_LOGICAL_STRUCTURE (#35, .T.);
@61 = CURVE_LOGICAL_STRUCTURE (#36, .T.);
@62 = CURVE_LOGICAL_STRUCTURE (#37, .T.);
@63 = CURVE_LOGICAL_STRUCTURE (#38, .T.);
@64 = CURVE_LOGICAL_STRUCTURE (#39, .T.);
@65 = EDGE (#47, #48, #56);
@66 = EDGE (#48, #47, #63);
@67 = EDGE (#48, #54, #62);
@68 = EDGE (#49, #49, #59);
@69 = EDGE (#51, #51, #58);
@70 = EDGE (#52, #52, #60);
@71 = EDGE (#53, #47, #62);
@72 = EDGE (#53, #54, #64);
@73 = EDGE (#54, #53, #57);
@74 = EDGE (#55, #55, #61);
@75 = EDGE_LOGICAL_STRUCTURE (#65, .T.);
@76 = EDGE_LOGICAL_STRUCTURE (#65, .F.);
@77 = EDGE_LOGICAL_STRUCTURE (#66, .T.);
@78 = EDGE_LOGICAL_STRUCTURE (#66, .F.);
@79 = EDGE_LOGICAL_STRUCTURE (#67, .T.);
```

```
@80 = EDGE_LOGICAL_STRUCTURE (#67, .F.);
@81 = EDGE_LOGICAL_STRUCTURE (#68, .T.);
@82 = EDGE_LOGICAL_STRUCTURE (#68, .F.);
@83 = EDGE_LOGICAL_STRUCTURE (#69, .T.);
@84 = EDGE_LOGICAL_STRUCTURE (#69, .F.);
@85 = EDGE_LOGICAL_STRUCTURE (#70, .T.);
@86 = EDGE_LOGICAL_STRUCTURE (#70, .F.);
@87 = EDGE_LOGICAL_STRUCTURE (#71, .T.);
@88 = EDGE_LOGICAL_STRUCTURE (#71, .F.);
@89 = EDGE_LOGICAL_STRUCTURE (#72, .T.);
@90 = EDGE_LOGICAL_STRUCTURE (#72, .F.);
@91 = EDGE_LOGICAL_STRUCTURE (#73, .T.);
@92 = EDGE_LOGICAL_STRUCTURE (#73, .F.);
@93 = EDGE_LOGICAL_STRUCTURE (#74, .T.);
@94 = EDGE_LOGICAL_STRUCTURE (#74, .F.);
@95 = EDGE_LOOP ((#75, #77));
@96 = EDGE_LOOP ((#76, #88, #92, #80));
@97 = EDGE_LOOP ((#78, #79, #90, #87));
@98 = EDGE_LOOP ((#81));
@99 = EDGE_LOOP ((#82));
@100 = EDGE_LOOP ((#83));
@101 = EDGE_LOOP ((#84));
@102 = EDGE_LOOP ((#85));
@103 = EDGE_LOOP ((#86));
@104 = EDGE_LOOP ((#91, #89));
@105 = EDGE_LOOP ((#93));
@106 = EDGE_LOOP ((#94));
@107 = VERTEX_LOOP (#50);
@108 = LOOP_LOGICAL_STRUCTURE (#95, .T.);
@109 = LOOP_LOGICAL_STRUCTURE (#96, .T.);
@110 = LOOP_LOGICAL_STRUCTURE (#97, .T.);
@111 = LOOP_LOGICAL_STRUCTURE (#98, .T.);
@112 = LOOP_LOGICAL_STRUCTURE (#99, .T.);
@113 = LOOP_LOGICAL_STRUCTURE (#100, .T.);
@114 = LOOP_LOGICAL_STRUCTURE (#101, .T.);
@115 = LOOP_LOGICAL_STRUCTURE (#102, .T.);
@116 = LOOP_LOGICAL_STRUCTURE (#103, .T.);
@117 = LOOP_LOGICAL_STRUCTURE (#104, .T.);
@118 = LOOP_LOGICAL_STRUCTURE (#105, .T.);
@119 = LOOP_LOGICAL_STRUCTURE (#106, .T.);
@120 = LOOP_LOGICAL_STRUCTURE (#107, .T.);
@121 = SURFACE_LOGICAL_STRUCTURE (#40, .T.);
@122 = SURFACE_LOGICAL_STRUCTURE (#41, .T.);
@123 = SURFACE_LOGICAL_STRUCTURE (#42, .F.);
@124 = SURFACE_LOGICAL_STRUCTURE (#43, .T.);
@125 = SURFACE_LOGICAL_STRUCTURE (#44, .F.);
@126 = SURFACE_LOGICAL_STRUCTURE (#45, .F.);
@127 = SURFACE_LOGICAL_STRUCTURE (#46, .F.);
@128 = FACE (, (#110, #118), #122);
@129 = FACE (, (#113), #126);
@130 = FACE (, (#115, #111), #127);
@131 = FACE (#108, (#108), #123);
@132 = FACE (#109, (#116, #109), #124);
@133 = FACE (#112, (#114, #112), #124);
@134 = FACE (#117, (#117), #125);
@135 = FACE (#119, (#119, #120), #121);
@136 = FACE_LOGICAL_STRUCTURE (#128, .T.);
@137 = FACE_LOGICAL_STRUCTURE (#129, .T.);
@138 = FACE_LOGICAL_STRUCTURE (#130, .T.);
@139 = FACE_LOGICAL_STRUCTURE (#131, .T.);
@140 = FACE_LOGICAL_STRUCTURE (#132, .T.);
@141 = FACE_LOGICAL_STRUCTURE (#133, .T.);
@142 = FACE_LOGICAL_STRUCTURE (#134, .T.);
@143 = FACE_LOGICAL_STRUCTURE (#135, .T.);
@144 = CLOSED_SHELL
   ((#137, #140, #139, #142, #143, #136, #141, #138));
@145 = SHELL_LOGICAL_STRUCTURE (#144, .T.);
@146 = REGION (#145, (#145));
ENDSEC;

ENDSTEP;
```