Chapter 8. Feature Naming and Annotation

Topic: *Feature Naming

Feature naming adds derived ANNOTATION entities to define (in more detail) the results of a modeling operation. Feature naming annotates entities which were created or removed as part of a modeling operation like blending or sweeping. It specifies what they are and what related entity created them, and annotates what relationship new entities have with the original entities.

Feature naming provides information that correlates the inputs and outputs of the specific modeling operation. For example, each edge of a sweep profile is associated with the lateral face generated. Because the input entities may not survive the operation, they are represented in the annotation entities indirectly through ATTRIB_TAG.

Annotation entities are kept in a list which is available through api_find_annotations and api_get_annotation_ctx. Developers may add application information by deriving from ATTRIB_TAG and pre-tagging the model. Copies of the existing ATTRIB_TAGs will be carried through to the final annotations.

Feature naming does not supply all of the answers about what occurs during a modeling operation. However, it does supply the hooks for a user-defined application to access so that it can acquire such information.

The information in the annotation entities depends on the modeling operation being annotated. For example, blending and sweeping each have a number of derived annotation types that are specific to these operations. Feature naming provides information on the original input entities, the generated output entities, the relationship between the input entities and the output entities, and what the output entities are. The annotations also flag the original entities so applications can query as to whether they exist or not after the operation.

Imagine a simple square (face) profile that is swept along a path to create a solid. The solid has eight new edges, five new faces, and four new vertices that were created to complete the solid. Four of the new edges are lateral edges and are associated with each of the four vertices in the square profile. The other four new edges are top edges and are associated with each of the four edges of the square profile. If this sweep operation used the draft option, there could be even more new edges, faces, and vertices.

For a blending example, imagine a rectangular block with a rounded chamfer to a single edge. After the blending operation, one edge and two vertices are removed completely from the model and are replaced by a new blend face, four new vertices, and four new edges. Two of the edges are cross-curves, or the rounded portion at the ends of the block. These are associated with the original two vertices. The other two edges are spring curves (sometimes called rail curves). The four new vertices connect together the four new edges, which in turn bound the new rounded chamfer blend face on the block.

Attaching Annotations

Topic: *Feature Naming

During the modeling operation, annotations are attached to the annotated entities through ATTRIB_ANNOTATION. This associates an ANNOTATION with its owning ENTITY and allows quick updating of the annotations as the operation progresses. These attributes are normally removed when the operation completes.

The ATTRIB_ANNOTATION instances are the specific ANNOTATION derived instances, such as SWEEP_ANNO_VERTEX_TOP. Because the ATTRIB_ANNOTATION does not normally persist, the only way to get the annotations is by using api_find_annotations.

The ATTRIB_ANNOTATION may be retained by setting the unhook_annotations option to FALSE (for example, when working in Scheme AIDE with annotations). The annotation option turns on the creation of the annotations in a modeling operation. If the unhook_annotations option is not turned off, the modeling operation automatically clears the ATTRIB_ANNOTATION, which would remove the handles and data that Scheme AIDE uses.

Acquiring Annotations

Topic: *Feature Naming

After a modeling operation, use the annotation class annotation_ctx to examine a list of the annotations. Annotations are added to the list by the annotation base class constructor and removed by the annotation base class lose method.

There is one session-wide annotation_ctx instance that is accessible through api_get_annotation_ctx. This global context will be used internally by all the annotation APIs, as well as the modeling APIs that are currently annotated. The annotation APIs that normally take a BULLETIN_BOARD parameter will ignore that parameter and will use the annotation list instead.

You can also acquire the list of the generated annotations from the annotation_ctx with api_find_annotations. This creates an entity list that can be handled and queried like any other entity list.

Query functions within ANNOTATION populate an ENTITY_LIST with the inputs or outputs. Each derived ANNOTATION type has more specific query functions for full detail. api_find_annotations permits searching for an annotation of a specific type. So, for example, to get the top vertex annotations from a sweep operation, the is_SWEEP_ANNO_VERTEX_TOP function can be passed as an argument.

Since annotations consume memory and could increase SAT file size, they are usually cleared manually before the next modeling operation, using the api_clear_annotations function.

When Entities Go Away

Topic: *Feature Naming

Depending on the modeling operation, input entities may or may not exist after an operation is complete. For example, blending operations typically remove vertices and edges and add new entities.

When such input entities no longer exist, they are still represented with an ATTRIB_TAG which contains a pointer to the original input entity and a flag indicating whether or not the entity has since been lost. ATTRIB_TAG is a base class that can be derived from and enhanced by a user-defined application. An application can use this to attach additional data to the input entities before an operation and have it carried forward to any annotations that reference it, even if the original entity is lost.

ATTRIB_TAG is designed to be derived from, so the application may store additional information there. Annotations contain a copy of the tag. If the application has not already tagged an input entity, the annotation will construct ATTRIB_TAGs as required. The utility functions get_actual_entity and get_actual_live_entity can simplify working with the tags.

Applications which derive from ATTRIB_TAG may want to set the error_no_input_tag option to TRUE to cause a sys_error if an input entity is not tagged. This is useful when the additional attribute information is necessary. It is also a useful debugging aid which helps to find places where the application did not tag all the inputs. If the error_no_input_tag option is FALSE, tags are simply generated on the fly as needed.

During the operation, annotations are attached to the inputs and outputs via ATTRIB_ANNOTATION. This attribute serves to pass on attribute notifications to the actual annotations and to let any copied ATTRIB_TAGs know if the input entity has been lost. They may optionally be kept after the operation as well. This is used in Scheme to support display of annotation information. That is, you can pick some topology and follow the attribute to see how it fit in a recent operation.

Feature Naming Theory

Topic: *Feature Naming

A "feature" is some operation that creates or modifies some geometry, or is some geometry that results from an operation. It is this resulting geometry that needs to be named, usually in terms of the inputs. An example of a feature is a sweep. The inputs are a list of edges that make up a profile, and another list of edges that represents a path to sweep along. There are several outputs, most noticeably a face for each edge of the profile. The feature modeler is interested in which profile and path edges "generated" each face.

Feature modelers need to determine and maintain dependency graphs so that edited features can be regenerated along with any dependent features. Annotations facilitate maintaining the dependency graph to a high level of granularity. For example, a sweep operation may involve many segment profiles. However, editing one segment of the profile may be more efficient in certain circumstances than regenerating the whole sweep.

Primitive geometry creation, such as edges from a sketcher, are named arbitrarily `e1`, `e2`, `e3`, etc. Secondary geometry, such as what is created by a sweep, is named by reference to the primitive geometry. For example, `Fs(e1, p1)` would be the face created by sweeping the profile edge `e1` along the path edge `p1`. `Es(e1, p1)` would be the edge bounding the far end of the sweep using the same inputs.

In a chained operation such as a sweep followed by a blend, the names generated in the early operation can be used in the later operation. For example, `Fb(Es(e1, p1))` indicates the face resulting from blending the edge bounding the far end of the sweep of `e1` along `p1`.

ACIS does not have any feature modeling software. It only has geometric modeling software, so applications must provide their own feature modeling engine.

Dependency Graph and Feature Naming

Topic: *Feature Naming

Annotations can be used to create dependency trees for knowing what features change when a given parameter changes. User interfaces may also want to highlight geometry created by a modeling operation, or geometry that depends on a proposed change or on the input geometry.

Annotations introduce a naming scheme for the elements of the modeling operation. Such naming schemes know the input geometry so that a given name can reference a specific element in the input geometry. Likewise, the naming schemes know about the output geometry. The output naming scheme are detailed enough to provide specific names, such as "left lateral edge" or "top edge" in a sweep.

Both the naming scheme and the dependencies are stored within the feature modeling engine. The new annotations are the source for some of the information that a feature modeling engine needs. The annotations are highly granular, noting both its place within the operation and its parent geometry.

Annotations by themselves do not provide all the answers as to the full set of geometry and the dependency information of a particular entity. The information can be built up incrementally when processing the annotations after an operation. Each time an entity is seen as a parent to another, the other entity can be added to its set of dependents. The set of dependents for each entity must be maintained by the feature modeling engine.

Testing

Topic: *Feature Naming

During development, you can use load “annotation.scm” to turn on a rubberbander that shows where annotations have been placed. As the mouse cursor is moved over the model, annotated entities are highlighted. The input and output entities are displayed in different colors, and a text description is written to the output window.

Unit tests can be developed using the annotation:assert Scheme extension. The basic idea is to turn on annotations, do an operation, and then assert that the annotations have been created.

Feature Naming Example

Topic: *Feature Naming

In this Scheme AIDE example, a profile is swept along a path to create a solid. Before the modeling operation, however, annotation option is turned on while the unhook_annotations option is turned off. This assures that the annotation data remains for the Scheme extensions to be able to query.

A simple sweep path is created for sweeping. Once the profile has been swept with annotations on, the annotation information can be queried. This is typically done by selecting an entity (such as an edge or face) of the resulting body and asking for a list of annotations using entity:annotations. This list of annotations can be further broken down into lists of inputs and outputs using annotation:input and annotation:output.

The annotation:member–name provides a small amount of information through the annotation as to what the owning entity is, such as being a “lateral face” from a sweep operation. entity:annotation–names can also be used to acquire the names of annotations in a list.

Scheme Example

```
; Turn on annotation options so that annotations are
; automatically created as part of the modeling operation.
(option:set ”annotation” #t)
;; #f
; Turn off the unhook annotation option so that termination
; of the modeling operation does not clear annotations.
(option:set ”unhook_annotations” #f)
;; #t

; Create a sweep path from a wire body of edges.
(define path1 (wire-body (list
    (edge:linear (position 0 0 0)(position 20 0 0))
    (edge:circular-3pt (position 20 0 0)
        (position 27 0 2.5) (position 30 0 10))
    (edge:linear (position 30 0 10)
        (position 30 0 20)))))
;; path1

; Sweep the edges into a face body.
(define a_body (sweep:law
    (edge:linear ( position 0 3 3)( position 0 3 -3))
    (edge:linear ( position 0 3 -3)
    ( position 0 -3 -3))))
;; a_body
; a_body => #[entity 8 1]

; Create the profile to use in sweeping
(define profile1 (car (entity:faces
    (sheet:2d a_body))))
;; profile1
; profile1 => #[entity 9 1]
; Sweep the profile along the path.
(define sweep1 (sweep:law profile1 path1))

; Create some lists for handling purposes.
; in_prof_edges are the edges of the input profile.
(define in_prof_edges (entity:edges profile1))
;; in_prof_edges
; out_sw_faces are the faces of the output swept model.
(define out_sw_faces (entity:faces sweep1))
;; out_sw_faces
; out_sw_edges are the edges of the output swept model.
(define out_sw_edges (entity:edges sweep1))
;; out_sw_edges
```

```
; Check the annotations attached to items.
; Create a list of annotations associated with an
; edge of the swept body.
; There are many edges associated with the output swept
; model. This only explores the annotations of the third
; edge.
(define anno1 (entity:annotations
    (list-ref out_sw_edges 2)))
; anno1 => (#[entity 93 1])
; Create a list of annotations that are inputs to
; the sweep operation.
(define anno_input1
    (annotation:inputs (list-ref anno1 0)))
;; anno_input1
; anno_input1 => (#[entity 94 1] #[entity 95 1])
; See what the annotation names are for a given
; input annotation.
(entity:annotation-names (list-ref anno_input1 0))
;; ("mid_top_vertex" "profile" "profile" "profile")
(entity:annotation-names (list-ref anno_input1 1))
;; ("path" "path" "path" "path" "path" "path" "path"
;; "path" "path" "path" "path" "path" "path" "path"
;; "path" "path")

; Create a second list of annotations from the first
; swept face. There are many to choose from. This only
; uses the first face in the list.
(define anno2 (entity:annotations
    (list-ref out_sw_faces 0)))
;; anno2
; anno2 => (#[entity 98 1])
; See what the annotation names are for the
; annotation attached to the first swept face.
(annotation:member-name (list-ref anno2 0)
    (list-ref out_sw_faces 0))
;; "lateral_face"
```

```
; Create a second list of input annotations,
; but this one is for the annotations of the first
; face of the swept body.
(define anno_input2
    (annotation:inputs (list-ref anno2 0)))
;; anno_input2
; anno_input2 => (#[entity 10 1] #[entity 95 1])
; See what the names are for the input annotations.
(entity:annotation-names (list-ref anno_input2 0))
;; ("top_edge" "profile" "profile" "profile"
;; "profile" "profile" "profile" "profile" "profile")
(entity:annotation-names (list-ref anno_input2 1))
;; ("path" "path" "path" "path" "path" "path" "path"
;; "path" "path" "path" "path" "path" "path" "path"
;; "path" "path")

; Create a list of annotations just from one of the
; four input profile edges.
(define anno3 (entity:annotations
    (list-ref in_prof_edges 0)))
;; anno3
; anno3 => (#[entity 97 1] #[entity 98 1]
; #[entity 99 1] #[entity 100 1] #[entity 96 1]
; #[entity 101 1] #[entity 102 1] #[entity 103 1]
; #[entity 104 1])
(annotation:member-name (list-ref anno3 0)
    (list-ref in_prof_edges 0))
;; "profile"
(annotation:member-name (list-ref anno3 8)
    (list-ref in_prof_edges 0))
;; "top-edge"

; Create a list of output annotations from the
; list of annotations associated with an edge (the
; first edge) of the input profile.
(define output1 (annotation:outputs
    (list-ref anno3 0)))
;; output1
; output1 => (#[entity 19 1])
; See what the name is of the annotation.
(entity:annotation-names (list-ref output1 0))
; ("lateral_face")
```

Example 8-1. Annotation Used With Sweeping

If you were converting Example 8-1 into C++, you would turn on annotations using api_set_int_option. It would not be required to turn off unhook_annotations.

After setting the options, establish the profile, path, and required sweep_options. The api_sweep_with_options then automatically creates the annotations.

Use api_find_annotations to acquire an entity list of the annotations for further query. Each entity annotation has its own set of methods for query, as well as their own “is_” functions to ascertain the annotation class type.

When finished, use api_clear_annotations to remove the annotations before the next modeling operation is performed.

The following example repeats the sweep modeling operation from Example 8-1. Then it loads the file annotation.scm which is an interactive rubberbander for seeing annotation information.

Scheme Example

```
; -------------------------------------------
; Repeat of code in Example 8-1. (below)
; -------------------------------------------

; Turn on annotation options so that annotations are
; automatically created as part of the modeling operation.
(option:set ”annotation” #t)
;; #f
; Turn off the unhook annotation option so that termination
; of the modeling operation does not clear annotations.
(option:set ”unhook_annotations” #f)
;; #t

; Create a sweep path from a wire body of edges.
(define path1 (wire-body (list
    (edge:linear (position 0 0 0)(position 20 0 0))
    (edge:circular-3pt (position 20 0 0)
        (position 27 0 2.5) (position 30 0 10))
    (edge:linear (position 30 0 10)
        (position 30 0 20)))))
;; path1

; Sweep the edges into a face body.
(define a_body (sweep:law
    (edge:linear ( position 0 3 3)( position 0 3 -3))
    (edge:linear ( position 0 3 -3)
    ( position 0 -3 -3))))
;; a_body
; a_body => #[entity 8 1]
; Create the profile to use in sweeping
(define profile1 (car (entity:faces
    (sheet:2d a_body))))
;; profile1
; profile1 => #[entity 9 1]
; Sweep the profile along the path.
(define sweep1 (sweep:law profile1 path1))

; ------------------------------------------
; Repeat of code in Example 8-1. (above)
; ------------------------------------------

; The scm/examples directory contains a script file that
; can be used to view annotations in an interactive manner.
; Make sure ”annotation.scm” is copied to the directory where
; ”acisinit.scm” is located, or know its path.
(load ”annotation.scm”)

; This loads the annotation rubberbander. When you move the
; mouse over entities of the swept model, the selected entity is
; in one color and its associated input or output is in another.
```

Example 8-2. Annotation Used With Sweeping

In Example 8-2 after the sweep modeling operation is performed, the file annotation.scm is loaded for the interactive rubberbander for seeing annotation information.

Move the mouse over topology to pick and highlight. Information about the selected item is output to the scheme command window. When an item is selected with the left mouse button, more information about that item is output to the command window. The right mouse button toggles between picking vertices and edges. Picking faces is always active. Use the shift key to pick back faces, or specifically, the second face from the front face.

The selected entity is white when using the annotation rubberbander. Output entities are red, input entities are magenta, and all other entities remain green.