Chapter 4.
Classes

Topic: Ignore

The class interface is a set of C++ classes, including their public and protected data and methods (member functions), that an application can use directly to interact with ACIS. Developers may also derive their own classes from these classes to add application-specific functionality and data. Refer to the *3D ACIS Online Help User's Guide* for a description of the fields in the reference template.

ATTRIB_EFINT

Class: Booleans, SAT Save and Restore

Purpose: Defines an attribute to record the intersections of an edge of one body with a face of the other body, during a Boolean operation.

Derivation: ATTRIB_EFINT : ATTRIB_SYS : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: "efint"

Filename: bool/boolean/kernbool/bool1/at_bool1.hxx

Description: Each edge of each body which intersects with any face of the other body (or indeed passes near enough for us to have to perform the detailed calculation) is given an intersection attribute for each such face. This EFINT attribute points to and *owns* a list of edge_face_int records, each of which represents one intersection of the edge with the surface of the face.

When a vertex of the intersection graph is created, the fact is recorded in the appropriate edge_face_int, so that it may be retrieved for other segments of the graph linked to it. If an intersection lies at one end of an edge, at a vertex of the body, that body vertex is also given a single EFINT attribute. This points to a single edge_face_int record, which is any one of the records attached to any of the edges attached to the vertex which represents this intersection.

Limitations: None

References: INTR edge_face_int
KERN ENTITY

Data:

```
public edge_face_int *intersect;
```
List of edge face intersections.

```
public double high_param;
```
High parameter of edge.

```
public double low_param;
```
Low parameter of edge.

```
public tedge_face_header *list;
```
List of tolerant edge intersections.

Constructor:

```
public: ATTRIB_EFINT::ATTRIB_EFINT (
    ENTITY*,                  // entity name
    EDGE*,                    // edge
    edge_face_int*            // edge face name
        = NULL,
    double                    // first parameter
        = 0,
    double                    // second parameter
        = 0,
    FACE* owner_face          // face of tolerant edge
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_EFINT(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: ATTRIB_EFINT::ATTRIB_EFINT (
    ENTITY*                  // entity (edge)
        = NULL,
    FACE*                    // face in the other body
        = NULL,
    edge_face_int*           // list of edge face
        = NULL,              // intersections
    double                   // edge start parameter
        = 0,
    double                   // sedge end parameter
        = 0,
    FACE*                    // face of edge
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_EFINT(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_EFINT::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_EFINT::~ATTRIB_EFINT ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_EFINT(...) then later x–>lose.)

Methods:

```
public: virtual void ATTRIB_EFINT::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: void ATTRIB_EFINT::disable ();
```

Set the entity pointer to NULL, to prevent the reuse of this attribute in the current Boolean.

```
public: EDGE* ATTRIB_EFINT::edge () const;
```

Pick out related edge.

```
public: ENTITY* ATTRIB_EFINT::entity () const;
```

Pick out related entity.

```
public: FACE* ATTRIB_EFINT::face () const;
```

Pick out related face.

```
public: edge_face_int* ATTRIB_EFINT::get_intersect (
    FACE*                   // input face
    ) const;
```

Finds the intersection list for a given face.

```
public: virtual int ATTRIB_EFINT::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_EFINT_TYPE. If level is specified, returns ATTRIB_EFINT_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_EFINT_LEVEL.

```
public: virtual logical
    ATTRIB_EFINT::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: void ATTRIB_EFINT::merge_tolerant_ef_ints (
    edge_face_int*          // intersections
    );
```

Merge graph edges and graph vertices.

```
public: void ATTRIB_EFINT::null_graph_vertex (
    VERTEX*                 // vertex
    );
```

Nulls the specified graph vertex.

```
public:virtual logical
    ATTRIB_EFINT::pattern_compatible () const ;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_EFINT::prepend (
    FACE*,                  // input face
    edge_face_int*          // edge face intersection
    ) ;
```

Appends an edge face intersection record to an attribute.

```
public: void ATTRIB_EFINT::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: void ATTRIB_EFINT::set_intersect (
    FACE*,                  // input face
    edge_face_int*          // intersection list
    ) ;
```

Sets the intersection list for a given face.

```
public: virtual const char*
    ATTRIB_EFINT::type_name () const;
```

Returns the string “efint”.

Related Fncs:

delete_efint_list, get_attrib_efint_list, init_attrib_efint_list, is_ATTRIB_EFINT

ATTRIB_FACEINT

Class: Booleans, SAT Save and Restore

Purpose: Defines an attribute to record the intersection of a face of one body with a face of the other body during a Boolean operation.

Derivation: ATTRIB_FACEINT : ATTRIB_SYS : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “faceint”

Filename: bool/boolean/kernbool/bool1/at_bool1.hxx

Description: This attribute is attached to a face of the tool body, and refers to a face of the blank body. This attribute may be attached by code that precedes a Boolean operation, such as blending.

Most Boolean operations do not bother to construct these attributes, as each face-to-face combination is looked at only once, but they may be attached before entering the Boolean code, if the intersections have already been determined by preliminary code (for example in blending).

If one of these attributes is stored, it is attached to the tool body face, and refers to the blank face.

In the special case of intersecting a body with itself (with no transformation), as may occur in testing for self-intersection, each face pair will in the normal course of events be inspected twice, and so it may be worth saving the results of the first operation. When it is seen a second time, the original tool face will be the blank, and vice versa, so a search on the blank face is needed also, and if found, the intersections must be flipped to reflect the inverted face order.

Limitations: None

References: KERN FACE, surf_surf_int

Data:

```
public FACE *face;
```

The other face.

```
public surf_surf_int *intersect;
```
List of surface to surface intersections.

Constructor:

```
public: ATTRIB_FACEINT::ATTRIB_FACEINT (
    FACE*                   // tool face
        = NULL,
    FACE*                   // blank face
        = NULL,
    surf_surf_int*          // surface–surface int
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_FACEINT(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_FACEINT::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual
    ATTRIB_FACEINT::~ATTRIB_FACEINT ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_FACEINT(...) then later x–>lose.)

Methods:

```
public: virtual void ATTRIB_FACEINT::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int ATTRIB_FACEINT::identity (
    int                       // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_FACEINT_TYPE. If level is specified, returns ATTRIB_FACEINT_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_FACEINT_LEVEL.

```
public: virtual logical
    ATTRIB_FACEINT::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public:virtual logical
    ATTRIB_FACEINT::pattern_compatible () const ;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_FACEINT::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: virtual const char*
    ATTRIB_FACEINT::type_name () const;
```

Returns the string “faceint”.

Related Fncs:

delete_efint_list, get_attrib_efint_list, init_attrib_efint_list, is_ATTRIB_FACEINT

ATTRIB_INTCOED

Class: Booleans, SAT Save and Restore

Purpose: Defines an attribute for linking intersection graph entities with the relevant body entities.

Derivation: ATTRIB_INTCOED : ATTRIB_SYS : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “intcoed”

Filename: bool/boolean/kernbool/boolean/at_bool.hxx

Description: Defines an attribute for linking intersection coedges with body faces to which they will attach. It is private to the Boolean operator code, but is required by more than one phase.

The wires of the graph consist of coedges, edges and vertices, together with their geometries. Each of these entities carries exactly one attribute, recording information relevant to its role in the later stages of Booleans. All these attributes are cleaned out during the latter stages of Booleans, as they cease to be useful.

ATTRIB_INTCOED is attached to each intersection coedge (in both blank and tool wires), and contains:

– A pointer to the face on the body that the coedge corresponds.
– The relationship between the portion of that face to the left of the coedge and in its neighborhood, and the surface of the other body. In this context, left refers to a coordinate system that the coedge direction is forward and the normal to the face is upward. (Throughout ACIS the face that a coedge is attached is on its left.) This relationship is of an enumerated type, and takes the values:

face_body_inside	Specifies the face/body is inside.
face_body_outside	Specifies the face/body is outside.
face_body_retain	Used for non-Boolean purposes to retain the operation.
face_body_discard	Used for non-Boolean purposes to discard the operation.
face_body_symmetric	If face lies in the surface of the other body, specifies that the normals are in opposite directions.
face_body_antisymmetric	If face lies in the surface of the other body, specifies that the normals are in the same direction.

– A classification of the coedge with respect to the face that it lies; this is another enumerated type, and takes the values:

edge_class	The coedge lies wholly on boundary of face.
boundary_class	The coedge lies within the face, but its start vertex lies on the boundary.
face_class	The coedge lies within the face, and its start vertex is properly within the face.

– A pointer to a coedge on the body face, the exact meaning depending upon the coedge classification:

edge_class	Corresponding coedge of the edge that this coedge lies.
boundary_class	The coedge lies on body face passing through the start vertex of intersection coedge. If this start vertex lies on a vertex of the face, the coedge of the face that starts at this vertex is chosen.
face_class	Pointer to an attribute on a coedge on this same geometric edge that has a body face coincident with this coedge's. A flag is set TRUE when the body face of this coedge is processed.

Limitations: None

References: KERN COEDGE, ENTITY

Data:

None

Constructor:

```
public: ATTRIB_INTCOED::ATTRIB_INTCOED (
    COEDGE*,                // coedge
    ATTRIB_INTCOED*         // ATTRIB_INTCOED
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_INTCOED(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: ATTRIB_INTCOED::ATTRIB_INTCOED (
    COEDGE*                 // coedge owner
        = NULL,
    ENTITY*                 // adjacent entity
        = NULL,
    face_body_rel           // adjacent relationship
        = face_body_unknown,
    COEDGE*                 // adjacent coedge
        = NULL,
    coedge_type             // adjacent type
        = unknown_class
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_INTCOED(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_INTCOED::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual
    ATTRIB_INTCOED::~ATTRIB_INTCOED ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_INTCOED(...) then later x–>lose.)

Methods:

```
public: COEDGE* ATTRIB_INTCOED::body_coedge () const;
```

Points to the coedge linking the edge to the face if the start vertex of this coedge lies on an edge of the face. If the start vertex is at a vertex of the face, this coedge is the one that starts at the vertex.

```
public: ENTITY* ATTRIB_INTCOED::body_entity () const;
```

Pick out a body entity.

```
public: ATTRIB_INTCOED*
    ATTRIB_INTCOED::coin_attrib () const;
```

Points to the attribute attached to the coedge on the same geometric edge associated with the corresponding race on the other body if the face relationship indicates a coincidence (face_body_symmetric or face_body_antisymmetric); otherwise, the pointer is meaningless and NULL returns.

```
public: face_body_conf ATTRIB_INTCOED::conf () const;
```

Simple read function to return the data associated with the confidence to be placed in the value of face_rel_data during the construction of the intersection graph.

```
public: virtual void ATTRIB_INTCOED::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: EDGE* ATTRIB_INTCOED::edge () const;
```

Pick out an edge entity.

```
public: FACE* ATTRIB_INTCOED::face () const;
```

Returns the face of the body on which this coedge lies.

```
public: face_body_rel ATTRIB_INTCOED::face_rel ()
    const;
```

Returns the containment of this face with respect to the other body.

```
public: logical ATTRIB_INTCOED::face_seen () const;
```

Used during Boolean Stage 1, returns FALSE, but it is TRUE when the body face to which this attribute refers is processed. At the end of Stage 1, if there are attributes remaining that have not been processed,the containments of the other body faces are unreliable, and so must be evaluated explicitly.

```
public: virtual int ATTRIB_INTCOED::identity (
    int                       // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_INTCOED_TYPE. If level is specified, returns ATTRIB_INTCOED_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_INTCOED_LEVEL.

```
public: virtual logical
    ATTRIB_INTCOED::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public:virtual logical
    ATTRIB_INTCOED::pattern_compatible () const ;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_INTCOED::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if (restore_version_number < CONSISTENT_VERSION)
    read_int                    face body relationship data
else
    read_enum                   Read the face_body_rel_map
                                variable of the face_body_rel
                                enumeration.
```

```
public: void ATTRIB_INTCOED::set_body_coedge (
    COEDGE*                 // new coedge
    );
```

Sets a new coedge pointer after the edge split occurs in bool2.

```
public: void ATTRIB_INTCOED::set_body_entity (
    ENTITY*                 // entity name
    );
```

Sets a new body entity.

```
public: void ATTRIB_INTCOED::set_coin_attrib (
    ATTRIB_INTCOED*         // ATTRIB_INTCOED
    );
```

Sets a new coincident face attribute pointer.

```
public: void ATTRIB_INTCOED::set_edge (
    EDGE*                   // edge name
    );
```

Sets a new edge.

```
public: void ATTRIB_INTCOED::set_face (
    FACE*                   // new face
    );
```

Sets the face adjacent to the graph coedge.

```
public: void ATTRIB_INTCOED::set_face_rel (
    face_body_rel,              // new relationship
    face_body_conf              // confirmation?
        = face_body_unconfirmed
    );
```

Sets the relationship of the coedge attribute.

```
public: void ATTRIB_INTCOED::set_face_seen ();
```

Marks this coedge's body face as having been processed.

```
public: void ATTRIB_INTCOED::set_type (
    coedge_type             // coedge type
    );
```

Sets the graph coedge type, which is one of the following: edge_class if it lies on the edge of the face, boundary_class if it starts on the boundary, or face_class if it starts in the interior.

```
public: void ATTRIB_INTCOED::transfer (
    const ENTITY_LIST& old_ents,// old entity list
    const ENTITY_LIST& new_ents // new entity list
    );
```

Resets pointers to entities in the first list to the corresponding entities in the second list.

```
public: coedge_type ATTRIB_INTCOED::type () const;
```

Returns the classification of the intersection edge with respect to the current body.

```
public: virtual const char*
    ATTRIB_INTCOED::type_name () const;
```

Returns the string "intcoed".

Related Fncs:

is_ATTRIB_INTCOED

ATTRIB_INTEDGE

Class: Booleans, SAT Save and Restore

Purpose: Defines an attribute for linking intersection edges with the intersecting entities.

Derivation: ATTRIB_INTEDGE : ATTRIB_SYS : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “intedge”

Filename: bool/boolean/kernbool/boolean/at_bool.hxx

Description: This class defines an attribute for linking intersection edges with the intersecting entities. This is a private class to the Boolean operator code, but is required by more than one phase.

ATTRIB_INTEDGE is attached to each intersection edge, and contains the following information for each body (this_body and other_body, meaning at this stage blank body and tool body respectively):

- A pointer to the entity on the body that this edge corresponds; an EDGE if it lies coincident with some part of that edge, otherwise the face in which it lies.
- The sense relating the intersection edge and the body edge if the body entity is an EDGE. This sense is FORWARD if the two edges are in the same direction, REVERSED if they are in opposite directions.
- A pointer to one of the coedges belonging to this edge in the wire corresponding to the tool body.
- A pointer to a *partner* ATTRIB_INTEDGE attribute, which at this stage must be NULL.

The following functions are defined for ATTRIB_INTEDGE.

Limitations: None

References: KERN COEDGE

Data:

None

Constructor:

```
public: ATTRIB_INTEDGE::ATTRIB_INTEDGE (
    EDGE*                   // edge owner
        = NULL,
    ATTRIB_INTEDGE*         // copy
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_INTEDGE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_INTEDGE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual
    ATTRIB_INTEDGE::~ATTRIB_INTEDGE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_INTEDGE(...) then later x–>lose.)

Methods:

```
public: virtual void ATTRIB_INTEDGE::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: logical ATTRIB_INTEDGE::fuzzy_int () const;
```

Sets whether this graph edge derives from a fuzzy region on an edge of one or both bodies. If so, containments are unreliable and should be derived from coherence with adjacent edges.

```
public: virtual int ATTRIB_INTEDGE::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_INTEDGE_TYPE. If level is specified, returns ATTRIB_INTEDGE_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_INTEDGE_LEVEL.

```
public: virtual logical
    ATTRIB_INTEDGE::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: ENTITY*
    ATTRIB_INTEDGE::other_entity () const;
```

Returns the edge for the other body.

```
public: REVBIT ATTRIB_INTEDGE::other_sense () const;
```

Returns the sense data for the other body.

```
public: ATTRIB_INTEDGE*
    ATTRIB_INTEDGE::partner () const;
```

Points to the corresponding attribute on the graph for the other body. This value is NULL when there is only one.

```
public:virtual logical
    ATTRIB_INTEDGE::pattern_compatible () const ;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_INTEDGE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: void ATTRIB_INTEDGE::set_fuzzy_int ();
```

Sets whether this graph edge derives from a fuzzy region on an edge of one or both bodies. Once set, this flag does not need to be reset.

```
public: void ATTRIB_INTEDGE::set_other_body (
    ENTITY*,                // new entity
    REVBIT                  // new sense data
    );
```

Sets the data associated with the other body.

```
public: void ATTRIB_INTEDGE::set_this_body (
    ENTITY*,                // new entity
    REVBIT                  // new sense data
    );
```

Sets the data associated with this body.

```
public: void ATTRIB_INTEDGE::set_tool_coedge (
    COEDGE*                 // new coedge
    );
```

Sets the tool coedge pointer.

```
public: ENTITY* ATTRIB_INTEDGE::this_entity () const;
```

Returns the edge for this body.

```
public: REVBIT ATTRIB_INTEDGE::this_sense () const;
```

Returns the sense data for this body.

```
public: COEDGE* ATTRIB_INTEDGE::tool_coedge () const;
```

Points to the coedges relating to the tool body, transferred to the partner edge when this is constructed.

```
public: void ATTRIB_INTEDGE::transfer (
    const ENTITY_LIST& old_ents,// old entity list
    const ENTITY_LIST& new_ents // new entity list
    );
```

Resets pointers to entities in the first list to the corresponding entities in the second list.

```
public: virtual const char*
    ATTRIB_INTEDGE::type_name () const;
```

Returns the string “intedge”.

Related Fncs:

is_ATTRIB_INTEDGE

ATTRIB_INTGRAPH

Class: Booleans, SAT Save and Restore

Purpose: Defines an attribute for classifying shells and lumps of two bodies participating in a Boolean operation.

Derivation: ATTRIB_INTGRAPH : ATTRIB_SYS : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “intgraph”

Filename: bool/boolean/kernbool/boolean/at_bool.hxx

Description: This class maintains a linked list of shell-lump objects. It is private to the Boolean operator code, but is required by more than one phase.

Limitations: None

References:

BOOL	shell_lump
by BOOL	shell_lump

Data:

None

Constructor:

```
public: ATTRIB_INTGRAPH::ATTRIB_INTGRAPH (
    BODY*                   // body name
        = NULL,
    shell_lump*             // shell lump list
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_INTGRAPH(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_INTGRAPH::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual
    ATTRIB_INTGRAPH::~ATTRIB_INTGRAPH ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_INTGRAPH(...) then later x–>lose.)

Methods:

```
public: virtual void ATTRIB_INTGRAPH::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int ATTRIB_INTGRAPH::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier, ATTRIB_INTGRAPH_TYPE. If level is specified, returns ATTRIB_INTGRAPH_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_INTGRAPH_LEVEL.

```
public: virtual logical
    ATTRIB_INTGRAPH::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: void ATTRIB_INTGRAPH::negate_tool_cont ();
```

Change the containments in the shell-lump list to reflect a negated tool body.

```
public:virtual logical
    ATTRIB_INTGRAPH::pattern_compatible () const ;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_INTGRAPH::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: shell_lump* ATTRIB_INTGRAPH::sl_list ();
```

Returns a shell lump list.

```
public: virtual const char*
    ATTRIB_INTGRAPH::type_name () const;
```

Returns the string “intgraph”.

Related Fncs:

is_ATTRIB_INTGRAPH

ATTRIB_INTVERT

Class: Booleans, SAT Save and Restore

Purpose: Defines an attribute for linking graph vertices with the intersection record(s) giving rise to them.

Derivation: ATTRIB_INTVERT : ATTRIB_SYS : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “intvert”

Filename: bool/boolean/kernbool/boolean/at_bool.hxx

Description: This class defines an attribute for linking graph vertices with the intersection record(s) giving rise to them. Where there are several, one from each body is chosen, as the necessary information is recorded in all.

ATTRIB_INTVERT is attached to each intersection vertex, and contains the following information for each body (as for ATTRIB_INTEDGE attributes):

- A pointer to the entity on the body that the vertex corresponds - a VERTEX if it lies coincident with that body vertex, an EDGE if it lies on that edge and not at either end, or NULL if it lies properly within a face.
- The parameter value along the edge, if the entity pointed to is an EDGE, undefined otherwise.
- A pointer to a partner VERTEX (not to an ATTRIB_INTVERT attribute), which at this stage must be NULL.

Limitations: None

References: KERN VERTEX

Data:

None

Constructor:

```
public: ATTRIB_INTVERT::ATTRIB_INTVERT (
    ATTRIB_INTVERT*          // existing attribute
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_INTVERT(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: ATTRIB_INTVERT::ATTRIB_INTVERT (
    VERTEX*                 // vertex owner
        = NULL,
    ENTITY*                 // this entity
        = NULL,
    double                  // this edge parameter
        = 0,
    edge_face_int*          // this edge-face int
        = NULL,
    ENTITY*                 // other entity
        = NULL,
    double                  // other edge parameter
        = 0,
    edge_face_int*          // other edge-face int
        = NULL,
    COEDGE*                 // this coedge
        = NULL,
    COEDGE*                 // other coedge
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_INTVERT(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_INTVERT::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual
    ATTRIB_INTVERT::~ATTRIB_INTVERT ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_INTVERT(...) then later x–>lose.)

Methods:

```
public: virtual void ATTRIB_INTVERT::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int ATTRIB_INTVERT::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_INTVERT_TYPE. If level is specified, returns ATTRIB_INTVERT_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_INTVERT_LEVEL.

```
public: virtual logical
    ATTRIB_INTVERT::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: void ATTRIB_INTVERT::kill_other_ef_int ();
```

Kills the other edge-face intersection. The edge-face intersection is the primary edge-face intersection record for this vertex in this body. This is only meaningful during the construction of the intersection of the graph (bool1).

```
public: void ATTRIB_INTVERT::kill_this_ef_int ();
```

Kills this edge-face intersection. The edge-face intersection is the primary edge-face intersection record for this vertex in this body. This is only meaningful during the construction of the intersection of the graph (bool1).

```
public: COEDGE* ATTRIB_INTVERT::other_coedge ();
```

Returns a pointer to the other coedge.

```
public: double ATTRIB_INTVERT::other_edge_param ();
```

Returns the other edge parameter. This is used only if entity is an edge because it gives the parameter value of the point along the edge.

```
public: edge_face_int*
    ATTRIB_INTVERT::other_ef_int ();
```

Returns the other edge-face intersection. The edge-face intersection is the primary edge-face intersection record for this vertex in this body. This is only meaningful during the construction of the intersection of the graph (bool1).

```
public: ENTITY* ATTRIB_INTVERT::other_entity ();
```

Returns the other entity, which is an edge or a vertex.

```
public: VERTEX* ATTRIB_INTVERT::partner ();
```

Returns the vertex created on the duplicate graph.

```
public:virtual logical
    ATTRIB_INTVERT::pattern_compatible () const ;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_INTVERT::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: void ATTRIB_INTVERT::set_other_body (
    ENTITY*,                // other entity
    double,                 // other edge parameter
    edge_face_int*          // other edge-face int
    );
```

Sets the properties of the other body involved in the intersection.

```
public: void ATTRIB_INTVERT::set_other_coedge (
    COEDGE*                 // new coedge
    );;
```

Sets the pointer to the other coedge.

```
public: void ATTRIB_INTVERT::set_partner (
    VERTEX*                 // vertex
    );
```

Sets the vertex created on a duplicate graph, which is used only during graph duplication.

```
public: void ATTRIB_INTVERT::set_this_body (
    ENTITY*,                // this entity
    double,                 // this edge parameter
    edge_face_int*          // this edge-face int
    );
```

Sets the properties of the body owning this graph; in the early stages, this is the blank body.

```
public: void ATTRIB_INTVERT::set_this_coedge (
    COEDGE*                 // new coedge
    );;
```

Sets the pointer to this coedge.

```
public: COEDGE* ATTRIB_INTVERT::this_coedge ();
```

Returns a pointer to this coedge.

```
public: double ATTRIB_INTVERT::this_edge_param ();
```

Returns this edge parameter. This is used only if entity is an edge because it gives the parameter value of the point along the edge.

```
public: edge_face_int*
    ATTRIB_INTVERT::this_ef_int ();
```

Returns this edge-face intersection. The edge-face intersection is the primary edge-face intersection record for this vertex in this body. This is only meaningful during the construction of the intersection of the graph (bool1).

```
public: ENTITY* ATTRIB_INTVERT::this_entity ();
```

Returns this entity, which is an edge or a vertex.

```
public: void ATTRIB_INTVERT::transfer (
    const ENTITY_LIST& old_ents,// old entity list
    const ENTITY_LIST& new_ents // new entity list
    );
```

Resets pointers to entities in the first list to the corresponding entities in the second list.

```
public: virtual const char*
    ATTRIB_INTVERT::type_name () const;
```

Returns the string “intvert”.

Related Fncs:

is_ATTRIB_INTVERT

glue_options

Class: Booleans

Purpose: Class to hold information and options for a glue operation.

Derivation: glue_options : AcisOptions : ACIS_OBJECT : –

SAT Identifier: None

Filename: bool/boolean/kernbool/boolean/glue_opts.hxx

Description: A glue_options object is to be used in conjunction with two bodies (blank and tool) whose intersection is known to lie along a set of coincident faces. See documentation for api_boolean_glue for the definition of coincident faces.

tfaces and bfaces are arrays of pointers to pairwise coincident faces of the tool and blank respectively. Therefore, the length of these arrays must be the same and must be equal to num_coi_faces.

Note This information is not included in the scheme glue options class.

There are various flags which are all unset (–1) by default. Certain combinations can be set to improve performance. It is important that the information provided is accurate, as the glue operation will rely heavily on this information.

Given bodies b1 and b2, a coincident patch P1 in b1 is a maximal set of connected faces of b1 such that there exists a corresponding maximal set P2 of connected faces in b2 and a (well–defined onto) coincidence mapping from P1 to P2.

Face f1 covers face f2 if the point set of f2 is a subset of the point set of f1. Patch P1 covers patch P2 if the point set of P2 is a subset of the point set of P1. Given a pair of coincident patches P1 and P2, P1 is a strict cover of P2 if the point set of P2 is a subset of the interior point set of P2.

Setting patch_and_face_cover to TRUE will induce a performance enhancement. patch_and_face_cover may be set to TRUE if the following conditions are met:

for every pair of coincident faces (to be specified in the glue operation), one face covers the other face;
for every pair of coincident patches, one patch covers the other patch.

In addition to setting patch_and_face_cover to TRUE, setting blank_patches_strict_cover to TRUE will induce another performance enhancement. blank_patches_strict_cover may be set to TRUE if the following conditions are met:

patch_and_face_cover is set to TRUE;
every patch in the blank (first) body is a strict cover of its corresponding patch in the tool (second) body.

non_trivial may be set to TRUE if it is guaranteed that the boolean operation will be non–trivial. In the case of glue–unite, this is when the tool body lies outside the blank body. In the case of glue–subtract, this is when the tool body is completely contained in the blank body. This will induce another performance enhancement. It is not dependent on the previous flags.

Limitations: None

References: KERN ENTITY

Data:

None

Constructor:

```
public: glue_options::glue_options (
    int                         // number of coincident
        = 0,                    // faces
    ENTITY**                    // tool faces
        = NULL,
    ENTITY**                    // blank faces
        = NULL
    );
```

C++ constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

```
public: virtual glue_options::~glue_options ();
```

Destructor.

Methods:

```
public: logical glue_options::check () const;
```

Performs sanity check on this object. Returns FALSE if face_pair_cover flag is set to UNSET or FALSE and blank_patches_strict_cover flag is et to TRUE. Returns TRUE in all other cases.

```
public: glue_options* glue_options::copy () const;
```

Returns a copy of this object.

```
public: void glue_options::copy_flags (
    const glue_options*     // input object
    );
```

Copies the flag values from the input object to this object.

```
public: ENTITY** glue_options::get_bfaces () const;
```

Returns the blank faces.

```
public: int
glue_options::get_blank_patches_strict_cover ()
const;
```

Returns the current value of blank_patches_strict_cover flag.

```
public: int glue_options::get_non_trivial () const;
```

Returns the current value of non_trivial flag.

```
public: int glue_options::get_num_coi_faces () const;
```

Returns the number of coincident faces.

```
public: int glue_options::get_patch_and_face_cover ()
const;
```

Returns the current value of patch_and_face_cover flag.

```
public: ENTITY** glue_options::get_tfaces () const;
```

Returns the tool faces.

```
public: logical glue_options::operator!= (
    glue_options const& in_glue_opt // object to
                                    // compare
    ) const;
```

Returns TRUE if the input object is not same as this object or FALSE otherwise.

```
public: logical glue_options::operator== (
    glue_options const& in_glue_opt // object to
                                    // compare
    ) const;
```

Returns TRUE if the input object is same as this object or FALSE otherwise.

```
public: void
glue_options::set_blank_patches_strict_cover (
    int                      // value to set
    );
```

Copies input value to blank_patches_strict_cover flag.

```
public: void glue_options::set_data (
    int,                    // number of coincident
                            // faces
    ENTITY**,               // tool faces
    ENTITY**                // blank faces
    );
```

Copies the input values to this object.

```
public: void glue_options::set_flags (
    int,                    // face pair cover
    int,                    // blank patches strict
                            // cover
    int                     // non trivial
    );
```

Copies the flag values to this object.

```
public: void glue_options::set_non_trivial (
    int                     // value to set
    );
```

Copies the input value to non_trivial flag.

```
public: void glue_options::set_patch_and_face_cover (
    int                     // value to set
    );
```

Copies the input value to patch_and_face_cover flag.

Related Fncs:

None

NO_MERGE_ATTRIB

Class: Booleans, SAT Save and Restore

Purpose: Specifies a user-defined attribute that signals that the edge is not to be merged out of the body.

Derivation: NO_MERGE_ATTRIB : ATTRIB_ST : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “no_merge_attribute”

Filename: bool/boolean/sg_husk/merge/mer_attr.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

```
public: NO_MERGE_ATTRIB::NO_MERGE_ATTRIB (
    ENTITY*                  // owning entity
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new NO_MERGE_ATTRIB(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void NO_MERGE_ATTRIB::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual
    NO_MERGE_ATTRIB::~NO_MERGE_ATTRIB ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new NO_MERGE_ATTRIB(...) then later x–>lose.)

Methods:

```
public: virtual void NO_MERGE_ATTRIB::debug_ent (
    FILE*                    // output file
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int NO_MERGE_ATTRIB::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier NO_MERGE_ATTRIB_TYPE. If level is specified, returns NO_MERGE_ATTRIB_TYPE for that level of derivation from ENTITY. The level of this class is defined as NO_MERGE_ATTRIB_LEVEL.

```
public: virtual logical
    NO_MERGE_ATTRIB::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public:virtual logical
    NO_MERGE_ATTRIB::pattern_compatible () const ;
```

Returns TRUE if this is pattern compatible.

```
public: void NO_MERGE_ATTRIB::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: virtual const char*
    NO_MERGE_ATTRIB::type_name () const;
```

Returns the string “no_merge_attribute”.

Internal Use: merge_owner, save, save_common, split_owner

Related Fncs: is_NO_MERGE_ATTRIB

shell_lump

Class: Booleans

Purpose: Records the classification of shells or wires that do not contribute to any intersection.

Derivation: shell_lump : ACIS_OBJECT : –

SAT Identifier: None

Filename: bool/boolean/kernbool/boolean/at_bool.hxx

Description: This class records the definition for the shell-in-lump list. For each shell in each constituent body, there is one of these records (except that for shells entirely outside the other body there need not be a corresponding record). The face-body relationship specifies the containment relationship between the shell and the other body; it is “unknown” if the shell intersects with a shell of the other body. For shells wholly coincident with shells on the other body and consisting of faces on a single surface, there is no intersection wire, and so the relationship is recorded here as “symmetric” or “antisymmetric” as appropriate. Other coincidences come through as intersections, and so are handled through the intersection wire.

Where a shell is inside a lump, or coincident with one of its shells, a pointer to that lump (there can be only one) is recorded here. For an intersecting shell, or one outside the other body (if recorded in this list) the lump pointer is NULL.

This class is private to the boolean operator code, but is required by more than one phase.

Limitations: None

References:

by BOOL KERN	ATTRIB_INTGRAPH
	ENTITY

Data:

None

Constructor:

```
public: shell_lump::shell_lump (
    shell_lump*,              // shell lump list
    ENTITY*,                  // shell or wire being
                              // described
    logical,                  // TRUE if blank-body
                              // shell,
                              // FALSE if tool-body
                              // shell
    ENTITY*                   // other entity
        = NULL,
    face_body_rel             // shell-lump
        = face_body_unknown   // relationship
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: shell_lump::shell_lump (
    shell_lump*,              // shell-lump list
    SHELL*,                   // shell being described
    logical,                  // TRUE if blank-body
                              // shell,
                              // FALSE if tool-body
                              // shell
    LUMP*                     // other entity
        = NULL,
    face_body_rel             // shell-lump
        = face_body_unknown   // relationship
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

None

Methods:

```
public: logical shell_lump::blank_entity () const;
```

Returns TRUE if the shell or wire belongs to the blank body, FALSE if it belongs to the tool body.

```
public: logical shell_lump::blank_shell () const;
```

Returns TRUE if the shell belongs to the blank body. It returns FALSE if the shell belongs to the tool body.

```
public: ENTITY* shell_lump::entity () const;
```

Return the shell or wire being described.

```
public: LUMP* shell_lump::lump () const;
```

Returns a pointer to the lump of the other body. Same as other_lump.

```
public: shell_lump* shell_lump::next () const;
```

Returns a pointer to the next shell_lump in the list of shell_lumps.

```
public: ENTITY* shell_lump::other_entity () const;
```

Return the other shell or wire.

```
public: LUMP* shell_lump::other_lump () const;
```

Return the other entity being described if it is a LUMP; otherwise, return NULL.

```
public: SHELL* shell_lump::other_shell () const;
```

Returns the shell of the other body that is coincident with this shell or one shell of the lump of the other body inside which the shell lies. This method returns NULL if the shell is outside everything, or there is one or more intersection wire detailing its interaction.

```
public: WIRE* shell_lump::other_wire () const;
```

Return the other entity being described if it is a LUMP; otherwise, return NULL.

```
public: face_body_rel shell_lump::rel () const;
```

Returns the relationship of the shell to the lumps of the other body.

```
public: void shell_lump::reset_lump (
    LUMP*                   // lump to be updated
    );
```

Normally the information in a shell_lump is not changed once it has been initialized, but chop sometimes splits lumps, thus requiring the information to be updated.

```
public: SHELL* shell_lump::shell () const;
```

Return the entity being described if it is a shell. If the entity is a wire, return its shell; otherwise, return NULL.

```
public: void shell_lump::transfer (
    const ENTITY_LIST& old_ents,// old entity list
    const ENTITY_LIST& new_ents // new entity list
);
```

Resets pointers to entities in the first list to the corresponding entities in the second list.

```
public: WIRE* shell_lump::wire () const;
```

Return the entity being described if it is a LUMP; otherwise, return NULL.

Related Fncs:

None