Chapter 32.
Classes Fa thru Kz

Topic: Ignore

FACE

Class: Model Topology, SAT Save and Restore

Purpose: Represents a bounded portion of a SURFACE.

Derivation: FACE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “face”

Filename: kern/kernel/kerndata/top/face.hxx

Description: A face is a bounded portion of a single geometric surface, the two-dimensional analog of a body. The boundary is represented by one or more loops or edges. Each face is simply connected, implying that one can traverse from any point on the interior of the face to any other point on the interior of the face without crossing the boundary of the face. In general, it is not meaningful to distinguish exterior and interior loops of edges, though for certain surface types this may be possible and some algorithms may do so.

Face loops need not necessarily be closed, and if not, either open end may be finite or infinite. If either end is infinite, then the face is infinite; if either end is finite, then the face is “incompletely-bounded”, or just “incomplete.” Although such faces can be represented in ACIS, most algorithms cannot handle such faces.

Users may also find the topological traversal functions located in kernel/kerndata/top/query.hxx useful for generating lists of faces on other topological entities or lists of edges and vertices on faces. The get_face_box function may be useful to retrieve or recalculate a face’s bounding box. The reset_boxes function may be useful to reset the bounding box of a face and its parents. Other functions of note include:

point_in_face	determines the containment of a point versus a face.
raytest_face	determines the intersection of a ray with a face.
sg_get_face_normal	Calculates a normal at a point on a face.
find_cls_ptto_face	finds the closest point to a specified point on a face.

Limitations: None

References:

KERN	LOOP, SHELL, SUBSHELL, SURFACE
by KERN	LOOP, SHELL, SUBSHELL, pattern_holder

Data:

None

Constructor:

```
public: FACE::FACE ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: FACE::FACE (
    FACE*,                  // old FACE
    LOOP*,                  // first loop
    logical                 // update FACE list
        = TRUE
    );
```

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, because this reserves the memory on the heap, a requirement to support roll back and history management.

Creates a FACE, using the given LOOP list, but taking geometry, senses, and shell and subshell owners from the old FACE. Optionally updates the SHELL or SUBSHELL FACE list to contain the new FACE (by default).

```
public: FACE::FACE (
    LOOP*,                  // first LOOP
    FACE*,                  // next FACE
    SURFACE*,               // SURFACE
    REVBIT                  // sense
    );
```

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, because this reserves the memory on the heap, a requirement to support roll back and history management.

Creates a FACE. initializing the record and interfacing with the bulletin board. The arguments initialize the first LOOP on the face, the next FACE on the BODY, the underlying SURFACE geometry, and the sense of the FACE relative to the surface (FORWARD or REVERSED) respectively. It increments the SURFACE use count to reflect this new use. It also sets the backpointers (to the FACE) in the LOOPs that must be correctly chained together before this constructor is called. The calling routine must set shell_ptr or subshell_ptr to refer to the owning SHELL or SUBSHELL, and if desired, bound_ptr, using set_shell or set_subshell and set_bound.

Destructor:

```
public: virtual void FACE::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 FACE::~FACE ();
```

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 FACE(...) then later x–>lose.)

Methods:

```
public: SPAbox* FACE::bound () const;
```

Returns a pointer to a geometric bounding region (a box), within which the entire FACE lies. The pointer is NULL if no such bound was calculated since the FACE was last changed.

```
protected: virtual logical
    FACE::bulletin_no_change_vf (
    ENTITY const* other,        // other entity
    logical identical_comparator// comparator
    ) const;
```

Virtual function for comparing subclass data – called by bulletin_no_change. For the identical_comparator argument to be TRUE requires an exact match when comparing doubles and returns the result of memcmp as a default (for non–overridden subclasses). FALSE indicates tolerant compares and returns FALSE as a default.

```
public: CONTBIT FACE::cont () const;
```

Returns the containment of the face (BOTH_OUTSIDE or BOTH_INSIDE). This value is meaningless if the face is single-sided.

```
public: logical FACE::copy_pattern_down (
    ENTITY* target              // target
    ) const;
```

Copies the pattern through all children of the target entity.

```
public: virtual void FACE::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: SURFACE* FACE::geometry () const;
```

Returns a pointer to the underlying SURFACE defining the FACE.

```
public: void FACE::get_all_patterns (
    VOID_LIST& list             // pattern list
    );
```

Returns all patterns.

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

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

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

Returns TRUE if this can be deep copied.

```
public: LOOP* FACE::loop () const;
```

Returns a pointer to the first LOOP of COEDGEs bounding the FACE.

```
public: FACE* FACE::next (
    PAT_NEXT_TYPE next_type // face type
        = PAT_CAN_CREATE    // for patterns
    ) const;
```

Returns the next face in a complete enumeration of all the faces in the shell.

The next_type argument controls how the next method treats patterns, and can take any one of three values:

PAT_CAN_CREATE: if the next face is to be generated from a pattern, create it if it doesn't yet exist and return its pointer.
PAT_NO_CREATE: if the next face is to be generated from a pattern, but hasn't yet been created, bypass it and return the pointer of the next already–created face (if any).
PAT_IGNORE: behave as though there is no pattern on the face.

```
public: FACE* FACE::next_in_list (
    PAT_NEXT_TYPE next_type //
        = PAT_CAN_CREATE    //
    ) const;
```

Returns a pointer to the next FACE in the list of FACEs contained directly by a SHELL or SUBSHELL.

```
public: ENTITY* FACE::owner () const;
```

Returns a pointer to the owning entity.

```
public: logical FACE::patternable () const;
```

Returns TRUE.

```
public: logical FACE::remove_from_pattern ();
```

Removes the pattern element associated with this entity from the pattern. Returns FALSE if this entity is not part of a pattern element, otherwise TRUE.

Note *The affected entities are not destroyed, but are merely made independent of the pattern. The pattern itself is correspondingly modified to "drop out" the newly disassociated element.*

```
public: logical FACE::remove_from_pattern_list ();
```

Removes this entity from the list of entities maintained by its pattern, if any. Returns FALSE if no pattern is found, otherwise TRUE.

```
public: logical FACE::remove_pattern ();
```

Removes the pattern on this and all associated entities. Returns FALSE if no pattern is found, otherwise TRUE.

```
public: void FACE::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 >= PATTERN_VERSION	
read_ptr	Pointer to record in save file for APATTERN on loop
if (apat_idx != (APATTERN*)(–1)))	
restore_cache();	
read_ptr	Pointer to record in save file for next FACE in shell or subshell
read_ptr	Pointer to record in save file for first LOOP bounding face
read_ptr	Pointer to record in save file for SHELL containing face
read_ptr	Pointer to record in save file for SUBSHELL containing face
read_ptr	Pointer to record in save file for SURFACE on which face lies
read_logical	(“forward”, “reversed”) Direction of face normal with respect to the surface
if (restore_version_number >= TWOSIDE_VERSION)	
read_logical	(“single”, “double”) Double sided face
if (sides_data)	if double sided face
read_logical	(“out”, “in”), Double sided face containment. Containment data.
else	containment data is FALSE.
else	Side data is SINGLE_SIDED and containment data is FALSE.

```
public: REVBIT FACE::sense () const;
```

Returns the sense of the FACE (FORWARD or REVERSED) relative to the SURFACE. Every SURFACE has a direction sense. Its normal direction is a continuous function of position. The normal to the FACE can be the same as that of the SURFACE at any position, or can be the reverse of it, as determined by sense. When a FACE bounds a region of space, its normal always points away from the region bounded.

```
public: REVBIT FACE::sense (
    REVBIT rev              // sense
    ) const;
```

Return the sense of the FACE compounded with the sense argument. Useful when traversing the FACE in a reverse direction.

```
public: void FACE::set_bound (
    SPAbox*                 // bounding box
    );
```

Sets the FACE's bounding SPAbox pointer to the given SPAbox. Before performing a change, it checks if the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void FACE::set_cont (
    CONTBIT,                // containment bit
    logical reset_pattern   // reset or not
    = TRUE
    );
```

Sets the FACE's containment bit to indicate whether the FACE is fully contained within the parent SHELL or not. Before performing a change, it checks if the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void FACE::set_geometry (
    SURFACE*,               // new SURFACE geometry
    logical reset_pattern   // reset or not
    = TRUE
    );
```

Sets the FACE's geometry pointer to the given SURFACE. A side effect of this method is the routine adjusts the use counts of the existing and new geometry and deletes the old if it is no longer referenced. Before performing a change, each routine checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void FACE::set_loop (
    LOOP*,                  // new LOOP
    logical reset_pattern   // reset or not
    = TRUE
    );
```

Sets the FACE's loop pointer to the given LOOP. Before performing a change, each routine checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void FACE::set_next (
    FACE*,                  // next FACE
    logical reset_pattern   // reset or not
    = TRUE
    );
```

Sets the FACE's next FACE pointer to the given FACE. Before performing a change, each routine checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void FACE::set_pattern (
    pattern* in_pat             // pattern
    );
```

Set the current pattern.

```
public: void FACE::set_sense (
    REVBIT,                 // sense
    logical reset_pattern   // reset or not
    = TRUE
    );
```

Sets the FACE's sense to FORWARD or REVERSED with respect to the SURFACE. Before performing a change, each routine checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void FACE::set_shell (
    SHELL*,                 // SHELL
    logical reset_pattern   // reset or not
    = TRUE                  //
    );
```

Sets the FACE's SHELL pointer to the given SHELL. Before performing a change, each routine checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void FACE::set_sides (
    SIDESBIT,                // sidedness
    logical reset_pattern    // reset or not
    = TRUE
    );
```

Sets the FACE's sides to single or double sided. Before performing a change, each routine checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void FACE::set_subshell (
    SUBSHELL*,               // SUBSHELL
    logical reset_pattern    // reset or not
    = TRUE
    );
```

Sets the FACE's SUBSHELL pointer to the given SUBSHELL. Before performing a change, each routine checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

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

Returns a pointer to the SHELL containing the FACE, either directly or through a hierarchy of SUBSHELLs.

```
public: SIDESBIT FACE::sides () const;
```

Returns SINGLE_SIDED if the FACE is single-sided, DOUBLE_SIDED if double-sided.

```
public: SUBSHELL* FACE::subshell () const;
```

Returns a pointer to the SUBSHELL containing FACE directly. The return is NULL if the FACE belongs directly to the owning SHELL.

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

Returns the string "face".

Internal Use: next_face, save, save_common

Related Fncs:

is_FACE

FileInfo

Class: SAT Save and Restore

Purpose: Contains additional required file header information.

Derivation: FileInfo : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerndata/savres/fileinfo.hxx

Description: Contains additional file header information, such as ID of the product used to save the model, ACIS version, millimeters per model unit, date model was saved, ACIS save file version, and other relevant model data.

Beginning with ACIS Release 6.3, it is **required** that the product ID and units scale be populated for the file header before you can save a SAT file, regardless of the save file version. If you do not set both of these data, ACIS will generate an error. Refer to the set_product_id and set_units methods of this class.

Limitations: None

References: None

Data:

None

Constructor:

```
public: FileInfo::FileInfo ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: FileInfo::FileInfo (
    FileInfo const&          // file name
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

Destructor:

```
public: FileInfo::~FileInfo ();
```

C++ destructor for FileInfo which deallocates memory.

Methods:

```
public: const char* FileInfo::acis_version () const;
```

Returns the ACIS version number used to save the model.

```
public: const char* FileInfo::date () const;
```

Returns the date on the save file.

```
public: int FileInfo::file_version () const;
```

Returns the save file version used in storing the file.

```
public: FileInfo& FileInfo::operator= (
    FileInfo const&         // file name
    );
```

Performs an assignment operation.

```
public: const char* FileInfo::product_id () const;
```

Returns the ID of the product.

```
public: void FileInfo::reset ();
```

Resets the values to the default settings for the file information.

```
public: void FileInfo::reset_vars ();
```

Routine to reset the values for the file information to the default values.

```
public: void FileInfo::restore ();
```

Restores the file information from a save file.

```
if (restore_version_number >= FILEINFO_VERSION)
    read_string                 Product
    read_string                 ACIS Version
    read_string                 Date
    read_real                   Units
    read_real                   Tolerance, abs
    read_real                   Tolerance, nor
```

```
public: void FileInfo::save ();
```

Saves the product ID, version, time, units, SPAresabs, and SPAresnor.

```
public: void FileInfo::set_masked (
    unsigned long,          // number of fields
    FileInfo const&         // file name
    );
```

Copy selected fields from another instance.

```
public: void FileInfo::set_product_id (
    const char*             // ID
    );
```

Sets the product ID. The product ID can be any string greater than 4 characters. *Spatial* recommends that the string contain your product name and the product version, so that SAT files generated by your product can be easily identified.

Beginning with ACIS Release 6.3, it is **required** that the product ID and units scale be populated for the file header before you can save a SAT file.

```
public: void FileInfo::set_units (
    double                  // number of millimeters
    );
```

Sets the model units scale (in millimeters). Set the scale to the appropriate units for your product (1.0 indicates that 1 unit equals 1 millimeter).

Beginning with ACIS Release 6.3, it is **required** that the product ID and units scale be populated for the file header before you can save a SAT file.

```
public: double FileInfo::tol_abs () const;
```

Returns the value of the SPAresabs when the model was saved.

```
public: double FileInfo::tol_nor () const;
```

Returns the value of the SPAresnor when the model was saved.

```
public: double FileInfo::units () const;
```

Returns the value of the millimeters per model unit.

```
public: void FileInfo::valid ();
```

Checks the values of the units and product id.

Related Fncs:

None

FileInterface

Class: SAT Save and Restore

Purpose: Defines the abstract base class.

Derivation: FileInterface : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/fileio/fileif.hxx

Description: Defines the abstract base class that defines the interface that ACIS uses to save and restore ENTITY data.

All ACIS save and restore operations use an object of this class to control the reading or writing of the data. There are two main reasons for having this class:

The first reason is to allow saving and restoring ENTITY data to targets other than a standard C stream file; i.e., a FILE*. To do this derive a new subclass from this one, which implements reading and writing for the new target.

The second reason is to allow saving and restoring unknown ENTITY data in binary format. This is the reason that this class has so many virtual methods. To support unknown ENTITY data, the data is tagged with its type when it is written to a file. This allows manipulation of the data when it is loaded back in even if the data is unknown.

When deriving a new class to support a different kind of storage target,derive it from the BinaryFile class, which is derived from this one and is declared in binfile.hxx. The BinaryFile class has a standard implementation for most of the virtual methods of this class that already take care of the details for saving and restoring the unknown ENTITY data. Implementation of the actual read and write methods are all that is necessary.

Limitations: None

References: None

Data:

None

Constructor:

```
public: FileInterface::FileInterface ();
```

C++ constructor, creating a FileInterface.

Destructor:

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

C++ destructor, deleting a FileInterface.

Methods:

```
public: virtual FilePosition
    FileInterface::goto_mark (
    FilePosition            // file position.
    ) = 0;
```

Goes to the mark.

```
public: virtual char FileInterface::read_char () = 0;
```

Reads a character. Written with C printf format “%c”.

```
public: virtual TaggedData*
    FileInterface::read_data ();
```

Reads the data.

```
public: virtual double
    FileInterface::read_double () = 0;
```

Reads a double. Written with C printf format “%g ”.

```
public: virtual int FileInterface::read_enum (
    enum_table const&       // enumeration table
    ) = 0;
```

Read an enumeration table. The <identifier> specifies which enumeration is active and its valid values. The <identifier> is not written to the file. A valid value only is written to the file. This is a character string or a long value from the enumeration <identifier> written with C printf format “%s”.

```
public: virtual float
    FileInterface::read_float () = 0;
```

Reads a float. Written with C printf format “%g ”.

```
public: virtual logical FileInterface::read_header (
    int&,                   // first integer
    int&,                   // second integer
    int&,                   // third integer
    int&                    // fourth integer
    );
```

Reads a header. The first record of the ACIS save file is a header, such as:
200 0 1 0

First Integer: An encoded version number. In the example, this is “200”. This value is 100 times the major version plus the minor version (e.g., 107 for ACIS version 1.7). For point releases, the final value is truncated. Part save data for the .sat files is not affected by a point release (e.g., 105 for ACIS version 1.5.2).

Second Integer: The total number of saved data records, or zero. If zero, then there needs to be an end mark.

Third Integer: A count of the number of entities in the original entity list saved to the part file.

Fourth Integer: The least significant bit of this number is used to indicate whether or not history has been saved in this save file.

```
public: virtual int FileInterface::read_id (
    char*,                  // ID string buffer
    int                     // buffer length
        = 0
    ) = 0;
```

Reads an identifier. The save identifier written with C printf format “%s ”.

```
public: virtual logical FileInterface::read_logical (
    const char* f           // FALSE keyword
        = ”F”,
    const char* t           // TRUE keyword
        = ”T”
    ) = 0;
```

Reads a logical. *(false_string, true_string, {or any_valid_string})*: Appropriate string written with C printf format "%s ".

```
public: virtual long FileInterface::read_long () = 0;
```

Reads a long. Written with C printf format "%ld".

```
public: virtual void*
    FileInterface::read_pointer () = 0;
```

Reads a pointer. Pointer reference to a save file record index. Written as "$" followed by index number written as a long.

```
public: virtual SPAposition
    FileInterface::read_position ();
```

Reads the position. *x, y, z* coordinates written as real numbers.

```
public: virtual int FileInterface::read_sequence ();
```

Reads a sequence. Written as "–" followed by the entity index written as long.

```
public: virtual short
    FileInterface::read_short () = 0;
```

Reads a short. Written with C printf format "%d".

```
public: virtual char* FileInterface::read_string (
    int& len                // length
    ) = 0;
```

Reads a string, allocates memory for it, and the argument returns the length of the string. Length written as long followed by string written with C printf format "%s".

```
public: virtual size_t FileInterface::read_string (
    char* buf,              // buffer
    size_t maxlen           // maximum length
        = 0
    ) = 0;
```

Reads a string, allocates memory for it, and the argument returns the length of the string. Length written as long followed by string written with C printf format “%s”.

```
public: virtual logical
    FileInterface::read_subtype_end () = 0;
```

Reads subtype end. Braces around the subtypes, written as “} ” in the SAT file.

```
public: virtual logical
    FileInterface::read_subtype_start () = 0;
```

Reads subtype start. Braces around the subtypes, written as “{ ” in the SAT file.

```
public: virtual SPAvector FileInterface::read_vector
();
```

Reads the vector. *x, y, z* components written as real numbers.

```
public: virtual FilePosition
    FileInterface::set_mark () = 0;
```

Sets the mark.

```
public: virtual logical
    FileInterface::unknown_types_ok ();
```

Determines if unknown ENTITY types are OK. This returns TRUE for everything except old style binary files, so it has a default implementation.

```
public: virtual void FileInterface::write_char (
    char                    // character
    ) = 0;
```

Writes a character. Written with C printf format “%c”.

```
public: virtual void FileInterface::write_data (
    const TaggedData&       // tagged data
    );
```

Writes the data.

```
public: virtual void FileInterface::write_double (
    double                  // real
    ) = 0;
```

Writes a real. Written with C printf format “%g ”.

```
public: virtual void FileInterface::write_enum (
    int,                    // value
    enum_table const&       // enumeration table
    ) = 0;
```

Writes enumeration table. The <identifier> specifies which enumeration is active and its valid values. The <identifier> is not written to the file. A valid value only is written to the file. This is a character string or a long value from the enumeration <identifier> written with C printf format “%s”.

```
public: virtual void FileInterface::write_float (
    float                   // float
    ) = 0;
```

Writes a float. Written with C printf format “%g ”.

```
public: virtual void FileInterface::write_header (
    int,                    // first integer
    int,                    // second integer
    int,                    // third integer
    int                     // fourth integer
    );
```

Writes a header. The first record of the ACIS save file is a header, such as:
200 0 1 0

First Integer: An encoded version number. In the example, this is “200”. This value is 100 times the major version plus the minor version (e.g., 107 for ACIS version 1.7). For point releases, the final value is truncated. Part save data for the .sat files is not affected by a point release (e.g., 105 for ACIS version 1.5.2).

Second Integer: The total number of saved data records, or zero. If zero, then there needs to be an end mark.

Third Integer: A count of the number of entities in the original entity list saved to the part file.

Fourth Integer: The least significant bit of this number is used to indicate whether or not history has been saved in this save file.

```
public: virtual void FileInterface::write_id (
    const char*,            // ID string
    int                     // ID level (1 or 2)
    ) = 0;
```

Writes an identifier. The save identifier written with C printf format “%s ”.

```
public: virtual void
    FileInterface::write_literal_string (
    const char*,            // string
    size_t len              // length
        = 0
    );
```

Writes a literal string.

```
public: virtual void FileInterface::write_logical (
    logical,                // logical value
    const char* f           // FALSE keyword
        = ”F”,
    const char* t           // TRUE keyword
        = ”T”
    ) = 0;
```

Writes a logical. *(false_string, true_string, {or any_valid_string})*: Appropriate string written with C printf format “%s ”.

```
public: virtual void FileInterface::write_long (
    long                    // long
    ) = 0;
```

Writes a long. Written with C printf format “%ld”.

```
public: virtual void FileInterface::write_newline (
    int                     // number of new lines
        = 1
    );
```

Writes a new line character.

```
public: virtual void FileInterface::write_pointer (
    void*                   // pointer
    ) = 0;
```

Writes a pointer. Pointer reference to a save file record index. Written as "$" followed by index number written as a long.

```
public: virtual void FileInterface::write_position (
    const SPAposition&      // position
    );
```

Writes a position. *x, y, z* coordinates written as real numbers.

```
public: virtual void FileInterface::write_sequence (
    int                     // integer
    );
```

Writes a sequence. Written as "–" followed by the entity index written as long.

```
public: virtual void FileInterface::write_short (
    short                   // short
    ) = 0;
```

Writes a short. Written with C printf format "%d".

```
public: virtual void FileInterface::write_string (
    const char*,            // string
    size_t len              // length
        = 0
    ) = 0;
```

Writes a string. Length written as long followed by string written with C printf format "%s".

```
public: virtual void
    FileInterface::write_subtype_end () = 0;
```

Writes a subtype end. Braces around the subtypes, written as "} " in the SAT file.

```
public: virtual void
    FileInterface::write_subtype_start () = 0;
```

Writes a subtype start. Braces around the subtypes, written as “{ ” in the SAT file.

```
public: virtual void
    FileInterface::write_terminator () = 0;
```

Writes a terminator. Written as “#” in the SAT file.

```
public: virtual void FileInterface::write_vector (
    const SPAvector&        // vector name
    );
```

Writes a vector. *x*, *y*, *z* components written as real numbers.

Related Fncs:

None

gedge

Class: Graph Theory

Purpose: Creates an instance of a graph edge for use in graph theory.

Derivation: gedge : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/law/generic_graph.hxx

Description: The concepts of vertex, edge, and graph have been implemented as the C++ classes gvertex, gedge, and generic_graph. (entity_gvertex is derived from gvertex except that it contains a pointer to an entity in the model. Such an entity could be a cell or a face.) A gvertex may be created with an optional char *name. A gedge may be created with two gvertex pointers. An empty graph may be created and edges and vertices may be added to it by calling its add_vertex and add_edge methods. Once created, a graph may be interrogated, ordered, or subsetted in a number of ways.

The C++ classes of gvertex and gedge are use counted in the same way that laws are use counted. That is to say that the are copied by calling the add method and deleted by calling the remove method.

To make a gvertex or gedge contain data, derive a class from the base classes of gvertex and gedge. Use a technique similar to the entity_gvertex class which enables it to contain an entity pointer.

Limitations: None

References: KERN gvertex

Data:

```
protected gvertex *v1;
```
First vertex of the edge.

```
protected gvertex *v2;
```
Second vertex of the edge.

```
public static int how_many;
```
Keeps track of how many gvertexes have been created.

Constructor:

```
public: gedge::gedge (
    gvertex const* in_v1,    // vertex one
    gvertex const* in_v2,    // vertex two
    double in_weight         // weight of the edge
        = 0.0
    );
```

Creates an instance of gedge between the two graph vertices supplied.

```
public: gedge::gedge ();
```

C++ allocation constructor requests memory for this object but does not populate it.

Destructor:

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

Do not call this destructor directly. An instance of gedge is deleted by calling the remove method. This is necessary, because gedge is use counted. This destructor will throw a sys_error if it is called when its use_count is not equal to zero.

Methods:

```
public: void gedge::add () const;
```

The C++ classes of gvertex and gedge are use counted in the same way that laws are use counted. That is to say that the are copied by calling the add method and deleted by calling the remove method.

```
public: void gedge::clear_kind ();
```

Sets the user-defined kind array for this graph item to NULL. kind is actually a dynamic array. The value argument specifies whether or not this graph edge is of the kind number specified.

```
public: virtual ENTITY* gedge::get_entity () const;
```

Returns NULL.

```
public: int gedge::get_kind_size () const;
```

Returns the number of entries in the kind array.

```
public: double gedge::get_weight () const;
```

Gets the weight of the gedge.

```
public: logical gedge::is_kind (
    int which               // kind to test
    ) const;
```

Tests to see if this instance of the graph edge is of a particular (user-defined) kind.

```
public: logical gedge::is_loop () const;
```

Returns TRUE if the first vertex is the same as the last vertex, thus forming a loop.

```
public: logical gedge::operator!= (
    gedge const& in_edge    // test graph edge
    ) const;
```

Determines whether or not the supplied graph edge is not equal to this graph edge.

```
public: logical gedge::operator== (
    gedge const& in_edge    // test graph edge
    ) const;
```

Determines whether or not the supplied graph edge is equal to this graph edge.

```
public: void gedge::remove ();
```

The C++ classes of gvertex and gedge are use counted in the same way that laws are use counted. That is to say that the are copied by calling the add method and deleted by calling the remove method.

```
public: void gedge::set_kind (
    int which,              // kind to use
    logical value           // turn on or off
    );
```

Assigns a user-defined kind to this graph edge. kind is actually a dynamic array. The value argument specifies whether or not this graph edge is of the kind number specified.

```
public: void gedge::set_weight (
    double in_weight        // weight
    ) const;;
```

Sets the weight of the gedge.

```
public: gvertex const* gedge::vertex1 () const;
```

Returns the first vertex associated with this graph edge.

```
public: gvertex const* gedge::vertex2 () const;
```

Returns the second vertex associated with this graph edge.

Internal Use: id, isa, same, type

Related Fncs:

None

generic_graph

Class: Graph Theory

Purpose: Creates an instance of a graph for the graph theory mathematical operations.

Derivation: generic_graph : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/law/generic_graph.hxx

Description: The concepts of vertex, edge, and graph have been implemented as the C++ classes gvertex, gedge, and generic_graph. (entity_gvertex is derived from gvertex except that it contains a pointer to an entity in the model. Such an entity could be a cell or a face.) A gvertex may be created with an optional char *name. A gedge may be created with two gvertex pointers. An empty graph may be created and edges and vertices may be added to it by calling its add_vertex and add_edge methods. Once created, a graph may be interrogated, ordered, or subsetted in a number of ways.

The generic_graph class has methods to tell if a graph is connected, a tree, linear, or a cycle. It also has methods to tell how many components the graph has and to return each of the components as a subgraph. Moreover, the components may be identified by giving an edge or vertex in them.

Limitations: None

References: None

Data:

None

Constructor:

```
public: generic_graph::generic_graph (
    char const* in_str      // name of the graph
        = NULL
    );
```

Creates a graph with the given name.

Destructor:

```
protected: generic_graph::~generic_graph ();
```

Destructor for the graph. This destructor will throw a sys_error if it is called when its use_count is not equal to zero.

Methods:

```
public: void generic_graph::add () const;;
```

Increments the use count of how many references there are to this generic_graph instance. The object will not be destroyed until all references to it have been removed.

```
public: void generic_graph::add_edge (
    char const*             // name for edge
    );
```

Adds the named graph edge to a graph structure.

```
public: void generic_graph::add_edge (
    gedge const*            // pointer to edge
    );
```

Adds the specified graph edge to the graph structure using its pointer.

```
public: void generic_graph::add_edge (
    gvertex const*,         // first vertex of edge
    gvertex const*,         // 2nd vertex of edge
    ENTITY* in_ent          // optional entity to
        = NULL              // associate with edge
    );
```

Adds a graph edge to the graph structure by specifying pointers to its vertices.

```
public: void generic_graph::add_edge (
    gvertex const*,         // first vertex of edge
    gvertex const*,         // 2nd vertex of edge
    double weight           // weight
        = 0.0
    );
```

Adds a graph edge to the graph structure by specifying pointers to its vertices. Optionally an entity can be associated with the graph edge, to, for example, tie the graph to features of a geometric model.

```
public: void generic_graph::add_vertex (
    char const*             // name of vertex
    );
```

Adds a vertex to the graph structure by specifying its name.

```
public: void generic_graph::add_vertex (
    gvertex const*          // pointer to vertex
    );
```

Adds a vertex to the graph structure by specifying a pointer to the graph vertex.

```
public: logical generic_graph::adjacent (
    gvertex const*,          // first gvertex
    gvertex const*           // second gvertex
    ) const;
```

Determines if the two specified gvertexes share a common gedge.

```
public: generic_graph* generic_graph::branch (
    generic_graph* trunk,    // linear trunk portion
                             // of the graph
    generic_graph* which,    // sub-portion of trunk
                             // from which to collect
                             // branches
    logical keep_trunk       // if true, include
                             // segments of the trunk.
                             // if false, include only
                             // branches.
    ) const;
```

Returns a graph of branches off of a specified portion of the given trunk.

```
public: generic_graph* generic_graph::branch (
    generic_graph* trunk,    // linear trunk portion
                             // of the graph
    int order,               // n'th gvertex on trunk
    logical keep_trunk       // if true, include
                             // segments of the trunk.
                             // if false, include only
                             // branches.
    ) const;
```

Returns the branch(es) from a specific ordered gvertex of the trunk.

```
public: void generic_graph::clear_kind ();
```

Sets the user-defined kind array for this graph item to NULL. kind is actually a dynamic array that can be used to assign arbitrary flags to gedges and gvertexes on the graph.

```
public: generic_graph* generic_graph::component (
    int                     // number of components
    ) const;
```

Specifies the number of components that part of the graph structure.

```
public: int generic_graph::component (
    gedge const*            // pointer to edge
    ) const;
```

Returns a number representing the component to which the graph edge belongs.

```
public: int generic_graph::component (
    gvertex const*          // pointer to vertex
    ) const;
```

Returns the number representing the component to which the graph vertex belongs.

```
public: int generic_graph::components () const;
```

Returns the number of components in the graph structure.

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

Copies the graph structure into another graph structure.

```
public: generic_graph*
    generic_graph::cut_edges () const;
```

This returns a new graph structure containing all of the graph edges that are considered cut edges. Cut edges are defined as those edges whose removal results in more graph components than were originally present.

```
public: generic_graph*
    generic_graph::cut_vertices () const;
```

This returns a new graph structure containing all of the graph vertices that are considered cut vertices. Cut vertices are defined as those vertices whose removal results in more graph components than were originally present.

```
public: generic_graph*
    generic_graph::cycle_edges () const;
```

This returns a new graph structure containing all of the graph edges that are considered cycle edges. Cycle edges are defined as the shortest path through a graph structure resulting in a closed loop.

```
public: int generic_graph::degree (
    gvertex const*          // pointer to vertex
    ) const;
```

Returns the degree of the specified graph vertex.

```
public: int generic_graph::find_all_edges_by_vertex (
    gvertex const*,         // first gvertex
    gvertex const*,         // second gvertex
    gedge**& out            // ge_list
        = * (gedge** *) NULL_REF,
    int                     // target
        = 0
    ) const;
```

Uses the two given gvertexes to find all gedges that connects the gvertexes. This method returns the number of gedges found. ge_list and target are optional. ge_list returns the gedges found. The caller may specify how many gedges are required by setting a target number. The default target is 0, which gets all gedges.

```
public: gedge const*
    generic_graph::find_edge_by_entities (
    ENTITY* ent1,           // first entity
    ENTITY* ent2            // second entity
    ) const;
```

Uses the two given entities to find two gvertex's, then uses these two gvertex's to find a gedge that is defined by them. Returns NULL if such gedge does not exist.

```
public: gedge const*
    generic_graph::find_edge_by_name (
    char const* v1          // name to search
    ) const;
```

Locates a graph edge in the graph structure by its specified name.

```
public: gedge const*
    generic_graph::find_edge_by_vertex (
    gvertex const*,          // first vertex
    gvertex const*,          // second vertex
    ENTITY const* ref_ent    // optionally return only
        = NULL               // gedges associated with
                             // a particular entity
    ) const;
```

Locates a graph edge in the graph structure by its bounding vertices.

```
public: generic_graph*
    generic_graph::find_shortest_cycle (
    gvertex const*           // starting vertex
    ) const;
```

Returns a graph structure which represents the shortest cycle that contains the given graph vertex.

```
public: generic_graph*
    generic_graph::find_shortest_path (
    gvertex const*,          // starting vertex
    gvertex const*,          // ending vertex
    logical weighted         // shortest (false) or
    = FALSE                  // lightest (true)
    ) const;
```

Returns a graph structure that represents the shortest path between the two specified graph vertices. If weighted is FALSE, the method will return the shortest path (fewest number of gvertexes in it.) If weighted is TRUE, the method will return the lightest path (where the sum of all the weights applied to gvertexes and gedges is lowest.)

```
public: gvertex const*
    generic_graph::find_vertex_by_entity (
    ENTITY* ent              // entity to search
    ) const;
```

Returns a pointer to the graph vertex by following its model entity.

```
public: gvertex const*
    generic_graph::find_vertex_by_name (
    char const* name        // name to search
    ) const;
```

Returns a pointer to the named graph vertex.

```
public: gedge** generic_graph::get_adjacent_edges (
    gvertex const*,         // test vertex
    int& size               // number of edges
    ) const;
```

Returns an array of graph edges that are adjacent to the specified vertex. User must supply a pointer to a variable representing the size of the array.

```
public: gvertex**
    generic_graph::get_adjacent_vertices (
    gvertex const*,         // test vertex
    int& size               // number of vertices
    ) const;
```

Returns an array of graph vertices that are adjacent to the specified vertex. User must supply a pointer to a variable representing the size of the array.

```
public: gedge** generic_graph::get_edges (
    int& size               // number of edges
    ) const;
```

Returns an array of graph edges that are part of the graph structure. User must supply a pointer to a variable representing the size of the array.

```
public: void generic_graph::get_entities (
    ENTITY_LIST&,           // pointer to entities
    logical use_ordering    // ordering on or off
        = FALSE
    ) const;
```

Lists all entities associated with all gedges and gvertexes of the graph.

```
public: void generic_graph::get_entities_from_edge (
    ENTITY_LIST&            // list of entities
    ) const;
```

Lists all entities associated with all gedges of the graph

```
public: void
    generic_graph::get_entities_from_vertex (
    ENTITY_LIST&,            // list of entities
    logical use_ordering     // ordering on or off
        = FALSE
    ) const;
```

Lists all entities associated with all gvertexes of the graph

```
public: gvertex** generic_graph::get_leaves (
    int& size                // size of returned array
    ) const;
```

Gets a list of all the gvertexes with exactly one gedge (leaves of the tree.)

```
public: int generic_graph::get_order (
    gvertex const*           // pointer to vertex
    ) const;
```

Once a graph has been ordered, the order of a vertex may be found by calling the get_order method.

```
public: gvertex** generic_graph::get_vertices (
    int& size                // number of vertices
    ) const;
```

Returns an array of graph vertices that make up the graph structure. User must supply a pointer to a variable which represents the size of the array.

```
public: generic_graph* generic_graph::intersect (
    generic_graph*           // test graph
    ) const;
```

Returns a graph structure that represents the intersection of this graph structure with the specified test graph structure.

```
public: logical generic_graph::is_connected () const;
```

Determines whether or not the graph is connected.

```
public: logical generic_graph::is_cut_edge (
    gedge const*            // test edge
    ) const;
```

Determines whether or not the specified graph edge is a cut edge.

```
public: logical generic_graph::is_cut_vertex (
    gvertex const*          // test vertex
    ) const;
```

Determines whether or not the specified graph vertex is a cut vertex.

```
public: logical generic_graph::is_cycle () const;
```

Determines whether or not the graph structure is cyclic.

```
public: logical generic_graph::is_cycle_vertex (
    gvertex const*          // test vertex
    ) const;
```

Determines whether or not the specified graph vertex is a cycle vertex.

```
public: logical generic_graph::is_linear () const;
```

Determines whether or not the graph structure is linear.

```
public: logical generic_graph::is_multiple_edge (
    gedge const*            // gedge
    ) const;
```

Returns TRUE if there is more than one gedge spanning this gedge's vertices.

```
public: logical generic_graph::is_simple (
    gedge const*            // gedge
    ) const;
```

Returns TRUE if the graph has no multiple edges.

```
public: logical generic_graph::is_subset (
    generic_graph const*     // graph that might be a
                             // subset of the THIS
                             // graph.
    ) const;
```

Returns TRUE if in_graph is a subset of the THIS graph.

```
public: logical generic_graph::is_tree () const;
```

Determines whether or not the graph structure is a tree.

```
public: generic_graph* generic_graph::kind (
    int which,               // kind to test
    logical value            // on or off
        = TRUE
    ) const;
```

This assigns a user-defined kind and its on/off status to the graph structure.

```
public: int generic_graph::max_kind () const;
```

Returns the largest number of kinds used to mark any gvertex or gedge. This is useful for determining the number of the next unused kind.

```
public: int generic_graph::max_order () const;
```

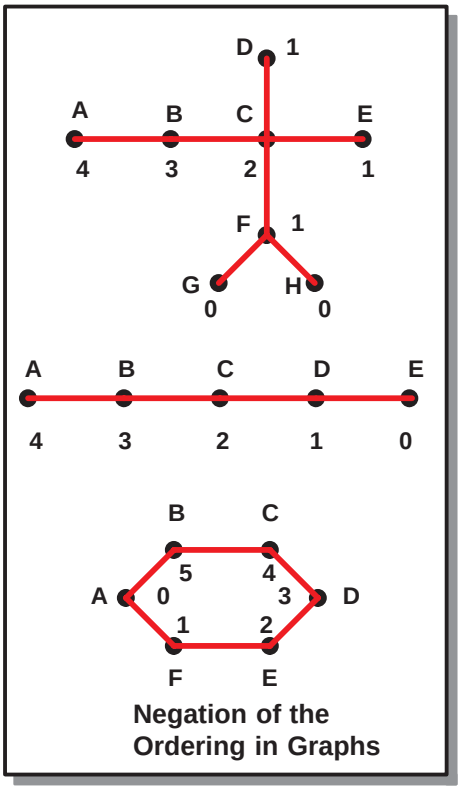
Once a graph has been ordered, the maximum order in the graph may be found by calling the max_order method.

```
public: int generic_graph::min_order () const;
```

Once a graph has been ordered, the minimum order in the graph may be found by calling the min_order method.

```
public: void generic_graph::negate ();
```

Once a graph has been ordered, its ordering may be negated with this method. Negation is a special operation and returns different results for cycles and trees depending upon the start vertex. In the following figure, the graph vertex A was initially 0 in the ordering before the negation operation.



Negation of the Ordering in Graphs

```
public: int generic_graph::number_of_edges () const;
```

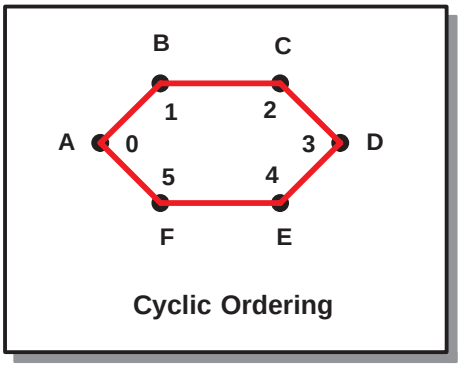
Returns the number of graph edges in the graph structure.

```
public: int generic_graph::number_of_vertices ()
const;
```

Returns the number of graph vertices in the graph structure.

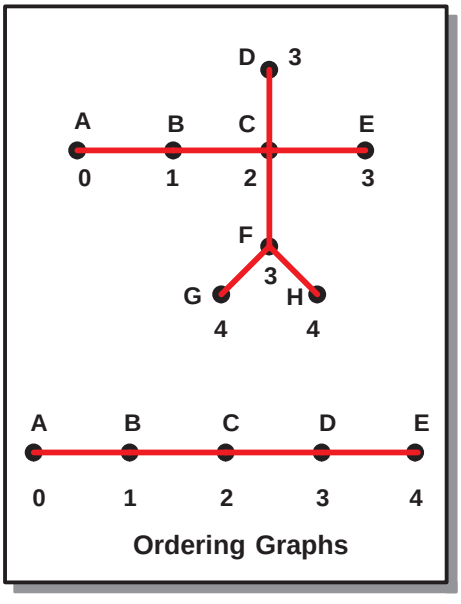
```
public: void generic_graph::order_cyclic (
    gvertex const*,          // first vertex
    gvertex const*           // last vertex
    );
```

If a graph is cyclic, then it may be ordered by the order_cyclic method. This sets a given vertex's order to zero and the other vertices in a cyclic order as shown in the following figure.




```
public: void generic_graph::order_from (
    generic_graph*          // graph
    );
```

The order–from method of ordering a graph works well for trees and linear graphs. The two graphs show in the Ordering Graphs figure have been ordered by distance from vertex A.



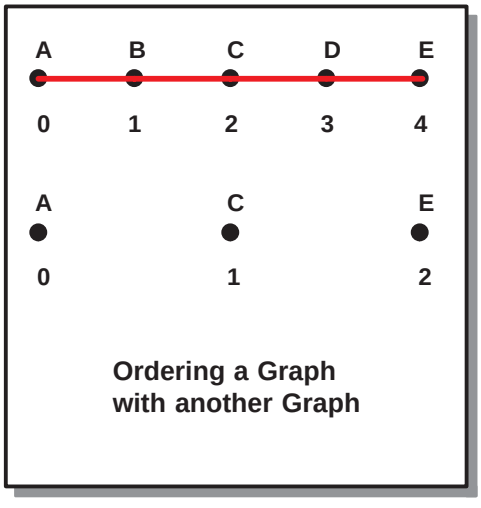
```
public: void generic_graph::order_from (
    gvertex const*          // starting vertex
    );
```

The order–from method of ordering a graph works well for trees and linear graphs. The two graphs show in the Ordering Graphs figure have been ordered by distance from vertex A.

```
public: void generic_graph::order_with (
    generic_graph*,         // other graph
    logical compress        // remove gaps in
                            // ordering if TRUE
        = TRUE
    );
```

Another way to order a graph G is to order it with respect to an ordered graph H such that G is a subgraph of H. The order_with method imposes the order of H onto G and rescales the ordering on G to remove gap. The type of ordering (i.e. cyclic or not) is inherited from the ordered graph H. If the compress option is turned on, the resulting gvertexes are numbered sequentially. In the example below, gvertexes in the uncompressed result would be numbered 124, but in the compressed result would be numbered 012.

The following figure shows a linear graph imposing its order on a subgraph.



```
public: void generic_graph::remove ();
```

Decrements the use count for the generic graph, and destroys the object when the use count reaches zero.

```
public: void generic_graph::set_kind (
    generic_graph*,          // reference graph
    int which,               // which kind
    logical value            // turn kind on if TRUE,
        = TRUE               // off if FALSE
    );
```

Turn the given kind on or off for all gvertexes and gedges in the reference graph. The reference graph is a subset of the full graph.

```
public: void generic_graph::set_order (
    gvertex const*,          // gvertex
    int                      // order
    );
```

Manually assigns an order to a gvertex in a graph.

```
public: int generic_graph::split_branches (
    generic_graph**& out_graphs // subgraph list
    );
```

Finds all branches in the graph and return a set of subgraphs that do not have a branch.

```
public: generic_graph* generic_graph::subset (
    int,                     // integer a
    int                      // integer b
    ) const;
```

The subset method with two integers takes a and b and returns a subgraph in one of two ways.

If a<b, then the set of all vertices with orders between a and b is returned along with all edges that have both of their adjacent vertices in this set.

If b<a, then the set of all vertices with orders not between a and b is returned along with all edges that have both of their adjacent vertices in this set.

```
public: generic_graph* generic_graph::subset (
    law*                    // law for evaluation
    ) const;
```

The subset method with a law returns the set of all vertices such that their order evaluates as true along with the all edges that have both of their adjacent vertices evaluating as true orders.

```
public: generic_graph* generic_graph::subtract (
    generic_graph*,         // graph to remove
    logical keep            // flag for keep
    ) const;
```

Removes the specified graph from this graph structure.

```
public: generic_graph*
    generic_graph::subtract_edges (
    generic_graph*          // input graph
    ) const;
```

Subtracts the gedges of the input graph from the full graph.

```
public: double generic_graph::total_weight () const;
```

Returns the sum of the weights of all the gedges in the graph.

```
public: generic_graph* generic_graph::unite (
    generic_graph*          // graph to add
    ) const;
```

Unites this graph with the specified graph. Graph edges and vertices only appear once.

```
public: logical generic_graph::vertex_exists (
    gvertex const*  in_vertex   // gvertex
    );
```

Returns TRUE if the given gvertex exists in the graph.

Internal Use: get_root, mark_branches

Related Fncs:

None

gvertex

Class: Graph Theory

Purpose: Creates an instance of a graph vertex for use in graph theory.

Derivation: gvertex : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/law/generic_graph.hxx

Description: The concepts of vertex, edge, and graph have been implemented as the C++ classes gvertex, gedge, and generic_graph. (entity_gvertex is derived from gvertex except that it contains a pointer to an entity in the model. Such an entity could be a cell or a face.) A gvertex may be created with an optional char *name. A gedge may be created with two gvertex pointers. An empty graph may be created and edges and vertices may be added to it by calling its add_vertex and add_edge methods. Once created, a graph may be interrogated, ordered, or subsetted in a number of ways.

The C++ classes of gvertex and gedge are use counted in the same way that laws are use counted. That is to say that the are copied by calling the add method and deleted by calling the remove method.

To make a gvertex or gedge contain data, derive a class from the base classes of gvertex and gedge. Use a technique similar to the entity_gvertex class which enables it to contain an entity pointer.

Limitations: None

References: by KERN gedge

Data:

```
protected char *internal_name;
```
Character representation used to refer to this vertex.

```
public static int how_many;
```
Keeps track of how many gvertexes have been created.

Constructor:

```
public: gvertex::gvertex (
    char const* name        // name of vertex
        = NULL
    );
```

Creates an instance of a graph vertex and supplies it with a name.

Destructor:

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

Do not call this destructor directly. An instance of gvertex is deleted by calling the remove method. This is necessary, because gvertex is use counted. This destructor will throw a sys_error if it is called when its use_count is not equal to zero.

Methods:

```
public: void gvertex::add () const;
```

The C++ classes of gvertex and gedge are use counted in the same way that laws are use counted. That is to say that the are copied by calling the add method and deleted by calling the remove method.

```
public: void gvertex::clear_kind ();
```

Sets the user-defined kind array for this graph item to NULL. kind is actually a dynamic array. The value argument specifies whether or not this graph edge is of the kind number specified.

```
public: virtual ENTITY* gvertex::get_entity () const;
```

This returns a pointer to the entity that the graph vertex refers to. Initially, this can be a CELL or a FACE.

```
public: int gvertex::get_kind_size () const;
```

Returns the size of the kind array.

```
public: logical gvertex::is_kind (
    int which               // kind to test
    ) const;
```

Determines whether or not this graph vertex is of the specified kind.

```
public: char const* gvertex::name () const;
```

Every graph vertices can be supplied a character string as a name. This method returns its name.

```
public: logical gvertex::operator!= (
    gvertex const& in_vertex// gvertex
    ) const;
```

Determines whether or not the supplied vertex is not equal to this graph vertex.

```
public: logical gvertex::operator== (
    gvertex const& in_vertex// supplied vertex
    ) const;
```

Determines whether or not the supplied vertex is equal to this graph vertex.

```
public: void gvertex::remove ();
```

The C++ classes of gvertex and gedge are use counted in the same way that laws are use counted. That is to say that the are copied by calling the add method and deleted by calling the remove method.

```
public: void gvertex::set_kind (
    int which,              // kind to use
    logical value           // turn on or off
    );
```

Assigns a user-defined kind to this graph edge. kind is actually a dynamic array. The value argument specifies whether or not this graph edge is of the kind number specified.

Internal Use: id, isa, same, type

Related Fncs: None

history_callbacks

Class: History and Roll, Callbacks

Purpose: Provides callbacks for history management.

Derivation: history_callbacks : toolkit_callback : –

SAT Identifier: None

Filename: kern/kernel/kerndata/bulletin/hist_cb.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

None

Destructor:

None

Methods:

```
public: virtual void
    history_callbacks::After_Roll_Bulletin_Board (
    BULLETIN_BOARD*,         // bulletin board
    logical discard          // discard
    );
```

Callback method, called after rolling a bulletin board. If discard is TRUE, the roll is due to error processing, and the BULLETIN_BOARD will be deleted along with all of its BULLETINs.

```
public: virtual void
    history_callbacks::After_Roll_State (
    DELTA_STATE*             // delta state
    );
```

Callback method, called after rolling one state.

```
public: virtual void
    history_callbacks::After_Roll_States ();
```

Callback method, called after rolling all states.

```
public: virtual void
    history_callbacks::Before_Roll_Bulletin_Board (
    BULLETIN_BOARD*,         // bulletin board
    logical discard          // discard
    );
```

Callback method, called before rolling a bulletin board. If discard is TRUE, the roll is due to error processing, and the BULLETIN_BOARD will be deleted along with all of its BULLETINs.

```
public: virtual void
    history_callbacks::Before_Roll_State (
    DELTA_STATE*            // delta state
    );
```

Callback method, called before rolling one state.

```
public: virtual void
    history_callbacks::Before_Roll_States ();
```

Callback method, called before rolling all states.

Related Fncs:

None

history_callbacks_list

Class: History and Roll, Callbacks

Purpose: Provides a list of callbacks for history.

Derivation: history_callbacks_list : toolkit_callback_list : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerndata/bulletin/hist_cb.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

None

Destructor:

None

Methods:

```
public: void history_callbacks_list::add (
    history_callbacks* cb   // callback
    );
```

Adds a callback to the list.

```
public: virtual void history_callbacks_list::
    After_Roll_Bulletin_Board (
    BULLETIN_BOARD*,        // bulletin board
    logical discard         // will be discarded
    );
```

Callback method, called after rolling a bulletin board. If discard is TRUE, the roll is due to error processing, and the BULLETIN_BOARD will be deleted along with all of its BULLETINs.

```
public: virtual void
    history_callbacks_list::After_Roll_State (
    DELTA_STATE*            // delta state
    );
```

Callback method, called after rolling one state.

```
public: virtual void
    history_callbacks_list::After_Roll_States ();
```

Callback method, called after rolling all states.

```
public: void history_callbacks_list::append (
    history_callbacks* cb   // callback
    );
```

Appends a history callback to the callback list.

```
public: virtual void history_callbacks_list::
    Before_Roll_Bulletin_Board (
    BULLETIN_BOARD*,        // bulletin board
    logical discard         // will be discarded
    );
```

Callback method, called before rolling a bulletin board. If discard is TRUE, the roll is due to error processing, and the BULLETIN_BOARD will be deleted along with all of its BULLETINs.

```
public: virtual void
    history_callbacks_list::Before_Roll_State (
    DELTA_STATE*            // delta state
    );
```

Callback method, called before rolling one state.

```
public: virtual void
    history_callbacks_list::Before_Roll_States ();
```

Callback method, called before rolling all states.

Related Fncs:

None

HISTORY_MANAGER

Class: History and Roll, SAT Save and Restore

Purpose: Creates a history state on the specified history stream.

Derivation: HISTORY_MANAGER : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/sg_husk/history/history.hxx

Description: Takes the bulletins in the current delta state and creates a history state on the specified history stream. The current delta is left with nothing in it. Returns the newly created history state. If the current delta state is empty (has no bulletins), NULL is returned. When all is done, the current delta state is (optionally) cleared.

Limitations: None

References: by KERN StreamFinder

Data:

None

Constructor:

None

Destructor:

None

Methods:

```
public: static DELTA_STATE*
    HISTORY_MANAGER::acquireCurrentDelta (
    HISTORY_STREAM*,         // history stream
    logical clearDeltaState  // clear status
        = TRUE
    );
```

Takes the bulletins in the current delta state and creates a history state on the specified history stream. The current delta is left with nothing in it. Returns the newly created history state. If the current delta state is empty (has no bulletins), NULL is returned. When all is done, the current delta state is (optionally) cleared.

```
public: static void HISTORY_MANAGER::changeToState (
    HISTORY_STREAM* pStream, // source stream
    DELTA_STATE* pTarget,    // target
    int& statesChanged       // counter for rolling
                             // states
    );
```

The state knows which stream it is in, so it does not need to be passed.

```
public: static outcome
    HISTORY_MANAGER::checkDeltaForDistribute (
    DELTA_STATE* pState,         // delta state
    StreamFinder* pStreamFinder // finds stream
    );
```

Perform advance checks on a DELTA_STATE to make sure it is OK to distribute it. Thus, problems can be detected before changing any of the data structure.

```
public: static void
    HISTORY_MANAGER::clearCurrentDelta ();
```

Discards all the bulletins in the current delta state and clears it.

```
public: static int HISTORY_MANAGER::count_bulletins (
    DELTA_STATE* pState      // delta state
    );
```

Gets the number of bulletins in the given delta state.

```
public: static void
    HISTORY_MANAGER::debugCurrentDelta ();
```

Dumps bulletin board into current delta.

```
public: static void HISTORY_MANAGER::detach (
    ENTITY*                 // entity
    );
```

Detaches the given entity from any history stream it may be attached to. Strips the entity of the connecting attribute.

```
public: static outcome
    HISTORY_MANAGER::distributeDeltaState (
    DELTA_STATE* pState,        // delta state
    StreamFinder* pStreamFinder,// stream finder
    logical clearDeltaState     // clear ds flag
        = TRUE,
    logical hideStates          // hide delta state
        = FALSE
    );
```

Takes the bulletins in the current delta state and “distributes” them onto history streams based on their “owning entities”. Bulletins that do not belong to any entity that has a history stream attached to it are simple left in the current delta state. When all is done, the current delta state is (optionally) cleared.

```
public: static HISTORY_STREAM*
    HISTORY_MANAGER::getAttachedStream (
    ENTITY*                 // entity
    );
```

Gets the history stream attached to this entity, if any provided here to hide details of the connecting attribute.

```
public: static logical
    HISTORY_MANAGER::isStateEmpty (
    DELTA_STATE* pState     // delta_state
    );
```

Tests for the existence of any BULLETINs in the given state.

```
public: static logical HISTORY_MANAGER::makeRootDS (
    DELTA_STATE* pState      // given pState
    );
```

Makes the given pState the root delta state of the history stream it is a part of. States prior to the given pState are deleted. The effect is that one cannot roll back over the changes in that delta state. It is useful when initializing the system to prevent rolling back over the initialization. The toolkit uses it to prevent rolling back over api_initialize_faceter.

```
public: static logical HISTORY_MANAGER::restore (
    HISTORY_STREAM*&,        // history stream
    ENTITY**                 // entity
    );
```

Restores a history stream that was saved.

```
public: static int HISTORY_MANAGER::rollNStates (
    HISTORY_STREAM* pStream, // history stream
    int nstates              // number of states
    );
```

Rolls a stream a given number of states or to the end of a branch, which ever comes first. Returns the number of states actually rolled.

```
public: static logical HISTORY_MANAGER::save (
    HISTORY_STREAM*,         // history stream list
    ENTITY_LIST&,            // entity list
    logical activeOnly       // active branch only
                             // if TRUE
    );
```

Saves the history stream and associated entities.

```
public: static void HISTORY_MANAGER::setNewBulletin (
    BULLETIN* b,             // bulletin
    ENTITY* n                // entity
    );
```

Sets the given BULLETIN to point to the given ENTITY, and sets its next and previous pointers to NULL.

Related Fncs:

None

HISTORY_STREAM

Class: History and Roll, SAT Save and Restore

Purpose: Implements a method for saving past states.

Derivation: HISTORY_STREAM : ACIS_OBJECT : –

SAT Identifier: “history_stream”

Filename: kern/kernel/kerndata/bulletin/bulletin.hxx

Description: Externally useful functions to control roll back.

Starts a new bulletin board. At the outermost level of checkpointing, removes any failed previous bulletin board and constructs a new one to run sequentially with any previous successful ones. At any other level of checkpointing, if the argument is FALSE, it takes no action, so that the current bulletin board continues in use. If the argument is TRUE, in effect it pushes a new bulletin board on a stack, whence it may subsequently be removed by a matching close_bulletin_board, restoring everything to its previous state.

Limitations: None

References:

KERN	ATTRIB_HISTORY, DELTA_STATE, DELTA_STATE_LIST, ENTITY, ENTITY_LIST
by KERN	ATTRIB_HISTORY, BULLETIN_BOARD, DELTA_STATE, HISTORY_MANAGER, model_context

Data:

`public ATTRIB_HISTORY* attribute;`
Persistent only to make a connection during save and restore.

`public DELTA_STATE *active_ds;`
Pointer to the active delta state.

`public DELTA_STATE *current_ds;`
Pointer to the current delta state.

`public DELTA_STATE *root_ds;`
Pointer to the root delta state.

`public DELTA_STATE_LIST* merged_states;`
Pointer to a list of delta states merged into this list.

```
public ENTITY_LIST* active_check_list;
```
Used in checking the history stream.

```
public STATE_ID current_state;
```
Current state.

```
public STATE_ID next_state;
```
Acts as a state number server giving a new (unused) state number on request. When model is rolled back to earlier state, the current state number is reset to the state number of the earlier state, but subsequent new states are taken from next_state. State numbers increment from 1.

```
public int logging_level;
```
The number of api_begin's minus the number of api_end's made so far. In effect, this is the current API nesting level.

```
public logical link_states;
```
Indicates if there are link states. Used by api_stop_modeler.

```
public unsigned max_states_to_keep;
```
Limit on the number of states to be kept to control the memory used by a stream. Enforced in note_state by pruning. Hidden states are not counted.

Constructor:

```
public: HISTORY_STREAM::HISTORY_STREAM ();
```

C++ allocation constructor requests memory for this object but does not populate it.

Destructor:

```
public: HISTORY_STREAM::~HISTORY_STREAM ();
```

C++ destructor, deleting a HISTORY_STREAM.

Methods:

```
public: void HISTORY_STREAM::add (
    DELTA_STATE*            // delta state
    );
```

Add delta state to history stream.

```
public: void
    HISTORY_STREAM::add_create_bulletins_to_root_ds (
    ENTITY_LIST&  survivors,        // survivors
    logical remove_existing_from_survivors// remove
                                    // or not
    );
```

Adds create bulletins to root DELTA_STATE, for use when pruning or saving and restoring empty histories.

```
public: logical HISTORY_STREAM::assign_tag (
    const ENTITY* ent,      // entity
    tag_id_type id          // tag
    );
```

Assign a tag to an entity.

```
public: void HISTORY_STREAM::attach (
    DELTA_STATE*,           // delta state
    DELTA_STATE*            // delta state
    );
```

Attach two delta states to one another in history stream.

```
public: logical HISTORY_STREAM::can_roll_back ();
```

Simple test to see whether stream can be rolled back.

```
public: logical HISTORY_STREAM::can_roll_forward ();
```

Simple test to see whether stream can be rolled forward.

```
public:outcome
    HISTORY_STREAM::check_tags_validity (); ;
```

Function to verify the stream is correct.

```
public: void HISTORY_STREAM::clear ();
```

Re–initialize to an empty stream with just the root_ds sys_error if logging_level is not equal to zero.

```
public: void HISTORY_STREAM::clear_history_ptrs ();;
```

Clear reference to this history from entities in delta state.

```
public: BULLETIN_BOARD*
    HISTORY_STREAM::current_bb ();
```

Obtains access to the current bulletin board for update functions.

```
public: DELTA_STATE*
    HISTORY_STREAM::current_delta_state ();
```

Returns pointer to current value of the DELTA_STATE.

```
public: void HISTORY_STREAM::debug (
    int id                  // id for delta states
        = 0,
    int ent_level           // bulletin board
        = 0,                // debugging level
    int level               // delta state
        = 1,                // debugging level
    FILE* fp                // debug file pointer
        = debug_file_ptr
    );
```

Prints debugging information about the history stream.

```
public: void HISTORY_STREAM::delete_delta_states ();
```

Removes delta states from the history stream.

```
public: HISTORY_STREAM* HISTORY_STREAM::detach (
    DELTA_STATE*            // delta state
    );
```

Detaches the given bulletin from the given bulletin board.

```
public: logical
    HISTORY_STREAM::distribution_on () const;
```

Returns the distribute_flag value.

```
public: void HISTORY_STREAM::dump (
    int level               // number of levels
        = 0
    );
```

Number of history stream levels to save to output file.

```
public: void HISTORY_STREAM::find_entities (
    enum ENTITY_TYPE,        // type of entity
    ENTITY_LIST& elist       // entity list
    );
```

Finds the entity from the history stream based on type.

```
public: void HISTORY_STREAM::find_entities (
    is_function,             // flag for an if entity
    ENTITY_LIST& elist       // entity list
    );
```

Finds the entity from the history stream based on functionality.

```
public: DELTA_STATE*
    HISTORY_STREAM::get_active_ds ();
```

Retrieves the active DELTA_STATE.

```
public: DELTA_STATE*
    HISTORY_STREAM::get_current_ds ();
```

Gets the current DELTA_STATE.

```
public: STATE_ID
    HISTORY_STREAM::get_current_state ();
```

Gets the current state.

```
public: void HISTORY_STREAM::get_delta_state (
    STATE_ID& cs,            // current state
    STATE_ID& ns,            // next state
    DELTA_STATE*& ds         // delta state
    );
```

Retrieves the delta state matching the state parameters.

```
public: ENTITY* HISTORY_STREAM::get_entity_from_tag (
    tag_id_type tag_no,          // tag
    outcome& result              // result
        = * (outcome*)NULL_REF
    );;
```

Get an ENTITY from the given tag.

```
public: int HISTORY_STREAM::get_logging_level ();
```

The logging level is the number of api_begin calls minus the number of api_end calls, and represents the current nesting of api calls.

```
public: DELTA_STATE* HISTORY_STREAM::get_root_ds ();
```

Retrieves the root DELTA_STATE.

```
public: DELTA_STATE*
    HISTORY_STREAM::get_state_from_id (
    STATE_ID id              // tag
    );
```

Get the delta state from a given tag.

```
public: void HISTORY_STREAM::get_tagged_entities (
    ENTITY_LIST& elist       // entity list
    );;
```

Function to verify the stream is correct.

```
public: void
    HISTORY_STREAM::initialize_delta_states ();
```

Resets the delta states to beginning.

```
public: logical HISTORY_STREAM::in_stream (
    DELTA_STATE* ds          // delta state
    );
```

Determines whether a given state is in this stream.

```
public: void HISTORY_STREAM::list_delta_states (
    DELTA_STATE_LIST& dslist// delta state list
    );
```

Lists the delta states.

```
public: void HISTORY_STREAM::merge (
    HISTORY_STREAM*          // history stream
    );
```

Merge this delta state in this history stream.

```
public: logical HISTORY_STREAM::mixed_streams (
    HISTORY_STREAM*& alternate_hs   // alternate
                                    // stream
    );;
```

Function to verify the stream is correct.

```
public: STATE_ID HISTORY_STREAM::new_state ();
```

Create a new modeler state identifier.

```
public:HISTORY_STREAM*
    HISTORY_STREAM::next_stream () const ;
```

Get the next stream.

```
public:logical
    HISTORY_STREAM::owns_entities () const ;
```

Read the flag for owning entities.

```
public: HISTORY_STREAM*
    HISTORY_STREAM::previous_stream () const;
```

Get the previous stream.

```
public: void HISTORY_STREAM::prune (
    DELTA_STATE* ds          // delta state
    );
```

Snips the graph of DELTA_STATEs just before the given state and deletes the piece that does not include active_ds. Thus one can prune forward branches by passing a state after active_ds. One can prune past history by passing active_ds or one prior to it. It is impossible to prune away active_ds.

```
public: void HISTORY_STREAM::prune_following ();
```

Prune away all states after active_ds. There is no numToSave here because we don't know what branch to chop if there is a branch. This is not believed to be a practical limitation.

```
public: void HISTORY_STREAM::prune_inactive ();
```

The active path runs from the root, to the current state of the model (active_ds). This routine prunes away states not in the active path.

```
public: void HISTORY_STREAM::prune_inactive_branch (
    DELTA_STATE* ds          // delta state
    );
```

The active path runs from the root, to the current state of the model (active_ds). This routine prunes away states not in the active path, limiting the prune to inactive branches.

```
public: void HISTORY_STREAM::prune_previous (
    int numToSave            // number to save
    );
```

Prune away the earlier parts of the stream, saving active_ds and numToSave earlier states. This can be used to control the memory required for history by limiting the number of states to keep.

```
public: void HISTORY_STREAM::remove (
    DELTA_STATE*             // delta state
    );
```

Remove a bulletin board from this delta state.

```
public: ENTITY*
    HISTORY_STREAM::remove_tag_reference (
    tag_id_type tag_no       // entity id
    );;
```

Remove an ENTITY from the TAG array.

```
public: void HISTORY_STREAM::reset_state (
    STATE_ID n               // state to reset to
    );
```

Resets the modeler state to the given state.

```
public: logical HISTORY_STREAM::restore ();
```

Restores history stream.

read_int	Current State
read_int	Next State
read_int	Maximum states to keep
if(restore_version_number >= ENTITY_TAGS_VERSION)	
read_int	new next tag
read_pointer	Pointer to record in SAT file with the current DELTA_STATE
read_pointer	Pointer to record in SAT file with the active DELTA_STATE
read_pointer	Pointer to record in SAT file with the root DELTA_STATE
read_pointer	Pointer to record in SAT file with ATTRIB_HISTORY
read_data	Intended for unknown data

```
public: tag_id_type
    HISTORY_STREAM::restore_tag_reference (
    const ENTITY* ent       // entity
    );
```

Add an ENTITY to the TAG array.

```
public: void HISTORY_STREAM::roll_links (
    DELTA_STATE*            // delta state
    );
```

This function manages the links between delta states during roll. For internal use only.

```
public: logical HISTORY_STREAM::save (
    ENTITY_LIST& elist,         // entity list
    HISTORY_STREAM_LIST& hslist,// history stream
                                // list
    DELTA_STATE_LIST& dslist    // delta state list
    );
```

Saves the id level, next state, maximum states to keep, pointers to the current delta state, active delta state, and root delta state, and attributes.

```
public: logical HISTORY_STREAM::set_current_state (
    STATE_ID cs,            // current state
    DELTA_STATE* ds         // delta state
    );
```

Sets the current state to the given ID in the DELTA_STATE.

```
public: logical HISTORY_STREAM::set_delta_state (
    STATE_ID cs,            // current state
    STATE_ID ns,            // next state
    DELTA_STATE* ds         // delta state
    );
```

Sets the current delta state to the given state.

```
public: void HISTORY_STREAM::set_distribute_flag (
    logical o               // distribution flag
    );;
```

Set the flag for distribution.

```
public: void HISTORY_STREAM::set_max_states_to_keep (
    int                     // number to set
    );
```

Establishes the maximum number of states to keep in history stream. Additional states above this maximum are pruned.

```
public: void HISTORY_STREAM::set_owners ();
```

Resets the owning stream after moving states between streams.

```
public: void HISTORY_STREAM::set_owns_entities (
    logical o               // owning flag
    );
```

Set the flag for owning entities.

```
public: logical HISTORY_STREAM::set_state_linking (
    logical sl              // state link flag
    );
```

Sets the state link.

```
public: int HISTORY_STREAM::size (
    logical include_backups // backup entities
        = TRUE              // counted if TRUE
     ) const;
```

Returns the amount of space taken by this history stream. This includes all the history stream structure and optionally the backup entities, but not the active entities.

```
public: tag_id_type HISTORY_STREAM::tag (
    const ENTITY* ent,      // entity
    logical check           // perform checks
        = TRUE,             // or not
    tag_id_type required_id // required flag
        = -1
    );
```

Return the tag on an ENTITY in the HISTORY_STREAM.

Internal Use: fix_pointers, full_size

Related Fncs: abort_bb, change_state, clear_rollback_ptrs, close_bulletin_board, current_bb, current_delta_state, debug_delta_state, delete_all_delta_states, delete_ds_branch, get_default_stream, initialize_delta_states, open_bulletin_board, release_bb, set_default_stream

HISTORY_STREAM_LIST

Class: History and Roll

Purpose: Stores a list of history streams.

Derivation: HISTORY_STREAM_LIST : –

SAT Identifier: None

Filename: kern/kernel/kerndata/bulletin/bulletin.hxx

Description: Refer to Purpose.

Limitations: NT, UNIX platforms only.

References: None

Data:

None

Constructor:

```
public: HISTORY_STREAM_LIST::HISTORY_STREAM_LIST ();
```

C++ allocation constructor requests memory for this object but does not populate it.

Destructor:

```
public: HISTORY_STREAM_LIST::~HISTORY_STREAM_LIST ();
```

This should not be called directly. Use remove instead. Laws may not be be deleted because other classes may point to them. To delete a copy of a law call the member function remove which decrements the use_count field and calls the destructor if use_count falls to zero.

Methods:

```
public: int HISTORY_STREAM_LIST::add (
    HISTORY_STREAM* e       // history stream
                            // to add
    );
```

Adds a HISTORY_STREAM to the HISTORY_STREAM_LIST.

```
public: void HISTORY_STREAM_LIST::clear ();
```

Removes all items from the HISTORY_STREAM_LIST.

```
public: int HISTORY_STREAM_LIST::count () const;
```

Returns the items in the HISTORY_STREAM_LIST.

```
public: void HISTORY_STREAM_LIST::init () const;
```

Initializes the HISTORY_STREAM_LIST.

```
public: int
    HISTORY_STREAM_LIST::iteration_count () const;
```

Returns the iteration count for the HISTORY_STREAM_LIST.

```
public: HISTORY_STREAM*
    HISTORY_STREAM_LIST::next () const;
```

Returns the next HISTORY_STREAM in the HISTORY_STREAM_LIST.

```
public: HISTORY_STREAM*
    HISTORY_STREAM_LIST::operator[] (
    int i                    // index of desired
                             // history stream
    ) const;
```

Returns the HISTORY_STREAM specified by the index number in the HISTORY_STREAM_LIST.

```
public: int HISTORY_STREAM_LIST::remove (
    HISTORY_STREAM const* ce// history stream
                                // to remove
    );
```

Removes the given HISTORY_STREAM from the HISTORY_STREAM_LIST.

Internal Use: lookup

Related Fncs: None

INTCURVE

Class: Model Geometry, SAT Save and Restore

Purpose: Defines a parametric curve as an object in the model.

Derivation: INTCURVE : CURVE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “intcurve”

Filename: kern/kernel/kerndata/geom/intcurve.hxx

Description: INTCURVE is a model geometry class that contains a pointer to a (lowercase) intcurve, the corresponding construction geometry class. In general, a model geometry class is derived from ENTITY and is used to define a permanent model object. It provides model management functionality, in addition to the geometry definition.

INTCURVE is one of several classes derived from CURVE to define a specific type of curve. The intcurve record consists of a pointer to an int_cur and a logical denoting its sense.

An INTCURVE is the general representation of any curve that is not defined by an explicit equation, but by reference to other geometric entities. This includes the intersection between two surfaces, the projection of a curve onto a surface, an exact spline curve, or any other general curve.

A use count allows multiple references to an INTCURVE. The construction of a new INTCURVE initializes the use count to 0. Methods are provided to increment and decrement the use count, and after the use count returns to 0, the entity is deleted.

Limitations: None

References: KERN intcurve

Data:

None

Constructor:

```
public: INTCURVE::INTCURVE ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: INTCURVE::INTCURVE (
    intcurve const&         // intersection curve
    );
```

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, because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void INTCURVE::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 INTCURVE::~INTCURVE ();
```

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 INTCURVE(...) then later x–>lose.)

Methods:

```
protected: virtual logical
    INTCURVE::bulletin_no_change_vf (
    ENTITY const* other,        // other entity
    logical identical_comparator// comparator
    ) const;
```

Virtual function for comparing subclass data – called by bulletin_no_change. For the identical_comparator argument to be TRUE requires an exact match when comparing doubles and returns the result of memcmp as a default (for non–overridden subclasses). FALSE indicates tolerant compares and returns FALSE as a default.

```
public: virtual void INTCURVE::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: curve const& INTCURVE::equation () const;
```

Returns the curve's equation, for reading only.

```
public: curve& INTCURVE::equation_for_update ();
```

Returns the curve's equation. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

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

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

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

Returns TRUE if this can be deep copied.

```
public: SPAbox INTCURVE::make_box (
    APOINT*,                // first point
    APOINT*,                // second point
    SPAtransf const*,       // transform
    double                  // tolerance
        = 0.0
    ) const;
```

Makes a SPAbox enclosing a segment of the INTCURVE between two points, and transforms it.

```
public: void INTCURVE::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms an INTCURVE. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void INTCURVE::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.

```
public: void INTCURVE::set_def (
    intcurve const&         // definition curve
    );
```

Sets the INTCURVE's definition curve to the given intcurve. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: curve* INTCURVE::trans_curve (
    SPAtransf const&            // transform
        = * (SPAtransf* ) NULL_REF,
    logical                     // reversed flag
        = FALSE
    ) const;
```

Transforms the curve's equation. If the logical is TRUE, the curve is reversed.

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

Returns the string "intcurve".

Internal Use: full_size

Related Fncs:

is_INTCURVE

intcurve

Class: Construction Geometry, SAT Save and Restore

Purpose: An interpolated curve type.

Derivation: intcurve : curve : ACIS_OBJECT : –

SAT Identifier: "intcurve"

Filename: kern/kernel/kerngeom/curve/intdef.hxx

Description: An intcurve is the general representation of any curve that is not defined by an explicit equation, but by reference to other geometric entities. This includes the intersection between two surfaces, the projection of a curve onto a surface, an exact spline curve, or any other general curve.

The intcurve class represents parametric object-space curves that map an interval of the real line into a 3D real vector space (object-space). This mapping is continuous, and one-to-one except possibly at the ends of the interval whose images may coincide. It is differentiable twice, and the direction of the first derivative with respect to the parameter must be continuous. This direction is the positive sense of the curve.

If the two ends of the curve are different in object space, the curve is open. If they are the same, it is closed. If the curve joins itself smoothly, the curve is periodic, and its period is the length of the interval that it is primarily defined. A periodic curve is defined for all parameter values by adding a multiple of the period to the parameter value so that the result is within the definition interval, and evaluating the curve at that resultant parameter. The point at the ends of the primary interval is known as the seam.

The intcurve class provides an abstraction of the concept of a parametric representation of an interpolated curve. This interpolated curve can be either an “exact” curve or an “approximate” curve that is a fit to a true curve within some fit tolerance.

The intcurve contains a “reversed” bit together with a pointer to another structure, an int_cur or something derived from it that contains the bulk of the information about the curve.

Providing this indirection serves two purposes. First, when an intcurve is duplicated, the copy simply points to the same int_cur, avoiding copying the bulk of the data. The system maintains a use count in each int_cur, that allows automatic duplication if a shared int_cur is to be modified, and deletes any int_cur no longer accessible.

Second, the int_cur contains virtual functions. These virtual functions perform all the operations defined for intcurves that depend on the method of definition of the true curve, so new curve types can be defined by declaring and implementing derived classes. The intcurve and everything using it require no changes to make use of the new definition.

The base class int_cur contains the following information for defining the curve:

- A use count indicating the number of times the int_cur is referred.
- A pointer to a bs3_curve, that represents a spline approximation to the true curve.
- A fitting tolerance representing the precision of the spline approximation to the true curve. This is 0.0 for an exact fit and greater than 0.0 for an approximation.
- Pointers to two surfaces containing the true curve. A class derived from int_cur can use them in different ways but the true curve must lie on each of the surfaces. Either or both surface can be NULL.
- A pointer to two 2D parametric space curves, one on each of the non-NULL, spline surfaces defined above, that represent the 3D spline approximation.

Classes derived from int_cur can contain additional information and record the creation method of the true spline curve.

This file defines the class intcurve, the base class int_cur, which implements curves of intersection between two surfaces (as well as exact spline curves), and an auxiliary class restore_ic_def, which is used to declare int_cur and any derived class to the "restore" system, to allow the correct derived class to be restored from backing store.

Limitations: None

References:

KERN	discontinuity_info, int_cur
by KERN	INTCURVE, imp_par_cur, int_cur, int_int_cur

Data:

None

Constructor:

```
public: intcurve::intcurve ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: intcurve::intcurve (
    bs3_curve,                    // bs3_curve
    double,                       // tolerance
    surface const&,               // first surface
    surface const&,               // second surface
    bs2_curve                     // first bs2_curve
        = NULL,
    bs2_curve                     // second bs2_curve
        = NULL,
    const SPAtransf&              // interval
        = * (SPAtransf*) NULL_REF,
    logical parametric_is_primary// parametric is
        = FALSE                   // primary
    );
```

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

Usually, this method is used for constructing exact spline curves.

```
public: intcurve::intcurve (
    curve_interp&,           // interpolated curve
    SPAbox const&            // bounding box
        = * (SPAbox* ) NULL_REF
    );
```

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

The curve_interp object contains all the data and methods needed to do the interpolation. In addition, a box is supplied within which the fit tolerance must be met.

```
public: intcurve::intcurve (
    intcurve const&          // intcurve
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: intcurve::intcurve (
    int_cur*                 // int_cur
    );
```

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

This creates an intcurve object when the underlying int_cur has been constructed. The int_cur is a derived type which does not need a curve fit or uses a non-standard one.

Destructor:

```
public: intcurve::~intcurve ();
```

C++ destructor, deleting an intcurve. Deletes an intcurve by manipulating the use count of the underlying int_cur structure.

Methods:

```
public: virtual int intcurve::accurate_derivs (
    SPAinterval const&          // interval
        = * (SPAinterval*) NULL_REF
    ) const;
```

Returns the number of derivatives that evaluate finds accurately and directly, rather than by finite differencing, over the given portion of the curve. If there is no limit to the number of accurate derivatives, this method returns the value ALL_CURVE_DERIVATIVES.

```
public: virtual const double*
    intcurve::all_discontinuities (
    int& n_discont,          // number of
                             // discontinuities
    int order                // order
    );
```

Return in a read-only array the number and parameter values of discontinuities of the curve, up to the given order (maximum three).

```
public: virtual double
    intcurve::approx_error () const;
```

Returns a distance value, which represents the greatest discrepancy between positions calculated by calls to eval or eval_position with the approximate results OK logical set by turns to TRUE and FALSE.

```
public: SPAbox intcurve::bound (
    double start,               // first position
    double end,                 // second position
    SPAtransf const& t          // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box enclosing the two given points on the undefined curve line.

```
public: virtual SPAbox intcurve::bound (
    SPAbox const&,              // bounding box
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box surrounding that portion of the curve within the given box.

```
public: virtual SPAbox intcurve::bound (
    SPAinterval const&,         // interval
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box surrounding that portion of the curve within the given parameter interval.

```
public: virtual SPAbox intcurve::bound (
    SPAposition const&,         // first position
    SPAposition const&,         // second position
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box enclosing the two given points on the undefined curve line.

```
public: int intcurve::bs1_hull_angles_ok () const;
```

Returns 1 if the bs1_curve hull turning angles are known to be acceptable, 0 if they are not acceptable, and –1 if unknown.

```
public: int
    intcurve::bs1_hull_self_intersects () const;
```

Returns 1 if the bs1_curve hull is known to self intersect, 0 if it does not, and –1 if unknown.

```
public: int intcurve::bs1_knots_on_curve () const;
```

Returns 1 if the bs1_curve knots are known to lie on the curve, 0 if they do not, and –1 if unknown.

```
public: int intcurve::bs2_hull_angles_ok () const;
```

Returns 1 if the bs2_curve hull turning angles are known to be acceptable, 0 if they are not acceptable, and –1 if unknown.

```
public: int
    intcurve::bs2_hull_self_intersects () const;
```

Returns 1 if the bs2_curve hull is known to self intersect, 0 if it does not, and –1 if unknown.

```
public: int intcurve::bs2_knots_on_curve () const;
```

Returns 1 if the bs2_curve knots are known to lie on the curve, 0 if they do not, and –1 if unknown.

```
public: virtual void intcurve::change_event ();
```

Notifies the derived type that the curve has been changed (e.g. the subset_range has changed) so that it can update itself.

```
public: virtual check_status_list* intcurve::check (
    const check_fix& input          // flags for
        = * (const check_fix*)      // allowed
        NULL_REF,                   // fixes
    check_fix& result               // fixes applied
        = * (check_fix*) NULL_REF,
    const check_status_list*        // checks to be
        = (const check_status_list*)// made, default
        NULL_REF                    // is none
    );
```

Check for any data errors in the curve, and correct the errors if possible. The various arguments provide control over which checks are made, which fixes can be applied and which fixes were actually applied. The function returns a list of errors that remain in the curve on exit.

The default for the set of flags which say which fixes are allowable is none (nothing is fixed). If the list of checks to be made is null, then every possible check will be made. Otherwise, the function will only check for things in the list. The return value for the function will then be a subset of this list.

```
public: virtual logical intcurve::closed () const;
```

Indicates if a curve is closed. A closed curve joins itself (smoothly or not) at the ends of its principal parameter range. This method always returns TRUE if periodic returns TRUE.

```
public: virtual void intcurve::closest_point (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot position
    SPAparameter const& param_guess // input guess
        = * (SPAparameter* )NULL_REF,// value of
                                    // param
    SPAparameter& param_actual      // actual value
        = * (SPAparameter* )NULL_REF// of param
    ) const;
```

Finds the closest point on the curve (the foot) to the given point, and optionally its parameter value. If an input parameter value is supplied (as the first parameter argument), the foot found is only a local solution nearest to the supplied parameter position. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
public: bs3_curve intcurve::cur (
    double tol              // tolerance
        = –1.0
    ) const;
```

Returns the underlying bs3_curve; otherwise, it returns NULL if there is no int_cur.

```
public: logical intcurve::cur_present () const;
```

Returns TRUE if the nth parameter–space curve is defined (i.e. pcur() would return a non–NULL pcurve pointer), FALSE otherwise.

```
public: virtual void intcurve::debug (
    char const*,            // title line
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Outputs a title line and the details of the intcurve for inspection to standard output or to the specified file.

```
public: virtual curve* intcurve::deep_copy (
    pointer_map* pm         // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
public: virtual const double*
    intcurve::discontinuities (
    int& n_discont,         // # discontinuities
    int order               // curve order
    ) const;
```

Returns the number and parameter values of discontinuities of the curve in a read–only array of the given order (maximum three).

```
public: virtual int intcurve::discontinuous_at (
    double t                // parameter value
    ) const;
```

Determines whether a particular parameter value is a discontinuity.

```
public: virtual curve_boundcyl
    intcurve::enclosing_cylinder (
    const SPAinterval&          // interval
        = * (SPAinterval*) NULL_REF
    ) const;
```

Returns a cylinder that encloses the portion of the curve bounded by the interval.

```
public: virtual void intcurve::eval (
    double,                       // parameter value
    SPAposition&,                 // position
    SPAvector&                    // first derivative
        = * (SPAvector* ) NULL_REF,
    SPAvector&                    // second derivative
        = * (SPAvector* ) NULL_REF,
    logical                       // perform a repeat
        = FALSE,                  // evaluation on a
                                  // curve whose
                                  // underlying
                                  // geometry data has
                                  // not changed?
    logical                       // approx results OK?
        = FALSE
    ) const;
```

Evaluates the curve, giving the position and the first and second derivatives.

```
public: virtual int intcurve::evaluate (
    double,                       // parameter value
    SPAposition&,                 // point
    SPAvector**                   // first derivative
        = NULL,
    int                           // second derivative
        = 0,
    evaluate_curve_side           // which side of
                                  // discontinuity to
        = evaluate_curve_unknown// evaluate
    ) const;
```

Evaluates the position and the first and second derivatives at given parameter value.

```
public: virtual int intcurve::evaluate_iter (
    double,                         // parameter
    curve_evaldata*,                // data supplying
                                    // initial values,
                                    // and set to reflect
                                    // the results of
                                    // this evaluation
    SPAposition&,                   // point on curve at
                                    // given parameter
    SPAvector**                     // array of pointers
        = NULL,                     // to vectors, of
                                    // size nd. Any of
                                    // the pointers may
                                    // be null, in which
                                    // case the
                                    // corresponding
                                    // derivative will
                                    // not be returned
    int                             // number of
        = 0,                        // derivatives
                                    // required (nd)
    evaluate_curve_side             // evaluation
                                    // location – above,
                                    // below or don't
        = evaluate_curve_unknown    // care
    ) const;
```

The evaluate_iter function is just like evaluate, but is supplied with a data object which contains results from a previous close evaluation, for use as initial values for any iteration involved.

```
public: virtual SPAvector intcurve::eval_curvature (
    double,                     // parameter value
    logical                     // perform a repeat
        = FALSE,                // evaluation on a
                                // curve whose underlying
                                // geometry data has not
                                // changed?
    logical                     // approx results OK?
        = FALSE
    ) const;
```

Finds the curvature at a point on the curve.

```
public: virtual SPAvector intcurve::eval_deriv (
    double,                 // parameter value
    logical                 // perform a repeat
        = FALSE,            // evaluation on a
                            // curve whose underlying
                            // geometry data has not
                            // changed?
    logical                 // approx results OK?
        = FALSE
    ) const;
```

Finds the parametric derivative, magnitude, and direction, at a point on the curve.

```
public: virtual SPAunit_vector
    intcurve::eval_direction (
    double,                 // parameter on the curve
    logical                 // determine tangency
        = FALSE,
    logical                 // approx results OK?
        = FALSE
    ) const;
```

Find the tangent direction at the given parameter value on the curve.

```
public: virtual SPAposition intcurve::eval_position (
    double,                 // parameter value
    logical                 // perform a repeat
        = FALSE,            // evaluation on a
                            // curve whose underlying
                            // geometry data has not
                            // changed?
    logical                 // approx results OK?
        = FALSE
    ) const;
```

Finds the position on a curve at the given parameter value.

```
public: virtual curve_extremum*
    intcurve::find_extrema (
    SPAunit_vector const&   // direction
    ) const;
```

Finds the extrema of an intersection curve in a given direction, ignoring its ends unless it is closed.

```
public: double intcurve::fitol () const;
```

Returns the fit tolerance; otherwise, it returns 0 if there is no precise int_cur.

```
public: virtual const discontinuity_info&
    intcurve::get_disc_info() const;
```

Returns read-only access to a discontinuity_info object, if there is one. The default version of the function returns NULL.

```
public: int_cur const&
    intcurve::get_int_cur () const;
```

Returns the fit tolerance, but should not be used unless absolutely necessary.

```
public: int intcurve::hull1_enclosure () const;
```

Returns 1 if the bs1_curve hull is known to enclose the curve, 0 if it does not, and –1 if unknown.

```
public: int intcurve::hull2_enclosure () const;
```

Returns 1 if the bs2_curve hull is known to enclose the curve, 0 if it does not, and –1 if unknown.

```
public: logical intcurve::join (
    intcurve& second,       // intcurve
    int order               // discontinuity order
        = -1
    );
```

Join two pieces of intcurve together, adding a discontinuity of the given order at the join (if order is not supplied, or is nonpositive, then it is calculated).

The curves must be suitable for joining. The following conditions must be satisfied:

1. The start of the second curve must match the end of ‘this’.
2. The underlying int_curs must have the same type.
3. The first curve must be unlimited above, and the second curve unlimited below.

If any of these conditions are violated, the function returns FALSE and the curves are unchanged.

The other ends may also match, in which case they will also be joined and the final curve will be periodic.

```
public: law* intcurve::law_form ();
```

Returns the law form for an intcurve.

```
public: virtual double intcurve::length (
    double,                  // first parameter
    double                   // second parameter
    ) const;
```

Returns the algebraic distance along the curve between the given parameters. The value is positive if the parameter values are given in increasing order and negative if they are in decreasing order. The result is undefined if either parameter value is outside the parameter range of a bounded curve. For a periodic curve, the parameters are not reduced to the principal range, and so the portion of the curve evaluated may include several complete circuits. This method is always a monotonically increasing function of its second argument if the first is held constant, and a decreasing function of its first argument if the second is held constant.

```
public: virtual double intcurve::length_param (
    double,                  // datum parameter
    double                   // arc length
    ) const;
```

Returns the parameter value of the point on the curve at the given algebraic arc length from that defined by the datum parameter. The result is not defined for a bounded nonperiodic curve if the datum parameter is outside the parameter range, or if the length is outside the range bounded by the values for the ends of the parameter range.

```
public: virtual curve* intcurve::make_copy () const;
```

Virtual function to copy a curve without knowing what its type is.

```
public: virtual curve_evaldata*
    intcurve::make_evaldata () const;
```

Construct a data object to retain evaluation information across calls to evaluate_iter. This is to allow subsidiary calls within an iterative evaluator to start iteration much closer to the required result than is possible just using the curve information itself.

```
public: void intcurve::make_single_ref ();
```

Ensure that the reference supplied points to a singly–used record. Take no action if it is already single, otherwise copy everything.

```
public: virtual curve& intcurve::negate ();
```

Negates an intcurve in place. This is a curve virtual function, which is why it returns a curve& instead of an intcurve&.

```
public: virtual curve& intcurve::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms a curve in place. This is complicated by the effort to maintain sharing when several intcurves sharing the same int_cur are transformed successively with the same transformation. A list of transformed versions of each int_cur is maintained, and this method searches for a match before making a new one.

```
public: intcurve intcurve::operator– () const;
```

Negates the curve.

```
public: intcurve& intcurve::operator= (
    intcurve const&          // intcurve
    );
```

Assignment operator, which copies only the intcurve record and adjusts the use counts of the underlying information.

```
public: virtual logical intcurve::operator== (
    curve const&             // intcurve
    ) const;
```

Tests two curves for equality. This method does not guarantee to say “equal” for effectively-equal curves, but it is guaranteed to say “not equal” if they are indeed not equal. Use the result for optimization, but not where it really matters. The default always says “not equal.”

```
public: virtual double intcurve::param (
    SPAposition const&,              // position
    SPAparameter const&              // parameter
        = * (SPAparameter*) NULL_REF
    ) const;
```

Returns the parameter value for a given point.

```
public: virtual double
    intcurve::param_period () const;
```

Returns the period of a periodic curve; otherwise, it returns 0 if the curve is not periodic.

```
public: virtual SPAinterval intcurve::param_range (
    SPAbox const&            // bounding box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Returns the range of parameter values.

```
public: virtual pcurve* intcurve::pcur (
    int                      // index
    ) const;
```

Returns the parametric curves with respect to the surfaces defining this intcurve. int may be 1 or 2, representing the two surfaces in order, or –1 or –2 meaning the negation of those pcurves.

```
public: bs2_curve intcurve::pcur1 (
    logical force           // force surface return
        = FALSE
    ) const;
```

Returns a curve in parameter space of surface returned by surf1 or surf2 respectively, if the surface is parametric. Returns NULL if the surface (as returned by the functions above) is NULL or not parametric.

```
public: bs2_curve intcurve::pcur2 (
    logical force           // force surface return
        = FALSE
    ) const;
```

Returns a curve in parameter space of surface returned by surf1 or surf2 respectively, if the surface is parametric. Returns NULL if the surface (as returned by the functions above) is NULL or not parametric.

```
public: virtual logical intcurve::pcur_present (
    int                     // nth parameter–space
                            // curve
    ) const;
```

Returns TRUE if the *n*th parameter-space curve is defined (i.e., pcur returns a non-NULL pcurve pointer); otherwise, it returns FALSE.

```
public: virtual logical intcurve::periodic () const;
```

Indicates if the curve is periodic. A periodic curve joins itself smoothly with matching derivatives at the ends of its principal parameter range so that edges may span the seam.

```
public: virtual SPAvector intcurve::point_curvature (
    SPAposition const&,             // point
    SPAparameter const&             // param guess
        = * (SPAparameter*) NULL_REF
    ) const;
```

Finds the curvature at a point on the intcurve.

```
public: virtual SPAunit_vector
    intcurve::point_direction (
    SPAposition const&,             // point
    SPAparameter const&             // param guess
        = * (SPAparameter*) NULL_REF
    ) const;
```

Finds the tangent direction to the intcurve at a given point.

```
public: virtual void intcurve::point_perp (
    SPAposition const&,             // position
    SPAposition&,                   // foot
    SPAunit_vector&,                // tangent
    SPAvector&,                     // curvature
    SPAparameter const&             // guess value
        = * (SPAparameter* ) NULL_REF,
    SPAparameter&                   // actual value
        = * (SPAparameter* ) NULL_REF,
    logical f_weak                  //
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve and the curve tangent direction and curvature at that point and its parameter value. If an input parameter value is supplied as the guess value, the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one at which the curve is nearest to the given point. Any of the return value arguments may be NULL reference, in which case it is simply ignored.

```
public: void intcurve::point_perp (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot
    SPAparameter const& guess       // guess value
        = * (SPAparameter*) NULL_REF,
    SPAparameter& actual            // actual value
        = * (SPAparameter*) NULL_REF,
    logical f_weak                  //
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve and its parameter value. If an input parameter value is supplied as the guess value, the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one at which the curve is nearest to the given point. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
public: void intcurve::point_perp (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot
    SPAunit_vector& foot_dt,        // normal
    SPAparameter const& guess       // guess value
        = * (SPAparameter* ) NULL_REF,
    SPAparameter& actual            // actual value
        = * (SPAparameter* ) NULL_REF,
    logical f_weak                  //
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve and the tangent direction to the curve at that point and its parameter value. If an input parameter value is supplied as the guess value, the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one at which the curve is nearest to the given point. Any of the return value arguments may be NULL reference, in which case it is simply ignored.

```
public: void intcurve::reparam (
    double,                 // start point
    double                  // end point
    );
```

Reparameterizes the splines to start and end at the given values, which are in increasing order.

```
public: void intcurve::restore_data ();
```

Restores the data from a save file. The restore operation switches on a table defined by static instances of the restore_cu_def class. This invokes a simple friend function which constructs an object of the right derived type. Then it calls the appropriate base class member function to do the actual work. The restore_data function for each class can be called in circumstances when it is known what type of surface is to be expected and a surface of that type is on hand to be filled in.

read_logical Curve direction either “forward” or “reversed”.

```
if (restore_version_number < INTCURVE_VERSION)
    // Restore as a surface–surface intersection object. The
    // restore function for int_int_cur handles the possibility
    // that it is in fact exact or a surf_int_cur.
    subtype_object * dispatch_restore_subtype
```

Called with “cur” and “surfintcur”. Restore just the data associated with that type of curve. In earlier versions, there was only one type of int_cur, which covered what is now “exact”, “surf”, and “int”. There was no ID.

```
else
    // Switch to the right restore routine, using the standard
    // system mechanism. Note that the argument is to enable
    // the reader to distinguish old–style types where “exact”
    // was both an int_cur and a spl_sur. They are now “exactcur”
    // and “exactsur”.
    subtype_object * dispatch_restore_subtype
```

Called with “cur”

curve::restore_data Restore the underlying curve. Generic curve data.

```
public: logical intcurve::reversed () const;
```

Returns TRUE if the intcurve is reversed.

```
public: SPAinterval intcurve::safe_range () const;
```

Returns the safe range or an empty interval if there is no int_cur.

```
public: virtual void intcurve::save () const;
```

Saves the curve type or id, then calls save_data.

```
public: void intcurve::save_data () const;
```

Saves the intcurve data to a save file.

```
public: void intcurve::set_bs_hull_angles_ok (
    int pcu_no,             // enclosure
    int hull_angles_ok      // unknown, false,
                            // or true
    );
```

Sets the property in the underlying int_cur, of the bs2_curve hull not turning too sharply. The first argument should be 1 or 2 to indicate the bs2_curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to indicate that the property is unknown, FALSE or TRUE.

```
public: void intcurve::set_bs_hull_self_intersects (
    int pcu_no,             // enclosure
    int hull_self_ints      // unknown, false,
                            // or true
    );
```

Sets the property in the underlying int_cur, of the bs2_curve hull self–intersecting (or not). The first argument should be 1 or 2, to indicate the bs2_curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to indicate that the property is unknown, FALSE or TRUE.

```
public: void intcurve::set_bs_knots_on_curve (
    int pcu_no,             // enclosure
    int knots_on_cu         // unknown, false,
                            // or true
    );
```

Sets the property in the underlying int_cur, of the bs2_curve of whether the knots lie on the intcurve. The first argument should be 1 or 2, to indicate the bs2_curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to indicate that the property is unknown, FALSE or TRUE.

```
public: void intcurve::set_cur (
    bs3_curve,              // bs3 curve data
    double tol              // tolerance
        = -1.0
    );
```

Replaces the underlying bs3_curve approximation. If the supplied tolerance is negative, then it will be left unchanged.

```
public: void intcurve::set_hull_enclosure (
    int pcu_no,             // enclosure
    int encl                // value
    );
```

Sets the curve enclosure in the underlying int_cur. The first argument should be 1 or 2, to indicate the bs2_curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to give the value for the hull enclosure.

```
public: void intcurve::set_periodic ();
```

Marks an intcurve as periodic. This is used after splitting a periodic intcurve to restore the periodic status that the split changed to closed.

```
public: virtual curve* intcurve::split (
    double,                         // param value
    SPAposition const&              // exact position
        = * (SPAposition* ) NULL_REF
    );
```

Divides an intcurve into two pieces at a parameter value. This method creates a new intcurve on the heap, but either one of the intcurves may have a NULL actual curve. The supplied curve is modified to be the latter section, and the initial section is returned as a value.

```
public: curve* intcurve::subset (
    SPAinterval const&      // interval
    ) const;
```

Constructs a new curve, which is a copy of the portion of the given one within the specified parameter bounds.

```
public: surface const& intcurve::surf1 (
    logical force           // force surface to
        = FALSE             // be returned
    ) const;
```

Returns the first surface supporting the curve. By default, surfaces are only returned if the true curve lies on the surface. Surfaces defining the curve but distant from it are not returned. To force the surface to be returned regardless, the logical flag should be set to TRUE.

```
public: surface const& intcurve::surf2 (
    logical force           // force surface to
        = FALSE             // be returned
    ) const;
```

Returns the second surface supporting the curve. By default, surfaces are only returned if the true curve lies on the surface. Surfaces defining the curve but distant from it are not returned. To force the surface to be returned regardless, the logical flag should be set to TRUE.

```
public: curve_tancone intcurve::tangent_cone (
    SPAinterval const& range,   // interval
    logical approx_OK,          // approx results OK?
    SPAtransf const& t          // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a cone bounding the tangent direction of a curve. The cone has its apex at the origin and a given axis direction and (positive) half-angle. If approx_OK is TRUE, then a quick approximation may be found. The approximate result may lie wholly within or wholly outside the guaranteed bound (obtained with a FALSE argument), but it may not cross from inside to outside. Flags in the returned object indicate whether the cone is in fact the best available, and if this result is inside or outside the best cone.

```
public: virtual logical intcurve::test_point_tol (
    SPAposition const&,             // point
    double                          // tolerance
        = 0,
    SPAparameter const&             // start point
        = * (SPAparameter* ) NULL_REF,
    SPAparameter&                   // end point
        = * (SPAparameter* ) NULL_REF
    ) const;
```

Tests a point-on-curve to a given tolerance. Only true points are near to the end points.

```
public: virtual int intcurve::type () const;
```

Returns the type of intcurve.

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

Returns the string “intcurve”.

```
public: virtual logical intcurve::undef () const;
```

Indicates if the curve is defined or undefined.

Internal Use: full_size

Related Fncs:

restore_intcurve, restore_int_int_cur

```
friend: intcurve operator* (
    intcurve const&,         // intcurve
    SPAtransf const&         // transform
    );
```

Returns a copy of the transformed curve.

int_cur

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines interpolated curves.

Derivation: int_cur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: int_cur

Filename: kern/kernel/kerngeom/curve/intdef.hxx

Description: This class defines interpolated curves, which are defined to allow the use of use-counts to avoid copying and to allow derivation to construct curves only approximated by the intcurve.

This class is supported by virtual functions that depend on the true definition of the curve. The virtual functions allow the derived curves to implement the functionality on their own. For curves with an exact bs3_curve, there is no need to implement the functionality because the methods written for the base class are sufficient.

Limitations: None

References:

KERN	discontinuity_info, summary_bs3_curve, surface
by KERN	intcurve, summary_bs3_curve
BASE	SPAinterval

Data:

`protected bs2_curve pcur1_data;`
A parametric-space curve with respect to the give surface. It is non-NULL only if the corresponding surface exists, and it is parametric.

`protected bs2_curve pcur2_data;`
A parametric-space curve with respect to the give surface. It is non-NULL only if the corresponding surface exists, and it is parametric.

`protected bs3_curve cur_data;`
The object-space approximation to the true curve.

`protected closed_forms closure;`
Takes the value OPEN, CLOSED or PERIODIC (or unset if the int_cur is undefined). If an approximating curve is present (int_cur_data) then the closure of the approximating curve will be consistent.

`protected discontinuity_info disc_info;`
Discontinuity information. If the supporting surfaces of the curve has discontinuities, or if the curve has a default (tangent) extension, then it will have discontinuities. These are stored here. Note that this is a copy of the data stored in the corresponding spl_sur, but with values outside the subset range removed. It is necessary to keep a separate copy to provide a read–only array to the data because the curve may be periodic, and the subset may span the periodic joins, resulting in discontinuities which may be outside the periodic range.

`protected double fitol_data;`
The precision to which the spline approximates to the true object-space curve.

`protected int bs1_properties;`
Refer to bs2_curve_properties below.

`protected int bs2_properties;`
Defines the following properties of the bs2_curves:

a. Whether or not all of the bs2_curve knots lie on the int_cur.
b. Whether all of the bs2_curve hull turning angles are not too sharp.
c. Whether the bs2_curve hull is known to self–intersect or not.
d. Whether the bs2_curve hull fully encloses the int_cur.

It has a value consisting of four digits.
The first digit has the following values :

0 if Knots on curve property is unknown
1 if Knots on curve property is FALSE
2 if Knots on curve property is TRUE

The second digit has the following values :

0 if Turning angles ok property is unknown
1 if Turning angles ok property is FALSE
2 if Turning angles ok property is TRUE

The third digit has the following values :

0 if Hull self–intersection property is unknown
1 if Hull self–intersection property is FALSE
2 if Hull self–intersection property is TRUE

The fourth digit has the following values :

0 if Hull enclosure property is unknown
1 if Hull enclosure property is FALSE
2 if Hull enclosure property is TRUE

`protected SPAinterval range;`
The full range of the int_cur, as returned by param_range. If an approximating curve is present (int_cur_data) then range and bs3_curve_range(cur_data) should be identical.

`protected SPAinterval safe_range;`
A sub-range of the curve that avoids any terminators at the ends of the curve is safe for relaxation. Outside this range, but inside the full curve range, the approximating curve is taken to define the curve. Typically when no terminators or surface singularities are present, the safe_range is the full range; if a terminator is present, the safe_range stops just short of it. The base class administers the safe_range; for example it updates it following re-parameterization. It is the responsibility of the derived class to set it initially.

```
protected logical calling_make_approx;
```
Prevents recursion to make_approx( ).

```
protected summary_bs3_curve* summary_data;
```
bs3_curve data in summary form. This field may be set on restore, if the full curve is not available. It may be used to make the actual bs3_curve.

```
protected surface *surf1_data;
```
The first of up to two surfaces defining the true curve. Derived classes may use this in different ways, but in the base class, the true curve lies on any surface specified here.

```
protected surface *surf2_data;
```
The second surface defining the true curve. Derived classes may use this in different ways, but in the base class, the true curve lies on any surface specified here.

Constructor:

```
protected: int_cur::int_cur ();
```

C++ allocation constructor requests memory for this object but does not populate it.

Initializes safe_range to initialize interval infinite fitol_data to 0 and all other data to NULL. This provides flexibility for the constructors for derived curve classes to set the common data members in the most convenient way.

```
protected: int_cur::int_cur (
    bs3_curve,                      // given curve
    double,                         // fit tolerance
    surface const&,                 // first surface
    surface const&,                 // second surface
    bs2_curve,                      // pcurve for
                                    // 1st surface
    bs2_curve,                      // pcurve for
                                    // 2nd surface
    const SPAinterval&              // safe ranges
        = * (SPAinterval*) NULL_REF,
    const discontinuity_info&       // discontinuity
        = * (discontinuity_info*) NULL_REF
    );
```

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

This constructor cannot be called directly to make an int_int_cur; however, the following procedure can be used to make an object of the int_int_cur class type.

1. Make an object of type int_int_interp (refer to the definition of int_int_interp in the curve_interp class description).
2. Call the int_int_interp method, make_int_cur. This method returns an int_cur class object.

```
protected: int_cur::int_cur (
    const int_cur&          // intersection curve
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
protected: int_cur::int_cur (
    SPAinterval,                    // interval
    closed_forms,                   // OPEN, CLOSED,
                                    // PERIODIC,
                                    // or undefined
    surface const&,                 // 1st surface
    surface const&,                 // 2nd surface
    bs2_curve,                      // pcurve for
                                    // 1st surface
    bs2_curve,                      // pcurve for
                                    // 2nd surface
    const SPAinterval&              // safe ranges
        = * (SPAinterval*)NULL_REF,
    const discontinuity_info&       // discontinuity
        = * (discontinuity_info*)NULL_REF
    );
```

A version of the constructor which takes the range and closure instead of the approximating curve. Available for derived class constructors.

Destructor:

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

C++ destructor, deleting an int_cur. Eliminates all the dependent spline curve and surface data. Each derived class must have a destructor if it adds further dependent data.

Methods:

```
protected: virtual int int_cur::accurate_derivs (
    SPAinterval const&          // int_cur interval
        = * (SPAinterval*) NULL_REF
    ) const;
```

Returns the number of derivatives that evaluate can find accurately and directly, rather than by finite differencing, over the given portion of the curve. If there is no limit to the number of accurate derivatives, this method returns the value, ALL_CURVE_DERIVATIVES.

```
protected: virtual void int_cur::append (
    int_cur&                // curves to be joined
    );
```

Concatenates the contents of two curves into one. The curves are guaranteed to be the same base or derived type, and to have contiguous parameter ranges (“this” is the beginning part of the combined curve, the argument gives the end part).

```
protected: virtual SPAbox int_cur::bound (
    SPAinterval const&          // range
        = * (SPAinterval*) NULL_REF
    ) const;
```

Finds an object-space bounding box, for the subset of the curve within the given parameter bounds. The default finds the bound on the spline approximation of the appropriate subset of the curve, expanded by the fit tolerance, so it is suitable for most derived classes.

```
protected: int int_cur::bs1_hull_angles_ok () const;
```

Returns 1 if the bs1_curve hull turning angles are known to be acceptable, 0 if they are not acceptable, and –1 if unknown.

```
protected: int
    int_cur::bs1_hull_self_intersects () const;
```

Returns 1 if the bs1_curve hull is known to self intersect, 0 if it does not, and –1 if unknown.

```
protected: int int_cur::bs1_knots_on_curve () const;
```

Returns 1 if the bs1_curve knots are known to lie on the curve, 0 if they do not, and –1 if unknown.

```
protected: int int_cur::bs2_hull_angles_ok () const;
```

Returns 1 if the bs2_curve hull turning angles are known to be acceptable, 0 if they are not acceptable, and –1 if unknown.

```
protected: int
    int_cur::bs2_hull_self_intersects () const;
```

Returns 1 if the bs2_curve hull is known to self intersect, 0 if it does not, and –1 if unknown.

```
protected: int int_cur::bs2_knots_on_curve () const;
```

Returns 1 if the bs2_curve knots are known to lie on the curve, 0 if they do not, and –1 if unknown.

```
protected: closed_forms
    int_cur::calculate_closure ();
```

Calculates the closure of the curve from geometric tests.

```
public: virtual void
    int_cur::calculate_disc_info ();
```

Calculates the discontinuity information for the int_cur if none had been stored in disc_info. This function is intended to support restore of old versions of int_cur.

```
protected: virtual
    check_status_list* int_cur::check (
    const check_fix& input          // flags for
        = * (const check_fix*)      // allowed
        NULL_REF,                   // fixes
    check_fix& result               // fixes
        = * (check_fix*) NULL_REF,  // applied
    const check_status_list*        // list of
        = (const check_status_list*)// checks to
        NULL_REF                    // be made
    );
```

Check for any data errors in the curve, and correct the errors if possible. The various arguments provide control over which checks are made, which fixes can be applied and which fixes were actually applied. The function returns a list of errors that remain in the curve on exit.

The default for the set of flags which say which fixes are allowable is none (nothing is fixed). If the list of checks to be made is null, then every possible check will be made. Otherwise, the function will only check for things in the list. The return value for the function will then be a subset of this list.

```
public: logical int_cur::closed () const;
```

Indicate whether a curve is closed, that is joins itself (smoothly or not) at the ends of its principal parameter range. This method always returns TRUE if periodic returns TRUE. The default version uses the corresponding function for the approximating spline.

```
protected: virtual void int_cur::closest_point (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot position
    SPAparameter const& param_guess // input guess
        = * (SPAparameter* )NULL_REF,// value of
                                    // param
    SPAparameter& param_actual      // actual value
        = * (SPAparameter* )NULL_REF// of param
    ) const;
```

Finds the closest point on the curve (the foot) to the given point, and optionally its parameter value. If an input parameter value is supplied (as the first parameter argument), the foot found is only a local solution nearest to the supplied parameter position. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
protected: void int_cur::
    closest_point_with_cache (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot position
    SPAparameter const&             // input guess
        = * (SPAparameter* )NULL_REF,// value of
                                    // param
    SPAparameter&                   // actual value
        = * (SPAparameter* )NULL_REF// of param
    ) const;
```

This method, rather than closest_point, should be called by classes derived from int_cur, to get the benefit of caching.

```
protected: virtual subtrans_object*
    int_cur::copy () const = 0;
```

Construct a new object as a copy of an old object. Duplication cannot be done by the constructor.

```
protected: bs3_curve int_cur::cur () const;
```

Returns underlying curve information.

```
protected: virtual void int_cur::debug (
    char const*,            // class–identifying line
    logical,                // option to suppress too
                            // much detail
    FILE*                   // file name
    ) const = 0;
```

Prints out a class-specific identifying line to standard output or to the specified file.

```
protected: void int_cur::debug_data (
    char const*,            // class–identifying line
    logical,                // option to suppress too
                            // much detail
    FILE*                   // file name
    ) const;
```

Prints out the details. The debug_data derived class can call its parent's version first, to put out the common data. If the derived class has no additional data it need not define its own version of debug_data, and it may use its parent's instead. A string argument provides the introduction to each displayed line after the first, and a logical sets brief output (normally removing detailed subsidiary curve and surface definitions).

```
public: virtual int_cur* int_cur::deep_copy (
    pointer_map* pm          // list of items within
        = NULL               // the entity that are
                             // already deep copied
    ) const = 0;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

In a *deep* copy, all the information about the copied item is self-contained in a new memory block. By comparison, a *shallow* copy stores only the first instance of the item in memory, and increments the reference count for each copy.

The pointer_map keeps a list of all pointers in the original object that have already been deep copied. For example, a deep_copy of a complex model results in self contained data, but identical sub-parts within the model are allowed to share a single set of data.

```
protected: void int_cur::delete_summary_data ();
```

Allows derived classes to delete summary_data when it goes out of date.

```
protected: void int_cur::disc_from_surfs ();
```

This function sets discontinuity information in the curve corresponding to discontinuities in the underlying surfaces.

```
protected: virtual curve_boundcyl
    int_cur::enclosing_cylinder (
    const SPAinterval&           // bounding interval
        = * (SPAinterval*) NULL_REF
    ) const;
```

Returns a cylinder which encloses the portion of the curve bounded by the interval.

```
protected: save_approx_level
int_cur::enquire_save_approx_level () const;
```

Gets the default level at which the approximating surface should be be stored.

```
protected: virtual void int_cur::eval (
    double,                     // given parameter
    SPAposition&,               // position returned
    SPAvector&                  // 1st derivative
        = * (SPAvector* ) NULL_REF,
    SPAvector&                  // 2nd derivative
        = * (SPAvector* ) NULL_REF,
    logical                     // approx. results?
        = FALSE
    ) const;
```

Finds the position and the first and second derivative on a curve at a given parameter value. Either eval or evaluate should be implemented for every derived curve class. If any return value is NULL, that value is not computed. If logical is TRUE, approximate results are returned.

```
protected: virtual int int_cur::evaluate (
    double,                     // parameter
    SPAposition&,               // point on curve at
                                // parameter
    SPAvector**                 // array of vectors
        = NULL,
    int                         // # derivatives
        = 0,
    evaluate_curve_side         // eval. location
        = evaluate_curve_unknown
    ) const;
```

Calculate derivatives. Once calculated the derivatives are stored in vectors provided by the user. This method returns the number it was able to calculate; this equals the number requested in all but the most exceptional circumstances. A certain number are evaluated directly and accurately; higher derivatives are automatically calculated by finite differencing; the accuracy of these decreases with the order of the derivative, as the cost increases.

```
protected: virtual int int_cur::evaluate_iter (
    double,                         // parameter
    curve_evaldata*,                // data supplying
                                    // initial values,
                                    // and set to reflect
                                    // the results of
                                    // this evaluation
    SPAposition&,                   // point on curve at
                                    // given parameter
    SPAvector**                     // array of pointers
        = NULL,                     // to vectors, of
                                    // size nd. Any of
                                    // the pointers may
                                    // be null, in which
                                    // case the
                                    // corresponding
                                    // derivative will
                                    // not be returned
    int                             // number of
        = 0,                        // derivatives
                                    // required (nd)
    evaluate_curve_side             // evaluation
                                    // location – above,
                                    // below or don't
        = evaluate_curve_unknown    // care
    ) const;
```

The evaluate_iter function is just like evaluate, but is supplied with a data object which contains results from a previous close evaluation, for use as initial values for any iteration involved.

```
protected: int int_cur::evaluate_iter_with_cache (
    double,                         // parameter
    curve_evaldata*,                // data supplying
                                    // initial values,
                                    // and set to reflect
                                    // the results of
                                    // this evaluation
    SPAposition&,                   // point on curve at
                                    // given parameter
    SPAvector**                     // array of pointers
        = NULL,                     // to vectors, of
                                    // size nd. Any of
                                    // the pointers may
                                    // be null, in which
                                    // case the
                                    // corresponding
                                    // derivative will
                                    // not be returned
    int                             // number of
        = 0,                        // derivatives
                                    // required (nd)
    evaluate_curve_side             // evaluation
                                    // location – above,
                                    // below,don't care
        = evaluate_curve_unknown,
    logical                         // approximations ok
        = FALSE
    ) const;
```

This non-virtual function looks in the cache for position and nd derivatives at the given parameter value. If found it returns them. Otherwise it computes them, puts them in the cache, and returns them. The evaluate_with_cache method instead of evaluate, should be called by classes derived from int_cur in order to get the benefit of caching.

```
public: virtual int int_cur::evaluate_surfs(
    double,                       // Parameter
    SPAposition&,                 // Point on curve at
                                  // parameter
    SPAvector*,                   // Derivatives
    int& nd_cu,                   // Number of curve
                                  // derivs required or
                                  // calculated
    int& nd_sf,                   // Number of surface
                                  // derivs required or
                                  // calculated
    evaluate_curve_side           // eval location
        = evaluate_curve_unknown,
    SPAposition &                 // Point on support
        = *(SPAposition*) NULL_REF,// surface 1
    SPAvector*                    // Derivs of first
        = NULL,                   // support surface
    SPAposition&                  // Point on support
        = *(SPAposition*) NULL_REF,// surface 2
    SPAvector*                    // Derivs of second
        = NULL,                   // support surface
    SPApar_pos&                   // Params on
        = * (SPApar_pos*) NULL_REF,// surface 1
    SPApar_vec*                   // Derivs of params
        = NULL,                   // on surface 1
    SPApar_pos&                   // Params on
        = * (SPApar_pos*) NULL_REF,// surface 2
    SPApar_vec*                   // Derivs of params
         = NULL,                  // on surface 2
    SPApar_pos const&             // opt. guess value
        = * (SPApar_pos* )NULL_REF,// for 1st par_pos
    SPApar_pos const&             // opt. guess value
        = * (SPApar_pos* )NULL_REF// for 2nd par_pos
    ) const;
```

An evaluator that takes surface arguments in addition to curve arguments. As well as returning curve position and derivatives, it returns the derivatives of the surface wrt t (these will often but not always be equal to the curve derivs) and also the derivatives of the surface parameters with respect to t. The array of vectors to return the curve derivatives must be of length at least nd_cu, and the various arrays of vectors to return the surface data can either be null, indicating that this particular derivative is not required, or be of length at least nd_sf.

The caller must supply an array of length nd_cu or NULL to indicate that derivatives are not required for vector.

Unlike the other evaluators, this function OVERWRITES the integer arguments specifying the numbers of derivatives required, with the number actually obtained. The function itself returns information about the surface data that was calculated:

0 => no surface data (e.g. exact_int_cur)
1 => data for first surface only
2 => data for second surface only
3 => data for both surfaces

This is the default implementation of the function, and is inefficient. It should be implemented for each int_cur type.

```
protected: int int_cur::evaluate_with_cache (
    double,                     // parameter
    SPAposition&,               // point on curve at
                                // parameter
    SPAvector**                 // array of vectors
        = NULL,
    int                         // # derivatives
        = 0,
    evaluate_curve_side         // eval. location
        = evaluate_curve_unknown,
    logical                     // approximations OK
        = FALSE
    ) const;
```

This non-virtual function looks in the cache for position and nd derivatives at the given parameter value. If found, it returns them. Otherwise it computes them, puts them in the cache, and returns them. The evaluate_with_cache, instead of evaluate, should be called by classes derived from int_cur in order to get the benefit of caching.

```
protected: virtual SPAvector int_cur::eval_curvature
(
    double,                 // parameter value
    logical                 // approx. results ?
        = FALSE
    ) const;
```

Find the curvature on a curve at a given point. If logical is TRUE, approximate results are returned.

```
protected: virtual SPAvector int_cur::eval_deriv (
    double,                 // parameter value
    logical                 // approx. results ?
        = FALSE
    ) const;
```

Finds the parametric derivative, magnitude, and direction on a given curve at the given parameter value. If logical is TRUE, approximate results are returned.

```
protected: virtual SPAunit_vector
    int_cur::eval_direction (
    double,                 // given direction
    logical                 // approx. results ?
        = FALSE
    ) const;
```

Find the tangent direction at the given parameter value on the curve. Default uses eval_deriv.

```
protected: virtual SPAposition int_cur::eval_position
(
    double,                 // given parameter
    logical                 // approx. results ?
        = FALSE
    ) const;
```

Finds the position on the curve at a given parameter value. If logical is TRUE, approximate results are returned.

```
protected: void int_cur::eval_with_cache (
    double,                     // given parameter
    SPAposition&,               // point found
    SPAvector&                  // first derivative
        = * (SPAvector*) NULL_REF,
    SPAvector&                  // second derivative
        = * (SPAvector*) NULL_REF,
    logical                     // approximations ok
        = FALSE
    ) const;
```

This non-virtual function looks in the cache for position and first and second derivatives at the given parameter value. If found it returns them, otherwise it computes them, puts them in the cache, and returns them.

The eval_with_cache method, rather than eval, should be called by classes derived from int_cur, so as to get the benefit of caching.

```
protected: virtual curve_extremum*
    int_cur::find_extrema (
    SPAunit_vector const&   // unit direction vector
    ) const;
```

Finds the extrema of an intersection curve in a given direction. This method ignores its ends unless it is closed. The default version uses the corresponding function for the approximating spline.

```
protected: double int_cur::fitol () const;
```

Returns fit tolerance data about the curve.

```
protected: int int_cur::hull1_enclosure () const;
```

Returns 1 if the bs1_curve hull is known to enclose the curve, 0 if it does not, and –1 if unknown.

```
protected: int int_cur::hull2_enclosure () const;
```

Returns 1 if the bs2_curve hull is known to enclose the curve, 0 if it does not, and –1 if unknown.

```
public: virtual law* int_cur::law_form ();
```

Returns the law form of an int_cur.

```
protected: virtual double int_cur::length (
    double,                 // start parameter
    double                  // end parameter
    ) const;
```

Returns the algebraic distance along the curve between the given parameters. If the sign is positive, the parameter values are given in increasing order; if the sign is negative,they are given in decreasing order. The result is undefined if either parameter value is outside the parameter range of a bounded curve. For a periodic curve, the parameters are not reduced to the principal range, and so the portion of the curve evaluated may include several complete circuits. This function is always a monotonically increasing function of its second argument if the first is held constant, and a decreasing function of its first argument if the second is held constant. The default version uses the corresponding function for the approximating spline.

```
protected: virtual double int_cur::length_param (
    double,                 // datum parameter
    double                  // arc length
    ) const;
```

Returns the parameter value of the point on the curve at the given algebraic arc length from that defined by the datum parameter, which is the inverse of the length function. The result is not defined for a bounded nonperiodic curve if the datum parameter is outside the parameter range, or if the length is outside the range bounded by the values for the ends of the parameter range. The default version uses the corresponding function for the approximating spline.

```
protected: SPAparameter int_cur::limit_param (
    SPAparameter const& param   // parameter value
    ) const;
```

Shifts the supplied parameter to be within the principle period of a periodic curve. This is used to ensure that bs2_curves are evaluated within their defined ranges.

```
protected: virtual void int_cur::make_approx (
    double fit,             // fit
    const intcurve& ic      // intcurve
        = * (intcurve*) NULL_REF
    ) const;
```

Make or remake the approximating curve. The intcurve argument 'ic' may be NULL but if it is supplied the function may be a little faster. The function stores the approximating curve and the actual fit error that was achieved in the int_cur, overriding the declared const of the method to do this.

```
protected: virtual curve_evaldata*
    int_cur::make_evaldata () const;
```

Construct a data object to retain evaluation information across calls to evaluate_iter. This is to allow subsidiary calls within an iterative evaluator to start iteration much closer to the required result than is possible just using the curve information itself.

```
protected: virtual void int_cur::operator*= (
    SPAtransf const&        // transformation matrix
    );
```

Transforms the bs3_curve and the tolerance. The default transforms the spline approximation and the surfaces, and scales the fit tolerance.

```
public: virtual int_cur& int_cur::operator= (
    int_cur const&          // address for int_cur
    );
```

Copies all the underlying information.

```
protected: virtual logical int_cur::operator== (
    subtype_object const&   // object subtype
    ) const;
```

Tests for equality. This is sufficient for many derived classes, and can be used by most others to check the basic representation. It does not guarantee that all effectively equal surfaces are determined to be equal, but it does guarantee that different surfaces are correctly identified as such. The default version checks the splines and surfaces, and checks that the derived types are the same. This may be sufficient for simple derived types;others may find it useful to call this as part of the operation.

```
protected: virtual double int_cur::param (
    SPAposition const&,             // given point
    SPAparameter const&             // initial
        = * (SPAparameter*) NULL_REF// param guess
    ) const;
```

Returns the parameter values for a given point on the curve. Drops a perpendicular to the spline approximation and returns the parameter value of the foot.

```
public: double int_cur::param_period () const;
```

Finds the parametric period of the interpolated curve, returning exactly 0 if the curve is not periodic. The default version uses the corresponding function for the approximating spline.

```
public: SPAinterval int_cur::param_range (
    SPAbox const&           // bounding box
        = * (SPAbox*)NULL_REF   //
    ) const;
```

Finds the parameter range of the interpolated curve as an interface. The default version uses the corresponding function for the approximating spline.

```
protected: double int_cur::param_with_cache (
    SPAposition const&,         // point found
    SPAparameter const&         // first derivative
        = * (SPAparameter*) NULL_REF
    );
```

This non-virtual function looks in the cache for a given position. If found it returns it; otherwise it computes it, puts it in the cache, and returns it.

The param_with_cache method, rather than paraml, should be called by classes derived from int_cur, so as to get the benefit of caching.

```
protected: virtual pcurve* int_cur::pcur (
    int                      // index for surface
                             // defining intcurve
    ) const;
```

Returns parametric curves with respect to the surfaces defining this intcurve. The argument may be 1 or 2, representing the two surfaces in order. The default uses pcur1 or pcur2, surf1 or surf2, and fitol, and it is suitable for most derived classes.

```
protected: virtual bs2_curve int_cur::pcur1 (
    logical force           // surface force return
        = FALSE
    ) const;
```

Returns curve in parameter space of surface returned by surf1 or surf2 respectively, if the surface is parametric. Returns NULL if the surface (as returned by the functions above) is NULL or not parametric.

```
protected: virtual bs2_curve int_cur::pcur2 (
    logical force           // surface force return
        = FALSE
    ) const;
```

Returns curve in parameter space of surface returned by surf1 or surf2 respectively, if the surface is parametric. Returns NULL if the surface (as returned by the functions above) is NULL or not parametric.

```
public: virtual logical int_cur::pcur_present (
    int                     // parameter-space index
                            // curve
    ) const;
```

Returns TRUE if the *n*th parameter-space curve is defined (i.e., pcur returns a non-NULL pcurve pointer); otherwise, it returns FALSE. The default tests the result of pcur1 or pcur2 as appropriate, and so it suffices for most derived classes. The argument may be 1 or 2 representing the two surfaces in order.

```
public: logical int_cur::periodic () const;
```

Indicates whether the curve is periodic, that is joins itself smoothly at the ends of its principal parameter range, so that edges may span the seam. The default version uses the corresponding function for the approximating spline.

```
protected: virtual SPAvector int_cur::point_curvature
(
    SPAposition const&,             // given point
    SPAparameter const&             // initial
        = * (SPAparameter*) NULL_REF// param guess
    ) const;
```

Finds the curvature on a curve at a given point.

```
protected: virtual SPAunit_vector
    int_cur::point_direction (
    SPAposition const&,             // given point
    SPAparameter const&             // initial
        = * (SPAparameter*) NULL_REF// param guess
    ) const;
```

Finds the tangent direction to a curve at a given point, which is assumed to be on the curve.

```
protected: virtual void int_cur::point_perp (
    SPAposition const&,             // given point
    SPAposition&,                   // point on curve
    SPAunit_vector&,                // tangent
                                    // returned
    SPAvector&,                     // curvature
                                    // returned
    SPAparameter const&             // initial
        = * (SPAparameter*) NULL_REF,// param guess
    SPAparameter&                   // actual param
        = * (SPAparameter*) NULL_REF,// returned
    logical f_weak                  // for future use
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve, tangent to the curve at that point, and its parameter value, and returns the curvature. If an input parameter value is supplied (as the fifth argument), the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one at which the curve is nearest to the given point. Any of the return value arguments may be a NULL reference, in which case they are ignored.

```
protected: void int_cur::point_perp_with_cache(
    SPAposition const&,          // given point
    SPAposition&,                // resulting foot
    SPAunit_vector&,             // direction
    SPAvector&,                  // curvature
    SPAparameter const&          // parameter guess
        = * (SPAparameter*)NULL_REF,
    SPAparameter &               // parameter returned
        = * (SPAparameter*)NULL_REF,
    logical f_weak               // for future use
        = FALSE
    ) const;
```

This non–virtual function looks in the cache for a given position and parameter guess if any. If found it returns the foot, direction, curvature and parameter. Otherwise it uses point_perp to find the result, places them in the cache, and returns them. The point_perp_with_cache method, rather than point_perp, should be called by classes derived from int_cur, so as to get the benefit of caching.

```
protected: virtual void int_cur::reparam (
    double,                  // start parameter
    double                   // end parameter
    );
```

Performs a linear transformation on the parameterization, so that it starts and ends at the given parameter values, which must be in increasing order.

```
protected: void int_cur::restore_common_data ();
```

Restores member function to do the actual work. This will normally be invoked as the first action of the corresponding function for a derived class, to read the common data, as well as doing all the work for the base class.

```
if ( restore_version_number >= APPROX_SUMMARY_VERSION )
    read_enum                       Read the enumeration for
                                    save_approx_level.
if ( level == save_approx_full )
    bs3_curve_restore               Parameter space curve on first
                                    surface
    read_real                       fit tolerance
else if ( level == save_approx_summary )
    summary_bs3_curve::restore      Parameter space curve
    read_real                       fit tolerance
    read_enum                       Read the enumeration for
                                    closed_forms
else
    read_interval                   Range for curve
    read_enum                       Read the enumeration for
                                    closed_forms
restore_surface                     First surface for curve definition
restore_surface                     Second surface for curve definition
bs2_curve_restore                   Parameter space curve on second
                                    surface
bs2_curve_restore                   Parameter space curve on second
                                    surface
if ( restore_version_number >= SAFERANGE_VERSION )
    read_interval                   Safe range for curve evaluation
else
    \\ Set safe range for curve evaluation
if ( restore_version_number >= DISCONTINUITY_VERSION )
    discontinuity_info::restore     Parameter values of discontinuities
```

```
protected: void int_cur::save_as_approx () const;
```

Permits this int_cur to be saved in a sharable format. It may be called for a null object, in which case a recognizable ID is added.

```
protected: void int_cur::save_common_data (
    save_approx_level       // level at which int_cur
                            // is to be stored
    ) const;
```

Save data common to all int_curs.

```
protected: virtual void int_cur::save_data () const;
```

Saves the data associated with the int_cur to the SAT file.

```
protected: void int_cur::set_bs_hull_angles_ok (
    int pcu_no,             // enclosure
    int hull_angles_ok      // value
    );
```

Sets the property of the hull not turning too sharply. The first argument should be 1 or 2 to indicate the curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to indicate that the property is unknown, FALSE or TRUE.

```
protected: void
    int_cur::set_bs_hull_self_intersects (
    int pcu_no,             // enclosure
    int hull_self_ints      // value
    );
```

Sets the property of the hull self intersecting. The first argument should be 1 or 2 to indicate the curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to indicate that the property is unknown, FALSE or TRUE.

```
protected: void int_cur::set_bs_knots_on_curve (
    int pcu_no,             // enclosure
    int knots_on_cu         // value
    );
```

Sets the property of whether the knots lie on the curve. The first argument should be 1 or 2 to indicate the curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to indicate that the property is unknown, FALSE or TRUE.

```
protected: void int_cur::set_cur (
    bs3_curve,              // bs3 curve data
    double tol              // tolerance
        = -1.0
    );
```

Replaces the underlying bs3_curve with another.

```
protected: void int_cur::set_hull_enclosure (
    int pcu_no,             // enclosure
    int encl                // value
    );
```

Sets the curve enclosure in the int_cur. The first argument should be 1 or 2, to indicate the bs2_curve which the enclosure is being set for, and the second integer argument should be –1, 0 or 1, to give the value for the hull enclosure.

```
protected: virtual void int_cur::set_safe_range ();
```

Sets the safe_range, which is used by the base class when it is uncertain how to process the base_range. The default version sets it to a NULL interval. Other implementations of this method are not available to the base class constructor, and they cannot be used when the curve is input from a data stream. Derived classes are responsible for setting the safe_range.

```
protected: virtual void int_cur::shift (
    double                  // shift value
    );
```

Shifts the parameter range of the spline curve by a given value. This is used only to move portions of a periodic curve by integral multiples of the period. The default just shifts the parameterization of the approximating splines.

```
protected: virtual void int_cur::split (
    double,                 // parameter value
    SPAposition const&,     // split point
    int_cur* [ 2 ]          // two pieces of curve
    ) = 0;
```

Divides a curve into two pieces at a given parameter value, adjusting the spline approximations to an exact value at the split point if necessary. If the parameter value is at the beginning, it sets the first piece to NULL and places the original curve in the second slot. If the parameter value is at the end, it places the original curve in the first slot and sets the second to NULL. It is pure virtual to force derived classes to have their own, though many will be able to use split_int_cur (the following method) to do much of the hard work.

```
protected: logical int_cur::split_int_cur (
    double,                 // parameter value
    SPAposition const&,     // split point
    int_cur*,               // input curve
    int_cur* [ 2 ]          // 2 piece of curve
    );
```

Divides a curve into two pieces at a given parameter value, adjusting the spline approximations to an exact value at the split point if necessary, except that a newly-created, but empty, int_cur, is supplied.

The same specification as for split, except that a newly-created, but empty, int_cur (normally in fact a derived object) is supplied to be the second part of the split, if necessary. The method returns TRUE if the second int_cur was used; otherwise, it returns FALSE. This method assumes that the pcurve on any surface is the locus of the foot of the perpendicular from the curve to the surface. It is not called by any intcurve member function, but it is available for use by a derived class, split, to split the spline curves.

```
protected: virtual int_cur* int_cur::subset (
    SPAinterval const&      // parameter interval
    ) const;
```

Constructs a new curve that is a copy of the part of the given one within the given parameter bounds. If this means the whole curve, the original curve, or there is no overlap, this method returns NULL. It is always called with a bounded, positive-length parameter range completely within the curve's defined parameter range.

```
protected: const double*
    int_cur::summary_knots () const;
```

Provides access to the summary_data for derived classes (so that they don't all have to be friends).

```
protected: int int_cur::summary_nknots () const;
```

Provides access to the summary_data for derived classes (so that they don't all have to be friends).

```
protected: virtual surface const* int_cur::surf1 (
    logical force           // surface force return
        = FALSE
    ) const;
```

Returns the corresponding surface pointer only if the true curve lies in that surface.

```
protected: virtual surface const* int_cur::surf2 (
    logical force           // surface force return
        = FALSE
    ) const;
```

Returns the corresponding surface pointer only if the true curve lies in that surface.

```
protected: virtual curve_tancone
    int_cur::tangent_cone (
    SPAinterval const&,     // bounding interval
    logical                 // inside or outside
    ) const;
```

Returns a cone bounding the tangent direction of the curve. The cone has its apex at the origin, and has a given axis direction and (positive) half-angle. If the logical is TRUE, then a quick approximation may be found. The approximate result may lie wholly within or wholly outside the guaranteed bound (obtained with FALSE), but it may not cross from inside to outside. Flags in the returned object indicate whether the cone is in fact the best available, and if not whether this result is inside or outside the best cone. The default finds the cone for the spline approximation to the curve, so will be suitable for most derived classes.

```
protected: virtual logical int_cur::test_point_tol (
    SPAposition const&,             // point
    double                          // tolerance
        = 0,
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter&                   // actual param
        = * (SPAparameter* ) NULL_REF
    ) const;
```

Tests if a point lies on the curve to a given precision. The default version uses the corresponding function for the approximating spline, to a tolerance expanded using the fit tolerance, and then tests the perpendicular to the true curve. It is suitable for most derived classes.

```
protected: virtual char const*
    int_cur::type_name () const = 0;
```

Returns the string “int_cur”.

```
protected: void int_cur::update_data (
    bs3_curve               // bs3_curve
    );
```

Update the range and closure information from a bs3_curve.

Internal Use: deep_copy_elements, full_size

Related Fncs: restore_int_int_cur