Chapter 3.
Attribute Overview

Topic: *Attributes

Attributes are a specialized type of entity used to attach data to entities; any entity may have zero or more attributes. Attributes can carry simple data, pointers to other entities, or links to application-specific variable length data. ACIS uses many “system attributes,” but users may also define application-specific attributes.

The ATTRIB base class in kern provides the data and functionality that all attributes share, for both user-defined attributes and system attributes. The ATTRIB class is derived from the ENTITY class, and performs housekeeping operations to maintain attribute lists attached to model entities. Attributes use the inherent ACIS attribute mechanism, so attributes are saved and restored when the entity to which they are attached is saved to or restored from a file.

In ACIS, the most common types of entities to have attributes attached are topological entities (implemented in the various topology classes), and many of the methods of the ATTRIB class are designed with this in mind. However, attributes can be attached to any type of entity.

Organization attribute classes are always the first level of derivation from the ATTRIB class. (This is similar to the organization class for derivations from ENTITY.) Classes for specific attributes are derived from the organization attribute classes.

User-defined attributes allow extending an ACIS geometric model into a true product model. Developers can derive their own attribute classes from the ACIS ATTRIB class to attach application-specific data to entities. The examples at the end of this topic show how to derive new attribute classes.

Generic attributes allow applications to exchange data not supported by ACIS. These attributes, which are implemented with the ATTRIB_GENERIC class and its derivations (defined in the Generic Attributes Component), may have values or names associated with them. Refer to the *Generic Attributes Component Manual* for information.

The PART class also provides basic attribute support. A generically–named attribute class assigns an attribute with a name (a text string) and a value (number, position, or text) to any entity. There are also functions for selecting entities based on these attributes.

Types of Attributes

Topic: *Attributes

Attributes can be categorized as:

Simple attributes . . . . . Carry simple data only. They may represent properties such as the material of a body or color of a face.

Complex attributes . . . . . Carry pointers to other entities. They may represent properties such as dimensions, constraints, or features. ACIS constructs complex attribute structures to facilitate the extension of ACIS to a specific product modeler.

Bridge attributes . . . . . Are used to link an ENTITY with some application-specific, variable length data.

Instruction attributes . . . . . May be explicitly placed on entities to force certain behavior.

If an entity needs to reference some variable length data, an attribute is created to bridge (connect) from the entity to the data. The variable length data must be placed in an acceptable data structure (such as an array, linked list, or tree) and the attribute must contain a pointer to this variable length data (Figure 3-1).

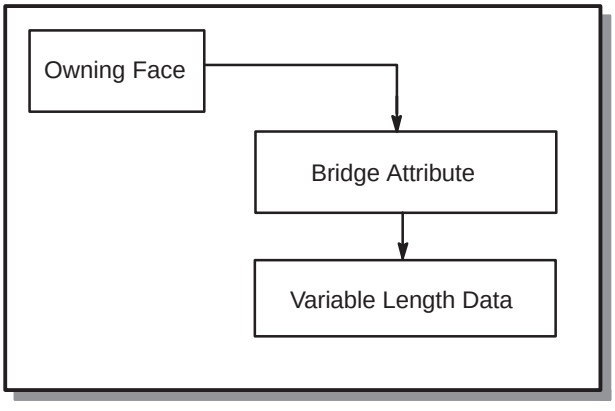

Figure 3-1. Bridge Attribute

A *bridge attribute* can be used to save and restore application–specific information. The functionality for saving and restoring lies in the ENTITY class and is therefore inherited by classes derived from ENTITY. ATTRIB is derived from ENTITY, so all classes derived from ATTRIB inherit methods enabling them to be saved and restored. Application–specific information is probably not contained in objects derived from ENTITY, so some means of bridging between the two sets of data (ACIS–specific and non–ACIS–specific data) is needed. Information that is related to an ENTITY but not contained in the ENTITY can be connected to the ENTITY by putting an attribute on the ENTITY. The attribute can either contain or point to the application–specific data.

A developer might use an attribute as a bridge to application-specific data when converting a subsystem into a separate system. For example, a developer might have originally created a subsystem with its own classes and structures, without intending it to be separate. If it is later decided to convert the subsystem into an independent system, rather than trying to fit the subsystem's classes and structures into the ATTRIB format, the developer can retain the original classes and structures by defining an attribute as a bridge from ACIS to the new system's data.

The methods of the attribute must save and restore and information; however, if the attribute merely points to the information is still responsible for causing the information to be saved and restored. If you put code in the SAVE_DEF block of code for an attribute that saves the application–specific data, this code will be executed when the attribute is saved. Naturally, you must put code in the RESTORE_DEF block of code to restore the same information that was written out in the SAVE_DEF block. If you don't, you will get an error upon restoring the file containing the data.

The content and format of the saved data is up to the user. Obviously, other users will not be able to interpret the application–specific data in the file unless their applications also contain the user–defined attribute class definition.

Instruction attributes can be used to force one of several possible actions. For example, a blend on a face that runs into a sharp edge may be forced to roll on to that edge rather than to form a cap. The roll on instruction would actually be placed on the encountered edge.

Instructions may have positions associated with them to allow different behaviors at different points along the blended edge. In this case, the instruction nearest to the point of intercept applies.

Attribute Methods

Topic: Attributes

An important feature of attributes is the ability of an attribute to control its behavior when its owning entity is split, merged, or translated during a modeling operation. Each attribute class provides methods (member functions) in its implementation file (.cxx) that control its behavior during splitting, merging, or translation.

When an owner entity is split, another entity is created and the application deletes, duplicates, or modifies the attribute. The default action is to do nothing. In Figure 3-2 if the small block is inserted into the larger block and subtracted, face *A* is separated into face *A* and face *A*'. For both faces to keep the attribute, it must be copied and attached to the face *A*'. The split_owner method controls this operation and accepts a pointer to the new entity (in this example, face *A*').

```
virtual void split_owner(ENTITY*);
```

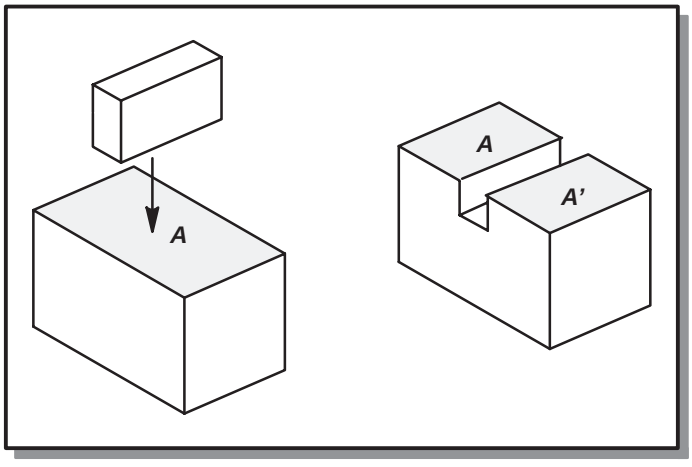

Figure 3-2. Split Entity

When two entities are merged , the attributes may need to be handled differently. In the example in Figure 3-3, if the body owning face *B* is merged with another body that owns face *B*', one face is deleted. The attributes assigned either to face *B* or to face *B*' may be assigned to the resulting face, or the attributes may be combined. For example, if both faces had a centroid attribute, the combination of the attributes for a merge would require a recalculation. The merge_owner method has two arguments: the entity pointer (to face *B*' in this example), and a logical flag that indicates whether the entity containing the attribute is the one being deleted (FALSE in this example).

```
virtual void merge_owner(ENTITY*, logical);
```

Note *In a merge,* merge_owner *is called once for each attribute on* B, *and once for each attribute on* B'.

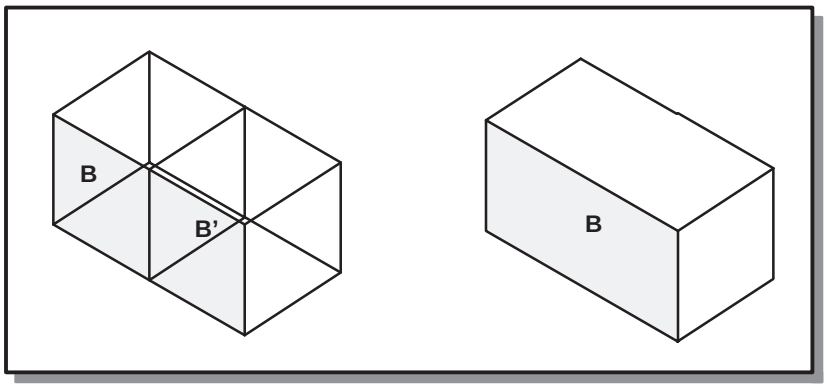

Figure 3-3. Merging Entities

The trans_owner method alerts attributes when the owner entity is transformed. Method trans_owner is called in all modeling operations that apply transformations to entities, such as api_apply_transf and api_change_body_trans, as well as Boolean operations. The trans_owner method has one argument: the transform.

```
virtual void trans_owner(SPAtransf const&);
```

Attribute classes also implement other common methods for such things as copying or replacing the owning entity. Many of these methods are defined using macros defined in the at_macro.hxx file. In addition, methods that are common to entities—and therefore, to attributes—are defined via macros defined in entity_macro.hxx. Other common methods are inherited from ATTRIB.

How ACIS Uses Attributes

Topic: Attributes

ACIS defines many system attributes that are used internally. Each component or area of functionality has an organization attribute class from which specific attribute classes are defined. For example, the ATTRIB_ST and ATTRIB_SYS classes are organization classes used by the Kernel Component and ATTRIB_IHL is the organization class used by the Interactive Hidden Line Component. The ACIS components may also use system attributes and thus define an organization attribute class. For example, ATTRIB_PHL is used by the Precise Hidden Line Component, and ATTRIB_HH is used by the Healing Component. This section describes a few examples of how attributes are used in ACIS.

Healing

The Healing Component uses aggregate attributes attached to the body and individual attributes attached to the individual entities in the body to control the healing process. These attributes are used to store healing options, tolerances, and results. The attributes are implemented using the ACIS attribute mechanism, with C++ classes derived from the ACIS base attribute class, ATTRIB. A user interface can be supplied by the application to provide access to, and possibly control of, the data in these attributes.

The individual attributes are derived from the class ATTRIB_HH_ENT and are named ATTRIB_HH_ENT_<something>. They are attached to entities in the calculate stages and store entity–specific information relating to each phase. For example, in the geometry simplification phase, the individual entity would be a face whose attribute contains a pointer to the simplified surface. In the stitching phase, the individual entity would be an edge whose attribute contains the edge pairing information. The individual attributes can also be used to mark specific entities to be left unchanged by the healing process.

The Healing Component also provides functionality to check the input body to determine if there are any errors before actually performing healing operations. If any errors are found by these analysis routines, they are stored in attributes attached to the bad entities. This information can help the user determine appropriate settings for healing tolerances.

Blending

The ATT_BL_INST class defines the attributes that provide special processing instructions to the blend algorithm. These attributes are attached to the body entities that are actually intercepted by the spring curves. The algorithm reads instructions from the intercepted entities. When more than one instruction attribute is found on a given entity, the closest one applies. When the attribute is found by blending, the instruction is “seen.” Then, at the end of blending, all the “seen” instruction attributes can be deleted.

Attributes are used to set up blends and blend sequences on edges and vertices. A blend is assigned by picking an edge or vertex and describing the blend characteristics, such as its radius. Attributes of class ATTRIB_BLEND are attached to the picked entities. When an entity with a blend attribute is picked to have its blend fixed, the entire set of connected entities with blend attributes is fixed in one operation. This mechanism gives added control and is used to implement various blend construction techniques.

Blend attribute methods test for equality of blends, continuity across a blend (i.e., position–continuous for chamfers, and slope–continuous for rounds), and indicate the size of a blend (in particular, whether it is zero).

Translators

The IGES Translator Component reads IGES files in ASCII format and converts the data into corresponding ACIS entities. Each IGES entity has two components: a directory entry (DE), which is two lines containing the generic description (e.g., a line); and the parameter data (variable) which contain the specifics (e.g., start and end point). Color, label, level, and line font and weight data is stored in attributes attached to the respective geometric entity in ACIS.

Faceting

Incremental faceting saves computation time when only a few faces of a model have undergone some changes. This is accomplished by attaching a mark attribute to each faceted face when passing its facets to the mesh manager.

The API function api_mark_faceted_faces allows the user to select whether or not to attach mark attributes to faceted faces. If marking faceted faces is selected, a subsequent call to either of the functions api_facet_unfaceted_entity or api_facet_unfaceted_entities causes only the unmarked faces (i.e., those that have changed since last faceted) to be faceted.

A mark attribute is lost when its owner face is engaged in any Boolean operation or a transformation operation other than translation or rotation. Therefore, a faceted face that has undergone a change will no longer be marked as faceted.

Attribute Pointers

Topic: *Attributes

The entity to which an attribute is attached “owns” the attribute and contains a pointer to that attribute. If more than one attribute is owned by an entity, the attributes are chained together in a (NULL-terminated) list. Each attribute in the list contains a pointer to the previous attribute in the list, a pointer to the next attribute in the list, and a pointer back to the owning entity (Figure 3-4).

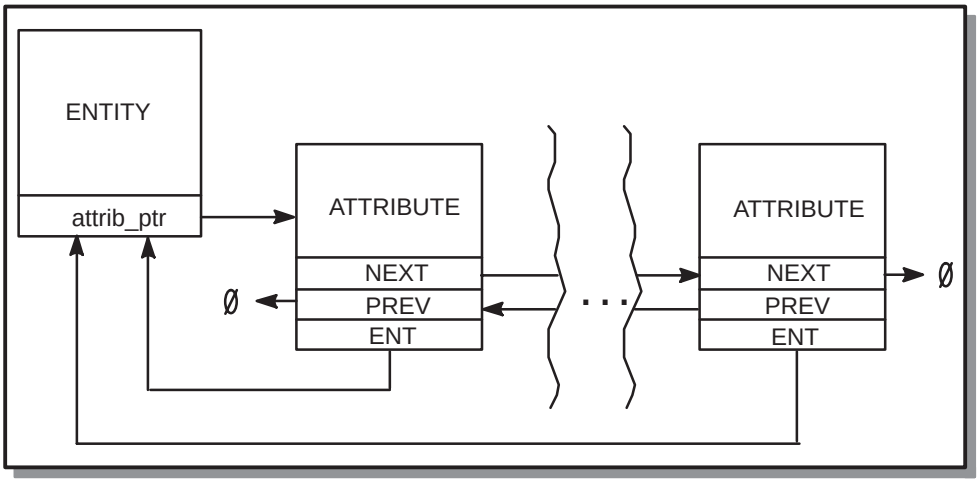

Figure 3-4. Attribute Pointers

Special Characters in Saved Attributes

Topic: *Attributes, *SAT Save and Restore

Attributes are saved and restored when the entity to which they are attached is saved to or restored from a SAT file. Several characters have special meaning when found in "unknown entity" string data in ACIS SAT (.sat) files. Therefore, these characters should not be contained in any user string data written to SAT files. This includes string data in attributes, but applies to string data in any unknown entities.

Note *This only applies to unknown entity string data written to text save files (*.sat*); binary files (*.sab*) are not affected.*

The portion of the ACIS SAT save file restore code that processes unknown entities reserves the following special characters:

{ An opening (left) curly brace begins a subtype definition.

} A closing (right) curly brace terminates a subtype definition.

$ A dollar sign indicates a pointer definition.

A pound sign terminates an entity record.

If these reserved characters are encountered in unknown entity string data when a .sat file is restored (read in), the unknown entity reader will not process the data correctly because these characters are interpreted as special tokens. Refer to Chapter 9, *SAT Save File Format*, in the *Kernel Component Manual* for more information.

Because this processing only applies to unknown entities, applications that “know about” entities containing these characters should be able to process the files. However, if the files are shared with other applications, problems may arise.

Attribute Derivation

Topic: *Attributes, *Extending ACIS

The base C++ class, ATTRIB, provides the data and functionality that all attributes share. The developer creates specific classes of attributes by deriving those classes from ATTRIB and adding extra information and functionality.

Each attribute class generally has a .hxx header file that defines the class, includes necessary header files, and prototypes related functions. It also has a .cxx source file that implements all of the methods of the attribute class (that were not defined as inline methods within the body of the class in the .hxx header file). The cxx source file includes the .hxx class header file and also defines class-specific macros.

Because both the header file and the implementation file are highly stylized, a number of attribute code macros are provided to simplify the derivation process. The macros also help to maintain compatibility with other attributes and entities. Refer to *User-Defined Attribute Macros* and *Predefined Attribute Macros* for more information.

Organization attribute classes are always the first level of derivation from the ACIS class ATTRIB (Figure 3-5). They define no methods or data of their own, and are not instantiated. *Specific attribute classes* are derived from the organization class (i.e., a second-level derivation from ATTRIB), define data and methods, and are instantiated. They may be categorized as simple, complex, or bridge attributes.

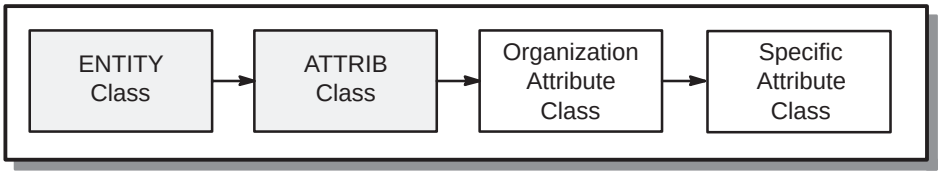

Figure 3-5. Attribute Derivation

Organization Attribute Class

Topic: Attributes

Because the ability to share models between development environments is an important feature of ACIS, it is recommended that each developer first create an organization attribute class that is immediately descended from ATTRIB and has a unique name in the ACIS community. (The organization attribute class is also known as the *master attribute*.) This is accomplished by using a *sentinel*, which is a two- or three-character string that is embedded in the class name and identifies the development organization.

Note *Contact Spatial's customer support department to have a unique sentinel assigned to your ACIS development project.*

The sole purpose of the organization class is to identify the development organization (company) owning that attribute. Organization classes cannot be instantiated.

Application-specific attribute class derivations are distinguishable from classes of the same name created by other developers by their organization name in SAT files. By a similar mechanism, all attributes from a specific developer are easily detected in an attribute list attached to an ENTITY.

Specific Attribute Classes

Topic: Attributes

Specific attribute classes are derived from the organization class. These may be ACIS system attributes or user-defined, application-specific attributes. Specific attribute classes are instantiated.

For application-specific attribute classes to behave similarly to ACIS attributes, each application-specific attribute should include method functions for creation, deletion, transfer to and from disk (save and restore), and migration following the merging or splitting of the owning entities. The roll back and bulletin board mechanisms built into ACIS apply to application-specific attributes.

Attribute Derivation Macros

Topic: *Attributes, *Extending ACIS

This section describes some of the macros used to facilitate the definition of new attribute classes; they are illustrated in examples. These macros are defined in file at_macro.hxx. Because attributes are derived from entities, attribute macros are very similar to the entity macros defined in entity.hxx. For more information, refer to the section on entity derivation macros in the *3D ACIS Application Development Manual*.

User–Defined Attribute Macros

Topic: *Attributes

Developers must define the following macros to use the predefined attribute macros.

THIS() . . . . . The name of this class being defined

PARENT() . . . . . The name of the immediate base class (parent)

THIS_LIB . . . . . The name of the module (library) in which this class is implemented

PARENT_LIB . . . . . The name of the module (library) in which the immediate base class (parent) is implemented

<*class*>_NAME . . . . . The entity's external identifier (<*class*> is the class name)

You must define THIS() and PARENT() with empty parentheses and THIS_LIB and PARENT_LIB without parentheses. A string must be provided for <class>_NAME.

Predefined Attribute Macros

Topic: *Attributes

The predefined attribute macros perform definitions and tasks common to all derived attribute classes. Each attribute class has many methods. Because most of the methods are common to all attribute classes, macros simplify the definition of these methods.

The following macros are used for constructing an application's master attribute. This is the sentinel-level organization attribute class that is constructed entirely by the macros:

MASTER_ATTRIB_DECL . . Declares an master (organization) attribute class, which has no data or methods of its own. This macro includes the macros defined by the ATTRIB_FUNCTIONS macro.

MASTER_ATTRIB_DEFN . . Generates all the code necessary to define a master (organization) attribute class.

The following attribute macro should be invoked in the header file for declaring a derived specific attribute:

ATTRIB_FUNCTIONS . . . . . Declares common utility routines (including ENTITY methods inherited by ATTRIB).

The following attribute macros should be invoked in the implementation file for defining a derived specific attribute. The macros should appear in this order.

ATTRIB_DEF . . . . . This is a heading macro that combines four macros (ATTCOPY_DEF, LOSE_DEF, DTOR_DEF, DEBUG_DEF) in sequence, for the common case of attributes that do not need any special action in the destructor or in duplication. If special handling is needed, the individual macros should be invoked instead of ATTRIB_DEF.

SAVE_DEF . . . . . Completes the debug_ent method and begins the save_common method. Place actions that write out specific attribute information to the save file and insert any pointers into the list immediately following this macro.

RESTORE_DEF . . . . . Completes the save_common method and begins the restore_common method. Place actions that read in specific attribute information from the save file immediately following this macro.

COPY_DEF . . . . . Completes the restore_common method and begins the copy_common method. Place actions that copy data items into the new object, using list to convert any pointers to indices immediately following this macro call.

SCAN_DEF . . . . . Completes the restore_common and begins the copy_scan method, which is used in many places in the system to gather a list of connected entities. Place actions that insert any pointers into the list immediately following this macro.

FIX_POINTER_DEF . . . . . Completes the copy_scan method and begins the fix_common method, which is called during the final stage of restore and copy to map indices into any array of entities to actual pointer values. Place actions that convert any pointers from array indices into array contents immediately following this macro

TERMINATE_DEF . . . . . Terminates the definitions supplied by the *_DEF macros.

In the case of attributes that need special handling in the destructor or in duplication, the following macros should be used (in this order) instead of ATTRIB_DEF:

ATTCOPY_DEF . . . . . Invokes the UTILITY_DEF macro, which provides stock definitions for several required ENTITY functions and static data for the restore function and dynamic virtual functions.

This macro then starts the definition of the fixup_copy method. Immediately following the ATTCOPY_DEF macro, place any special action required to duplicate the attribute for roll back purposes. The attribute itself has been copied member-wise, but if there is a subsidiary structure hanging off a simple pointer (a character string hanging off a char pointer, for example) it requires copying or otherwise processing here.

ATTCOPY_DEF is used like ATTRIB_DEF, but must be followed by LOSE_DEF, DTOR_DEF, and DEBUG_DEF (in that order).

LOSE_DEF . . . . . Defines this attribute's lose method. Place any special actions required to lose any dependent entities immediately following this macro.

DTOR_DEF . . . . . Defines the attribute's destructor, which is called to permanently release all memory back to the operating system. Place any special actions required during attribute destruction immediately following this macro call. For example, a subsidiary data structure may need to be deleted.

DEBUG_DEF . . . . . Calls the debug_ent method, which is used to produce neatly formatted, human readable information about the contents of the entity in the debug file. Place actions that write out useful information from the attribute for debugging purposes immediately following this macro.

Creating an Organization Attribute Class

Topic: *Attributes, *Extending ACIS, *Examples

Each application developer must create an organization class from which to derive all application specific attributes. This class cannot be instantiated. Its purpose is to uniquely identify the owning development organization. This section provides an example of creating an organization attribute class by copying the .hxx and .cxx files that define an existing ACIS organization attribute class and modifying those files.

Header File

Topic: Attributes

To create an organization attribute class header file:

1. Name the attribute organization class ATTRIB_<*sentinel*> where <*sentinel*> is a unique two- or three-letter code obtained from *Spatial*'s customer support department. In this example, the sentinel "ABC" is used.

2. Copy the at_tsl.hxx file (the header file for the ATTRIB_TSL class) from the ACIS installation directory and rename it at_abc.hxx.

3. Change the copied file as shown in the following example. Replace all the bold names in the file (**tsl** or **TSL**) with the new organization class's sentinel. In this example, the sentinel "ABC" (shown in bold italics as ***abc*** or ***ABC***) is used.

Note *In this example, the organization attribute class is defined in the KERN module (where* ATTRIB_TSL *is defined. To define a class in a different module, additional changes to the file would be needed.*

Example 3-1 of the attribute header file shows the sentinels that are to be replaced in **bold** font. In the new file, the new sentinel is shown in ***bold italic*** font.

C++ Example

```
// Attribute declaration for a private container attribute. Each
// application developer receives one of these customized for
// their use.
// All attributes specific to the application developer are then
// made derived classes of this attribute, ensuring that
// different developers can assign identifiers independently
// without mutual interference.

#if !defined(ATTRIB_TSL_CLASS)
#define ATTRIB_TSL_CLASS

#include "kernel/dcl_kern.h"
#include "kernel/kerndata/attrib/attrib.hxx"

extern DECL_KERN int ATTRIB_TSL_TYPE;
#define ATTRIB_TSL_LEVEL (ATTRIB_LEVEL + 1)

MASTER_ATTRIB_DECL( ATTRIB_TSL, KERN )
#endif
```

Example 3-1. Original at_tsl.hxx

After customizing at_tsl.hxx into at_abc.hxx, the organization class file should appear similar to Example 3-2.

C++ Example

```
#if !defined(ATTRIB_ABC_CLASS)
#define ATTRIB_ABC_CLASS

#include "kernel/dcl_kern.h"
#include "kernel/kerndata/attrib/attrib.hxx"

extern int ATTRIB_ABC_TYPE;
#define ATTRIB_ABC_LEVEL (ATTRIB_LEVEL + 1)

MASTER_ATTRIB_DECL( ATTRIB_ABC, KERN )
#endif
```

Example 3-2. Customized at_abc.hxx

Macro MASTER_ATTR_DECL completes the file. The first argument is the new organization attribute class name; the second is the module name.

Implementation File

Topic: Attributes

To create an organization attribute class .cxx implementation file:

1. Copy the at_tsl.cxx file from the ACIS installation directory and rename it at_abc.cxx. (If you do not have the source file, enter the content from this example into a file.)

2. Change the bold sentinels as was done for the at_abc.hxx header file.

3. Enter statements to include the header files necessary to define the attribute.

Note *File* kernel/acis.hxx *should always be the first header file included in every ACIS application* .cxx *implementation file.*

Example 3-3 of the attribute implementation file shows the sentinels that are to be replaced in **bold** font. In the new file, the new sentinel is shown in ***bold italic*** font.

C++ Example

```
#include <stdio.h>
#include ″kernel/acis.hxx″
#include ″kernel/dcl_kern.h″
#include ″kernel/kerndata/attrib/at_tsl.hxx″
#include ″kernel/kerndata/data/datamsc.hxx″

// Define macros for this attribute and its parent, to provide
// the information to the definition macro.

#define THIS() ATTRIB_TSL
#define THIS_LIB KERN
#define PARENT() ATTRIB
#define PARENT_LIB KERN

// Identifier used externally to identify a particular entity
// type. This is only used within the save/restore system for
// translating to/from external file format, but must be unique
// amongst attributes derived directly from ATTRIB, across all
// application developers.

#define ATTRIB_TSL_NAME ″tsl″

MASTER_ATTRIB_DEFN( ″tsl master attribute″ )
```

Example 3-3. Original at_tsl.cxx

The macro MASTER_ATTRIB_DEFN generates all the code necessary to define the organization attribute class. The character string is the value returned by the type_name method. After customizing .at_tsl.cxx into .at_abc.cxx, the organization class file should appear similar to Example 3-4.

C++ Example

```
#include <stdio.h>
#include ″kernel/acis.hxx″
#include ″kernel/dcl_kern.h″
#include ″kernel/kerndata/data/datamsc.hxx″
#include ″at_abc.hxx″

// Define macros for this attribute and its parent, to provide
// the information to the definition macro.

#define THIS() ATTRIB_ABC
#define THIS_LIB NONE
#define PARENT() ATTRIB
#define PARENT_LIB KERN

// Identifier used externally to identify a particular entity
// type. This is only used within the save/restore system for
// translating to/from external file format, but must be unique
// amongst attributes derived directly from ATTRIB, across all
// application developers.

#define ATTRIB_ABC_NAME ″abc″

MASTER_ATTRIB_DEFN( ″abc master attribute″ )
```

Example 3-4. Customized at_abc.cxx

Creating a Specific Attribute Class

Topic: *Attributes, *Extending ACIS, *Examples

After the organization attribute class (which cannot be instantiated) has been defined, specific attribute classes (which can be instantiated) may be derived from it.

Derived Simple Attribute

Topic: Attributes

The example in this section shows the new class ATTRIB_COL, which is derived from the organization class ATTRIB_ABC. ATTRIB_COL contains an integer that denotes one of eight colors. The attribute is attached to a body.

Color Attribute Header File

Topic: Attributes

To create a header (color_attrib.hxx) file for ATTRIB_COL:

1. Enter the #if !defined . . . #endif preprocessor commands to prevent duplicating the declaration.
2. Define the symbol ATTRIB_COL_CLASS.
3. Include the organization attribute class's header file.
4. Declare the global, dynamically assigned integer identifier for this attribute class.
5. Declare the identification level of this data type to be one level removed from the organization attribute class.
6. Declare that ATTRIB_COL is derived from ATTRIB_ABC.
7. Declare an integer that defines the color. (This is the only data for the class.)
8. Enter a constructor to create a color attribute for a given color and attach it to a given entity. A constructor that takes zero arguments must be provided for the save/restore functions. The constructor shown in the following example takes zero arguments because all arguments are defaulted. This is the recommended way of providing both the zero argument and standard constructor, rather than writing multiple constructors.
9. Create the member access (get) and setting (set) functions.
10. Declare the methods for split_owner and merge_owner. (Other common methods, such as trans_owner, are not overloaded in this example.)
11. Call the ATTRIB_FUNCTIONS macro to declare the functions required to complete the declaration of an attribute. The macro includes declarations for all ENTITY virtual functions, including make_copy, identity, type_name, size, and debug_ent. The macro also includes the declarations for the functions to save, restore, and copy the attribute.

Example 3-5 shows what what a color attribute file (color_attrib.hxx) could look like.

C++ Example

```
#if !defined(ATTRIB_COL_CLASS)
#define ATTRIB_COL_CLASS

#include "at_abc.hxx"

extern int ATTRIB_COL_TYPE;
#define ATTRIB_COL_LEVEL (ATTRIB_ABC_LEVEL + 1)

class ATTRIB_COL: public ATTRIB_ABC {
    int color_data;

public:
```

```
    ATTRIB_COL(ENTITY* = NULL, int = 0);

    int color() const {return color_data;}
    void set_color(int);

// Functions called to aid attribute migration during modeling
// operations.

    virtual void split_owner(ENTITY*);

// The owner of this attribute is about to be split in two – the
// argument is the new piece. The owner of this attribute is
// about to be merged with the given entity. The logical argument
// is TRUE if the owner is to be deleted in the merge.

    virtual void merge_owner(
        ENTITY*,              // ”other entity”
        logical               // deleting_owner
    );

    ATTRIB_FUNCTIONS(ATTRIB_COL, KERN);
};

#endif
```

Example 3-5. color_attrib.hxx Color Attribute Header File

Color Attribute Implementation File

Topic: Attributes

To create a color attribute implementation (color_attrib.cxx) file:

1. Include necessary system header files.
2. Include the header file declaring this specific attribute.
3. Include the header files necessary to define new entities.
4. Define THIS, THIS_LIB, PARENT, and PARENT_LIB macros to make subsequent definitions of this attribute and its parent easier.
5. Define the identifier used externally to identify a particular entity type.
6. Call macros used to implement the standard attribute methods and data. (These macros are discussed elsewhere in this chapter.)
7. Write any additional functions that were declared in the .hxx file.
    a. Define the constructor. The example calls the ATTRIB constructor to set the owning entity pointer then sets the member data.

b. Define the function to set the member data.

c. Define the virtual function called when the owner entity splits. The new face has no color, so the color attribute is duplicated onto the new face.

d. Define the virtual function called when two entities merge. In the merge function, several possibilities exist. If the two entities have the same color, the result has that color. If the entities have different colors, the result takes the color of the owning entity not deleted. If only the deleted entity has a color, that color is assigned to the result.

Example 3-6 shows what what a color attribute implementation file (color_attrib.cxx) could look like.

C++ Example

```
#include <stdio.h>
#include <memory.h>
#include ″kernel/acis.hxx″
#include ″kernel/kerndata/data/datamsc.hxx″
#include ″color_attrib.hxx″

// Implementation of color attribute. This is attached to body,
// face or edge objects, and is used to demonstrate attribute
// migration

// Define macros for this attribute and its parent, to simplify
// later stuff and make it suitable for making into macros.
#define THIS() ATTRIB_COL
#define THIS_LIB NONE
#define PARENT() ATTRIB_ABC
#define PARENT_LIB NONE

// Identifier used externally to identify a particular entity
// type. This is used in the save/restore system for translating
// to/from external file format.
#define ATTRIB_COL_NAME ″color″

// Implement the standard attribute functions.
ATTRIB_DEF(″color_attribute″)
```

```
    // Define color names for printout
    static char* col_name[] = {
        ”black”,
        ”red”,
        ”green”,
        ”blue”,
        ”cyan”,
        ”yellow”,
        ”magenta”,
        ”white”
    };
    debug_string(”color”, col_name [color_data], fp);

SAVE_DEF
    write_int(color_data);
    // Save specific data

RESTORE_DEF
    set_color(read_int());

COPY_DEF
    set_color(from–>color());

SCAN_DEF
    // (no specific pointer data)

FIX_POINTER_DEF
    // (no specific pointer data)

TERMINATE_DEF

// make a color attribute
ATTRIB_COL::ATTRIB_COL(
    ENTITY* owner,
    int col
    ) : ATTRIB_ABC(owner)

{
    // Initialize members.
    color_data = col; }

// Set the member data.
void ATTRIB_COL::set_color(
    int new_col
    )
{
    backup();
    color_data = new_col;
}
```

```
// Virtual function called when an owner entity is being split.
void ATTRIB_COL::split_owner(
    ENTITY* new_ent
    )
{
    // Duplicate the attribute on the new entity
    new ATTRIB_COL(new_ent, color_data);
}

// Virtual function called when two entities are to be merged.
void ATTRIB_COL::merge_owner(
    ENTITY* other_ent,
    logical delete_owner
    )
{
    // If the owner of this attribute is going to be deleted, and
    // there is no color attached to the other entity, then we
    // transfer to that other entity.
    if (delete_owner) {
        ATTRIB* other_att = find_attrib(
            other_ent,
            ATTRIB_ABC_TYPE,
            ATTRIB_COL_TYPE
        );
        if(other_att == NULL) {
            // No color on other entity, so transfer ourselves
            move(other_ent);
        }
    }
}
```

Example 3-6. color_attrib.cxx Color Attribute Implementation File

Example Color Attribute Application

Topic: Attributes

Example 3-7 shows an application program that uses the color attribute. The program initializes ACIS, makes a block, and applies a color attribute to the block. The program then writes out a debug file and a SAT file.

C++ Example

```
#include <stdio.h>
#include <stdlib.h>
#include ”kernel/acis.hxx”
#include ”kernel/logical.h”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kernapi/api/kernapi.hxx”
#include ”kernel/kerndata/lists/lists.hxx”
#include ”kernel/kerndata/top/body.hxx”
#include ”kernel/kernutil/vector/position.hxx”
#include ”constrct/kernapi/api/cstrapi.hxx”
#include ”kernel/kerndata/data/debug.hxx”
#include ”color_attrib.hxx”

void check_outcome (outcome result, char *string);
logical initialize_acis_components();
logical terminate_acis_components();

void main()
{
    // Initialization of the modeler must be done
    // before any other calls.
    outcome result = api_start_modeller( 0 );
    check_outcome(result, ”Error starting modeler”);

    // Call the main library initialization routine.
    // This routine is meant to initialize the various ACIS
    // libraries that will be linked in.
    initialize_acis_components();

    // Create positions for geometry elements
    SPAposition pts[2];
    pts[0] = SPAposition( 10, 10, 10);
    pts[1] = SPAposition( 15, 20, 30);

    BODY* my_block;

    // Create a solid block.
    result = api_solid_block( pts[0], pts[1], my_block);
    check_outcome(result, ”Error solid block”);


    BODY* b1;
    api_make_cuboid(10, 10, 10, b1);
```

```
    // Apply a color attribute of type 1, "red" to the block.
    new ATTRIB_COL (my_block, 1);
    // Write out the debug data file
    FILE* fp = fopen("Example3-7.dbg", "w");
    debug_entity(my_block, fp);
    fclose(fp);

    // Write out the "SAT" data file
    FILE* save_file = fopen("Example3-7.sat", "w");
    ENTITY_LIST slist;
    slist.add(my_block);
    api_save_entity_list(save_file, TRUE, slist);
    fclose(save_file);

    terminate_acis_components();
    printf("Program ended without error.\n");
}

void check_outcome(outcome result, char* string){
    if (result.ok())
        return;
    printf("Error %s\n", string);
    printf("Error %d – %s\n",result.error_number(),
         find_err_mess( result.error_number() ));
    exit(1);
}

logical initialize_acis_components(){
    logical result = TRUE;
    result &= api_initialize_kernel().ok();
    result &= api_initialize_constructors().ok();
    return result;
}

logical terminate_acis_components(){
    logical result = TRUE;
    result &= api_terminate_kernel().ok();
    result &= api_terminate_constructors().ok();
    return result;
}
```

Example 3-7. Example Application Program

Derived Complex Attribute

Topic: Attributes

Complex attributes contain pointers that reference other entities. When an attribute contains pointers, copying a body requires some special operations.

The following example modifies the previous “simple color attribute” example to implement a “complex color attribute.” This attribute must ensure that the copy, save, and restore functions are conducted correctly. The complex color attribute in this example contains a pointer to the body that possesses the face.

Complex Attribute Header File

Topic: Attributes

To create the .hxx file for the complex color attribute example, the following steps need to be completed in addition to those described for creating the simple color attribute .hxx file (refer to Example 3-6):

1. Add a pointer to the owning body in this declaration.
2. Add an initialization for the body pointer with a default NULL value to the constructor.
3. Add prototypes for functions that get and set the body pointer.

Except for the addition of the body pointers, Example 3-8 is identical to Example 3-5. The changes are shown in bold.

C++ Example

```
// User attribute class definition for a color.

#if !defined(ATTRIB_COL_CLASS)
#define ATTRIB_COL_CLASS
#include ”at_abc.hxx”
extern int ATTRIB_COL_TYPE;
#define ATTRIB_COL_LEVEL (ATTRIB_ABC_LEVEL + 1)

class ATTRIB_COL: public ATTRIB_ABC {
    int color_data;
    BODY* my_body;

public:

    ATTRIB_COL(ENTITY* = NULL, int = 0, BODY* = NULL);

    int color() const {return color_data;}
    void set_color(int);

    BODY* the_body() const { return my_body; }
    void set_the_body(BODY*);

// Functions called to aid attribute migration during modeling
// operations.
```

```
    virtual void split_owner(ENTITY*);

// The owner of this attribute is about to be split in two – the
// argument is the new piece. The owner of this attribute is
// about to be merged with the given entity. The logical argument
// is TRUE if the owner is to be deleted in the merge.

    virtual void merge_owner(
        ENTITY*,              // ”other entity”
        logical               // deleting_owner
    );

    ATTRIB_FUNCTIONS (ATTRIB_COL, KERN);
};
#endif
```

Example 3-8. Complex Color Attribute Header File

Complex Attribute Implementation File

Topic: Attributes

ACIS debugging functions that print entity information to a specified debug file are used with this example. They are declared in the file kerndata/data/debug.hxx. The function debug_old_pointer formats an entity pointer into a string for printing, but does not place the entity in the debug list, since it assumes it is already there. The string contains the decimal value of the pointer and an integer index for the pointer. The index is used to cross-reference it in the debug printout. This example uses debug_old_pointer because the body is probably already on the debugging list. This assumption is valid since, when debugging a body, the attribute will be printed to the debug file after the body is printed.

To create the .cxx file for the complex color attribute example, the following changes need to be made to the simple color attribute .cxx file:

1. Output the body pointer to the debug file using debug_old_pointer.

2. Save the body pointer in the save file by adding it to the entity list and writing the index returned during the list.add to the file. When the file is restored, the attribute is reconstructed with the correct body pointer. (list is an argument to save_common, the function defined by SAVE_DEF.)

3. Add a body pointer to RESTORE_DEF. When a pointer is restored, it is read as an integer index that must be cast into a pointer of the appropriate type. The next phase of part restoration converts the index into a valid pointer.

4. Add a body pointer in the COPY_DEF section. The from pointer is an argument to the copy_common function that points to the attribute being copied. If the attribute contains pointers to other attributes or entities, the pointer in the copy must be set to an integer index returned from doing a lookup of the original pointer in the ENTITY_LIST list. The integer is cast into the appropriate type for assignment to the pointer in the copy that is later converted to the correct pointer. The ENTITY_LIST list is an argument to the copy_common function.

5. Insert pointer data in the SCAN_DEF section. The SCAN_DEF macro defines the function copy_scan. When a body is being copied, all data is scanned to find all pointers and convert them into indices. If the attribute contains any pointers, they must be added to the ENTITY_LIST list, which is an argument to the function.

6. Insert pointer data in the FIX_POINTER_DEF section. The final stage of restoring a body from a file, or of copying the body, is to convert all indices stored in pointer data fields into real pointers. FIX_POINTER_DEF is a macro that defines the function, fix_pointers. The read_array function looks up the integer index in the ENTITY array and returns the corresponding ENTITY pointer.

7. Add the body pointer to the constructor.

8. Add the my_body initialization to the constructor.

9. Add a function to set the body pointer.

Except for the addition of the body pointers, Example 3-9 is identical to Example 3-6 . The changes are shown in bold.

C++ Example

```
// Implementation of color attribute
#include <stdio.h>
#include <memory.h>
#include ″kernel/acis.hxx″
#include ″color_attrib.hxx″
#include ″kerndata/data/datamsc.hxx″

// Define macros for this attribute and its parent, to simplify
// later stuff and make it suitable for making into macros.
#define THIS() ATTRIB_COL
#define THIS_LIB NONE
#define PARENT() ATTRIB_ABC
#define PARENT_LIB NONE

// Identifier used externally to identify a particular entity
// type. This is used in the save/restore system for translating
// to/from external file format.
#define ATTRIB_COL_NAME ″color″
```

```
// Implement the standard attribute functions.
ATTRIB_DEF(”color_attribute”)
    // Define color names for printout
    static char* col_name[] = {
        ”black”,
        ”red”,
        ”green”,
        ”blue”,
        ”cyan”,
        ”yellow”,
        ”magenta”,
        ”white”
    };

    debug_string(”Color”, col_name[color_data], fp);
    debug_old_pointer(”Body pointer”, (ENTITY*)my_body, fp);

SAVE_DEF
    write_int(color_data);     // Save specific data
    write_ptr((ENTITY*)my_body, list);

RESTORE_DEF
    set_color(read_int());
    set_the_body((BODY*)read_ptr());

COPY_DEF
    set_color(from->color());
    set_the_body((BODY*)list.lookup((ENTITY*)(from->the_body())))
;

SCAN_DEF
    list.add((ENTITY*)my_body);

FIX_POINTER_DEF
    my_body = (BODY*)read_array(array, (int)my_body);

TERMINATE_DEF

// make a color attribute

ATTRIB_COL::ATTRIB_COL(
    ENTITY* owner,
    int col,
    BODY* the_body
) : ATTRIB_ABC(owner)
{
    // Initialize members.
    color_data = col;
    my_body = the_body;
}
```

```
// Set the member data.
void ATTRIB_COL::set_color(
    int new_col
    )
{
    backup();
    color_data = new_col;
}

// Set the body function.
void ATTRIB_COL::set_the_body(
    BODY* the_new_body
    )
{
    backup();
    my_body = the_new_body;
}

// Virtual function called when an owner entity is being split.
void ATTRIB_COL::split_owner(
    ENTITY* new_ent
    )
{
    // Duplicate the attribute on the new entity
    new ATTRIB_COL(new_ent, color_data);
}

// Virtual function called when two entities are to be merged.
void ATTRIB_COL::merge_owner(
    ENTITY* other_ent,
    logical delete_owner   )
{
    // If the owner of this attribute is going to be deleted, and
    // there is no color attached to the other entity, then we
    // transfer ourselves to that other entity.
        if(delete_owner) {
        ATTRIB* other_att = find_attrib(
            other_ent,
            ATTRIB_ABC_TYPE,
            ATTRIB_COL_TYPE
        );
```

```
        if(other_att == NULL) {
            // No color on other entity, so transfer ourselves
            move(other_ent);
        }
    }
}
```

Example 3-9. Complex Color Attribute color_attrib.cxx File

Complex Attribute Boolean Conditions

Topic: *Attributes

The split and merge functions do not address all changes to complex attributes. In Figure 3-6, assume that both faces *A* and *B* have instances of the same complex attribute. During a unite of the “tool” body containing face *A* with the “blank” body containing face *B*, the attribute on *B* is notified of a split. However, face *A* does not participate in the Boolean because it is neither split nor merged. Therefore, after the Boolean, the face *A* body pointer is still pointing at the now deleted tool body. This must be repaired after the Boolean.

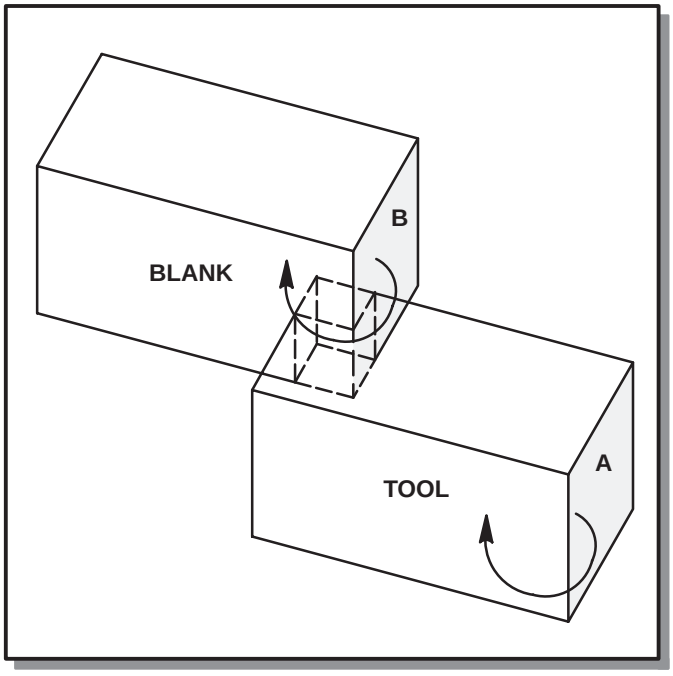

Figure 3-6. Complex Attribute

Because the virtual functions do not solve all of these problems, the programmer must be aware that the application may have to do extra work. The application may have to scan the resulting body to determine whether attributes have been updated, and update them where necessary.

For example, assume you have two blocks, and each block uses a simple attribute to identify each FACE. Now subtract one block (the tool), as shown in the figure below. This subtraction creates three new FACEs in the blank.

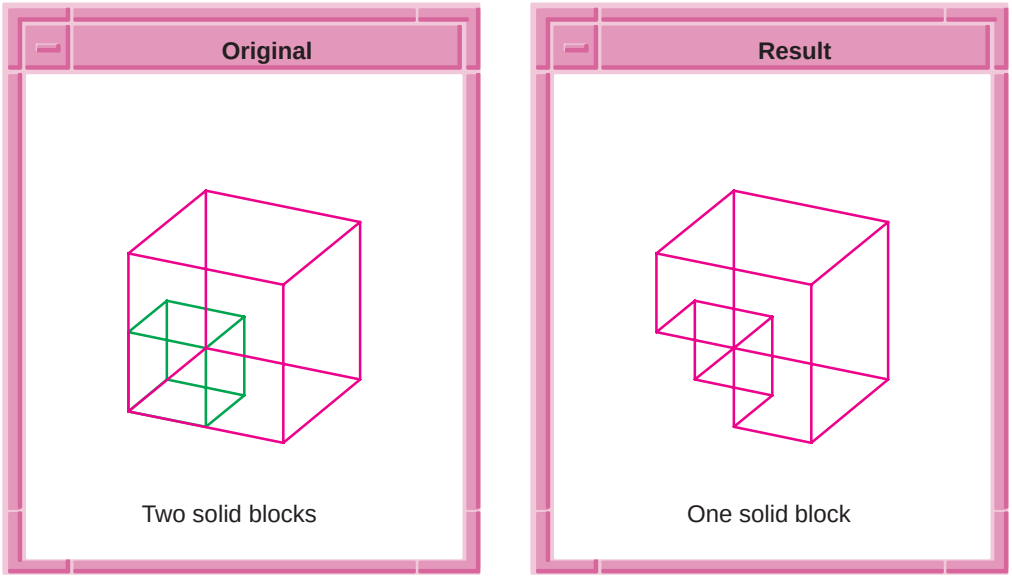

Figure 3-7. bool:subtract

During a Boolean operation, the blank and tool faces are split where they intersect. Each split creates a new face and one or more edges. By default, the new faces have no attributes. Later in the Boolean, containment is determined and the faces which are not part of the result are lost. There is no guarantee which faces will be lost. In some cases it will be the original and in others the faces created by the split.

So in this example, if you need to have the attributes which are tagged to the tool FACEs automatically be transferred to the corresponding new FACEs created in the blank, it can not be done directly. To do that, you would override the split_owner method of the attributes. In that method, make a copy of the attribute and put it on the face being created by the split.